NIKLAUS WIRTH

ALGORITMY A ŠTRUKTÚRY ÚDAJOV

2. vydanie 1989

alfa

VYDAVATEĽSTVO TECHNICKEJ A EKONOMICKEJ LITERATÚRY
BRATISLAVA

OBSAH

Predslov 11

Prehlásenie 18

1 Základné štruktúry údajov 19

1.1 Úvod 19
1.2 Typ údajov 22
1.3 Primitívne typy údajov 26
1.4 Štandardné primitívne typy 28
1.5 Typ interval 31
1.6 Štruktúra pole 32
1.7 Štruktúra záznam 38
1.8 Varianty štruktúr záznam 43
1.9 Štruktúra množina 47
1.10 Reprezentácia štruktúr pole, záznam a množina 53
1.10.1 Reprezentácia polí 54
1.10.2 Reprezentácia štruktúr záznam 59
1.10.3 Reprezentácia množín 60
1.11 Sekvenčný súbor 61
1.11.1 Základné operátory typu súbor 65
1.11.2 Štruktúrovanie súborov 68
1.11.3 Texty 71
1.11.4 Program na editovanie súboru 80

2 Triedenie 90

2.1 Úvod 90
2.2 Triedenie polí 93
2.2.1 Triedenie priamym vkladaním 94
2.2.2 Triedenie priamym výberom 99

2.2.3 Triedenie priamou výmenou . . . 101
2.2.4 Triedenie vkladaním so zmenšovaním kroku (Shellovo triedenie) . . . 106
2.2.5 Stromové triedenie . . . 109
2.2.6 Triedenie rozdeľovaním . . . 116
2.2.7 Hľadanie mediána . . . 124
2.2.8 Porovnanie metód vnútorného triedenia . . . 126
2.3 Triedenie sekvenčných súborov . . . 129
2.3.1 Priame zlučovanie . . . 129
2.3.2 Prirodzené zlučovanie . . . 137
2.3.3 Vyvážené viaccestné zlučovanie . . . 146
2.3.4 Polyfázové triedenie . . . 153
2.3.5 Distribúcia začiatočných behov . . . 167

3 Rekurzívne algoritmy . . . 180

3.1 Úvod . . . 180
3.2 Kedy nepoužívať rekurziu . . . 183
3.3 Dva príklady rekurzívnych programov . . . 187
3.4 Algoritmy prehľadávania s návratom . . . 195
3.5 Problém ôsmich dám . . . 203
3.6 Problém stabilného manželstva . . . 209
3.7 Problém optimálneho výberu . . . 218

4 Dynamické informačné štruktúry . . . 227

4.1 Rekurzívne typy údajov . . . 227
4.2 Smerníky alebo referencie . . . 231
4.3 Lineárne zoznamy . . . 238
4.3.1 Základné operácie . . . 238
4.3.2 Usporiadané zoznamy a reorganizované zoznamy . . . 244
4.3.3 Aplikácia: Topologické triedenie . . . 254
4.4 Stromové štruktúry . . . 263
4.4.1 Základné pojmy a definície . . . 263
4.4.2 Základné operácie na binárnych stromoch . . . 274
4.4.3 Vyhľadávanie a pridávanie do stromov . . . 279
4.4.4 Rušenie vrcholov v strome . . . 290
4.4.5 Analýza vyhľadávania a pridávania . . . 292
4.4.6 Vyvážené stromy . . . 296
4.4.7 Pridávanie do vyvážených stromov . . . 298
4.4.8 Rušenie vrcholov vo vyvážených stromoch . . . 305
4.4.9 Optimálne vyhľadávacie stromy . . . 310
4.4.10 Zobrazenie stromovej štruktúry . . . 318
4.5 Viaccestné stromy . . . 331

4.5.1 B-stromy . . . 335
4.5.2 Binárne B-stromy . . . 349
4.6 Transformácie kľúča . . . 358
4.6.1 Voľba transformačnej funkcie . . . 360
4.6.2 Ošetrovanie kolízií . . . 361
4.6.3 Analýza transformácie kľúča . . . 367

5 Jazykové štruktúry a kompilátory . . . 378

5.1 Definícia a štruktúra jazyka . . . 378
5.2 Analýza vety . . . 382
5.3 Konštrukcia syntaktického grafu . . . 388
5.4 Konštrukcia syntaktického analyzátora pre danú syntax . . . 392
5.5 Konštrukcia programu syntaktickej analýzy riadeného tabuľkou . . . 397
5.6 Prekladač z BNF do štruktúr údajov . . . 401
5.7 Programovací jazyk PL/0 . . . 411
5.8 Syntaktický analyzátor jazyka PL/0 . . . 416
5.9 Zotavenie sa zo syntaktických chýb . . . 426
5.10 Procesor jazyka PL/0 . . . 440
5.11 Generovanie cieľového kódu . . . 444

Prílohy . . . 464

A. Množina znakov ASCII . . . 464
B. Syntaktické diagramy jazyka pascal . . . 465

Zoznam programov . . . 475

Register . . . 477

PREDSLOV

Počas uplynulého desaťročia sa z programovania stala vedná disciplína, ktorej dokonalé ovládanie zohráva základnú a rozhodujúcu úlohu pri riešení mnohých technických projektov. Súčasne sa ukázalo, že túto disciplínu možno podrobiť systematickému vedeckému spracovaniu. Z remesla sa teda stal nový vedný odbor. Prvé významné práce týkajúce sa vývoja programovania vytvorili E. W. DIJKSTRA a C. A. R. HOARE. Dijkstra prostredníctvom *Poznámok k štruktúrovanému programovaniu* [0-1] predpovedá programovaniu, ako predmetu vedy, nové méty, ba dokonca „vyzýva k jeho súboju s intelektom". Začína sa hovoriť o „revolúcii" v programovaní. Hoare vo svojom *Axiomatickom základe programovania* [0-2] zase jasne ukázal, že programy je možné podrobiť exaktnej analýze založenej na matematických axiómach. Z oboch uvedených prác vyplýva, že pri programovaní sa dá predísť viacerým chybám, ak programátor pozná a vhodne používa tie metódy a techniky, ktoré dosiaľ intuitívne a často nevedomky aplikoval. Tieto práce sa zameriavajú na problematiku kompozície a analýzu programov alebo, presnejšie povedané, na štruktúru algoritmov reprezentovaných programovými textami. Zrejme, že systematický a vedecký prístup k tvorbe programov má najväčší význam pri veľkých a zložitých programoch narábajúcich s komplikovanými štruktúrami údajov. Z tohto aspektu metodológia programovania úzko súvisí s problematikou štruktúr údajov. Napokon programy sú konkrétnymi stvárneniami abstraktných algoritmov opierajúcich sa o určité reprezentácie a štruktúry údajov. Ďalšou významnou prácou v oblasti štruktúr údajov sú nepochybne Hoarove *Poznámky k štruktúrovaniu údajov* [0-1], v ktorých sa snažil,

a úspešne, vniesť poriadok do chaotickej terminologickej a koncepčnej mnohotvárnosti štruktúr údajov. Hoare s opodstatnením tvrdí, že pri programovaní nie je možné uvažovať o štruktúrach údajov oddelene, bez zreteľa na algoritmy manipulujúce s týmito údajmi, a obrátene, že štruktúra a výber vhodných algoritmov často silne závisia od vyskytujúcich sa štruktúr údajov. Stručne povedané: problematiky kompozície programu a štruktúrovania údajov sú neoddeliteľne späté.

Skutočnosť, že táto kniha začína kapitolou o *štruktúrach údajov*, má dva dôvody. Prvým je akýsi intuitívny pocit prioritnejšieho postavenia údajov oproti algoritmom: najprv musíme mať nejaké objekty a až potom môžeme na nich aplikovať určité operácie. Po druhé, (a to je pravá príčina) o čitateľovi tejto knihy sa predpokladá, že pozná základy programovania. Jednako však v úvodných kurzoch programovania sa lektori obyčajne zameriavajú na algoritmy manipulujúce iba s jednoduchými štruktúrami údajov. Preto rozsiahlejšia úvodná kapitola zaoberajúca sa problematikou štruktúr údajov má svoje opodstatnenie a bude zaiste prínosom.

V celej knihe, a najmä v prvej kapitole, sa budeme pridŕžiavať Hoarovej teórie a terminológie [0-1], ktoré boli aplikované v programovacom jazyku *pascal* [0-3]. Podstata tejto teórie spočíva v tom, že údaje predstavujú najprv abstrakcie skutočných javov. Vystupujú teda najskôr ako abstraktné štruktúry, ktoré sa v bežných programovacích jazykoch nedajú realizovať (a ani ich nie je potrebné vždy realizovať). V priebehu procesu tvorby programu sa reprezentácia údajov postupne zjemňuje. Takéto zjemňovanie prebieha krok za krokom súbežne so zjemňovaním použitých algoritmov. Pritom v rámci každého kroku sa reprezentácia údajov stále viac a viac prispôsobuje obmedzeniam vyplývajúcim z konkrétneho programovacieho prostredia [0-4]. Preto pred ďalšími úvahami uvedieme niekoľko základných princípov štruktúrovania údajov. Teória definuje a vychádza zo *základných štruktúr*. Od týchto sa požaduje, aby boli ľahko implementovateľné na existujúcich počítačoch. Iba v takomto prípade ich môžeme pokladať za skutočné elementárne prvky reprezentácie údajov, za *molekuly*, získané v poslednom kroku zjemňovacieho procesu. Sú to *záznam* (**record**), *pole* (**array**) a *množina* (**set**). Nie je prekvapením, že tieto základné molekuly zodpovedajú elementárnym matematickým pojmom.

Základným princípom teórie štruktúr údajov je rozlišovanie medzi základnými a vyššími štruktúrami. Základné štruktúry — molekuly — pozostávajú z atómov a slúžia ako stavebné prvky na vytváranie vyšších štruktúr údajov. Premenné základných štruktúr menia iba svoju hodnotu, a nikdy nie svoju štruktúru a množinu hodnôt, ktoré môžu nadobúdať. Preto veľkosť pamäti, ktorú zaberajú, ostáva konštantná. Vyššie štruktúry sú naproti tomu charakteristické tým, že v priebehu výpočtu programu menia svoju hodnotu a štruktúru. Na ich implementáciu sú preto potrebné náročnejšie techniky. Sekvenčné súbory údajov, alebo jednoducho povedané postupnosti, vystupujú v uvedenej klasifikácii ako „hybridy". Dĺžka súborov je síce premenlivá, ale táto zmena neovplyvňuje ich štruktúru. Pretože sekvenčný súbor zohráva pri programovaní skutočne jednu z najvýznamnejších úloh, bol zahrnutý medzi základné štruktúry prvej kapitoly tejto knihy.

Druhá kapitola sa zaoberá *algoritmami triedenia*. Rozoberá množstvo rozličných metód, ktoré slúžia na ten istý účel. Matematické analýzy niektorých algoritmov triedenia ukazujú výhody a nevýhody príslušných metód. Súčasne upozorňujú programátora na dôležitosť analýzy algoritmov pri voľbe správnej stratégie riešenia daného problému. Rozdelenie metód triedenia na triedenie polí a triedenie súborov (často nazývané tiež vnútorné a vonkajšie triedenie) ukazuje rozhodujúci vplyv reprezentácie údajov na výber vhodných algoritmov a ich zložitosť. Triedeniu je v tejto knihe vyhradený pomerne veľký priestor, a to hlavne z toho dôvodu, že práve triedenie v sebe sklbuje mnohé princípy programovania a situácie, ktoré sa často vyskytujú aj v iných aplikáciách. Často sa nám dokonca zdá, že by bolo jednoducho možné zrealizovať kompletný kurz programovania iba na základe výberu príkladov z oblasti triedenia.

Ďalšou témou, ktorej sa obyčajne nedostáva miesto v úvodných kurzoch programovania, hoci hrá významnú úlohu pri koncipovaní riešenia mnohých algoritmických problémov, je *rekurzia*. Z toho dôvodu je tretia kapitola venovaná *rekurzívnym algoritmom*. Na konkrétnych príkladoch bude ukázané, že rekurzia je vlastne zovšeobecnením iterácie. Ako taká je dôležitým a významným konceptom pri programovaní. V mnohých programovacích kurzoch však, žiaľ, rekurziu ukazujú na takých príkladoch, v ktorých by bolo použitie iterácie

výhodnejšie. Preto sa tretia kapitola sústreďuje na viaceré príklady tých problémov, ktorých riešenie sa dá pomocou rekurzie vyjadriť pekne a prirodzene. Použitie iteračných algoritmov v takýchto prípadoch by viedlo k neprehľadným a ťažkopádnym programom. Ideálnym prípadom použitia rekurzie sú algoritmy, pri ktorých sa používa *sledovanie s návratom* (tzv. backtracking). Najprirodzenejším použitím rekurzie sú však algoritmy, ktoré manipulujú s rekurzívne definovanými štruktúrami údajov. Tieto prípady budeme preberať v poslednej kapitole, tretia kapitola je však užitočným úvodom do tejto problematiky.

Štvrtá kapitola sa zaoberá *dynamickými štruktúrami údajov*. Sú to také údaje, ktorých štruktúra sa v priebehu výpočtu programu mení. Ukážeme, že rekurzívne štruktúry údajov sú významnou podtriedou všeobecne používaných dynamických štruktúr údajov. Aj keď je rekurzívna definícia v týchto prípadoch prirodzená, a aj možná, predsa sa v bežnej programátorskej praxi zvyčajne nepoužíva. Namiesto nej sa pri implementácii používa mechanizmus, ktorý núti programátora používať explicitné referencie a premenné typu *smerník* (ukazovateľ). V knihe berieme ohľad na túto prax a podávame jej súčasný obraz: Štvrtá kapitola sa zaoberá programovaním so smerníkmi, zoznamami, stromami a prezentuje príklady s komplikovanejšími štruktúrami údajov. Ukazuje aj (trochu nevhodne nazývané) spracovanie zoznamov. Značný priestor je vyhradený stromovým štruktúram a ich prehľadávaniu. Kapitola sa končí problematikou rozptylových tabuliek, pri ktorých sa uplatňuje vyhľadávanie pomocou transformácie kľúča. Táto technika sa často uprednostňuje pred prehľadávacími stromami. Týmto sa súčasne naskytá možnosť porovnať dve rôzne techniky s pomerne častými aplikáciami.

Posledná, piata kapitola obsahuje stručný úvod do problematiky *formálnych jazykov* a *syntaktickej analýzy*. Uvedieme proces tvorby kompilátora malého a jednoduchého programovacieho jazyka určeného pre jednoduchý počítač. Túto kapitolu zaraďujeme do knihy z troch dôvodov: Po prvé, úspešný programátor by mal mať aspoň základné vedomosti o podstatných problémoch a technikách procesu prekladu programovacieho jazyka. Po druhé, počet aplikácií, ktoré vyžadujú definíciu jednoduchého vstupu alebo riadiaceho jazyka pre príslušné operácie, stále narastá. Po tretie, formálne jazyky definujú rekurzívnu

štruktúru nad postupnosťami symbolov; ich procesory sú preto výborným príkladom užitočnosti aplikácie rekurzívnych techník, ktoré sú rozhodujúce pre získanie prehľadnej štruktúry v tých prípadoch, keď programy začínajú byť enormne veľké. V rámci kapitoly priblížime jednoduchý programovací jazyk, nazývaný PL/0. Vybrali sme ho preto, že predstavuje určitý kompromis medzi jazykom, ktorý by bol pre našu demonštráciu príliš jednoduchý, a jazykom, ktorého kompilátor by veľkosťou svojich programov zaiste prekročil rozumný rozsah tejto knihy, (ktorá, samozrejme, nie je určená iba pre špecialistov na tvorbu kompilátorov).

Programovanie je konštruktívna činnosť. Ako je možné naučiť sa takejto tvorivej činnosti? Jednou možnosťou by bolo „vykryštalizovanie" základných princípov tvorby programov z mnohých prípadov a ich systematické aplikovanie. Programovanie je však rozsiahly a mnohotvárny odbor vyžadujúci často komplexné intelektuálne činnosti.

Predstava, že programovanie by bolo možné zhrnúť do výuky receptov, je mylná. Z množstva učebných metód nám zostane iba starostlivý výber a prezentácia vzorových príkladov. Pochopiteľne, nemožno predpokladať, že každý človek je rovnako schopný získať bohaté vedomosti z riešenia príkladov; vo väčšine prípadov sa ráta so študentovou usilovnosťou a intuíciou. To platí obzvlášť pri relatívne náročných a dlhých príkladoch programov. Zaradenie takýchto programov do tejto knihy teda nebolo celkom náhodné. Dlhšie programy sú v programátorskej praxi bežným zjavom a sú aj vhodnejšie na demonštrovanie náročnej, ale podstatnej zložky programovania, nazývanej štýl a metodická štruktúra. Súčasne slúžia i ako cvičenia k „umeniu čítať" programy, ktoré sa obyčajne zanedbáva v prospech „umenia písať" programy. Toto je podstatný dôvod na zahrnutie veľkých celistvých príkladov programov. Čitateľ bude mať možnosť sledovať proces tvorby programu, viaceré závažné fázy vývoja programu, pričom tento vývoj bude postupovať v intenciách metódy postupného zjemňovania detailov.

Osobne považujem za dôležité ukázať programy vo svojom konečnom tvare s dostatočným prihliadnutím na detaily, pretože pri programovaní sú práve detaily veľmi dôležité. Aj keď pre vedecky mysliaceho človeka by zrejme stačilo uviesť princíp algoritmu a jeho matematickú

analýzu s vylúčením technických detailov, praktikovi by toto zrejme nestačilo. Preto som sa rozhodol pridržiavať sa prísne pravidla, že programy vo svojej konečnej podobe budú napísané v takom jazyku, v ktorom sa už dajú spracovať na počítači. Tým však vzniká problém nájsť takú formu zápisu programu, v ktorej program už môže byť spracovaný na počítači, a súčasne, aby tento jazyk bol natoľko počítačovo nezávislý, ako sa od učebnice očakáva. Z tohto aspektu nevyhovovali ani bežne používané programovacie jazyky, ani rôzne abstraktné notácie. Vhodným kompromisom sa ukázal byť programovací jazyk *pascal*; bol vyvinutý presne na uvedené účely, a preto sa používa i v tejto knihe. Programátori, ktorí poznajú iné vyššie programovacie jazyky, ako *algol 60*, alebo *PL/I*, ľahko porozumejú programom napísaným v jazyku pascal. To však neznamená, že nie je potrebná žiadna príprava. Kniha „Systematické programovanie" [0-5] je ideálny základ, pretože v nej sa takisto používa zápis v jazyku pascal. Predkladaná kniha však nie je príručkou jazyka pascal, na tento účel slúži iná vhodnejšia literatúra [0-6].

Táto kniha je zbierkou a súčasne spracovaním látky z rôznych kurzov programovania, ktoré sa uskutočňovali na Vysokej škole technickej (ETH) v Zürichu. Chcel by som týmto vyjadriť vďaku mojim spolupracovníkom na ETH za mnohé myšlienky a názory vyjadrené v tejto knihe. Takisto sa chcem zmieniť o podnetnom vplyve pracovných stretnutí s členmi IFIP-skupiny 2.1 a 2.3 a najmä o plodných diskusiách s E. W. DIJKSTROM a C. A. R. HOAROM. Na záver by som nechcel zabudnúť vyjadriť svoju vďaku i ETH za veľkorysé poskytnutie výpočtovej techniky, bez ktorej by vydanie tejto knihy nebolo možné. Všetkým tým, ktorí priamo alebo nepriamo prispeli k realizácii tejto knihy patrí moja úprimná vďaka.

N. Wirth

Zoznam použitej literatúry

0-1. DAHL, O. J. — DIJKSTRA, E. W. — HOARE, C. A. R.: Structured Programming. New York, Academic Press (1972).
0-2. HOARE, C. A. R.: Axiomatic Basis of Computer Programming. Comm. ACM, 12, No. 10, (1969), s. 576—583.
0-3. WIRTH, N.: The Programming Language Pascal. Acta Informatica. No. 1, (1971), s. 35—63.
0-4. WIRTH, N.: Program Development by Stepwise Refinement. Comm. ACM 14, No. 4, (1971), s. 221—227.
0-5. WIRTH, N.: Systematic Programming. Englewood Cliffs, N. J.: Prentice-Hall Inc. (1973). (Slovenský preklad: Systematické programovanie. Bratislava, Alfa/SNTL 1981.)
0-6. JENSEN, K. — WIRTH, N.: PASCAL-Unser Manual and Report. Lecture Notes in Computer Science, Vol. 18. Berlin, New York, Springer-Verlag (1974).

PREHLÁSENIE

Náš generálporučík L. Euler vydáva oficiálne prehlásenie, v ktorom sa otvorene priznáva:

III. Že on, nesmrteľný fénix matematikov sa vždy zapýri pri pomyslení na prečin proti zdravému rozumu a zaužívaným predstavám, ktorého sa dopustil formulovaním záverov zo svojich vzorcov. Usúdil totiž, že teleso sa po vychýlení vracia späť do pôvodnej polohy.

IV. Že vynaloží všetko svoje úsilie na to, aby sa v budúcnosti jeho myšlienky uberali správnym smerom a nezvádzali ho k mylným tvrdeniam. Už vôbec netúži po tom, aby sa pod vplyvom podobných chybných uzáverov dostal do rozporu s nemeckými filozofmi, ku ktorým prechováva svoju úctu. Prosí preto, na kolenách sa kajúci, logikov o prepáčenie za vetu, ktorú napísal ako dôsledok jedného paradoxu:
„I keď sa to prieči realite, musíme viac veriť našim výpočtom ako zdravému rozumu."

V. Že v budúcnosti sa bude vždy pridržiavať pravidla:
„Dvakrát meraj a raz strihaj!" Preto, keď si najbližšie vysúka rukávy s cieľom počítať tri dni a tri noci, tak najprv štvrťhodinku dôkladne podumá nad najvhodnejšou výpočtovou metódou a až potom sa pustí do počítania. A tak výsledok, ku ktorému by predtým dospel po 60-tich stranách úmorného rátania, si dokáže odvodiť i na 10-tich riadkoch.

Vyňaté a voľne preložené z:
François-Marie Arouet Voltaire:
Hanopis doktora Akakiu
(Diatribe du docteur Akakia) November 1752

1 ZÁKLADNÉ ŠTRUKTÚRY ÚDAJOV

1.1 ÚVOD

Moderný číslicový počítač bol vyvinutý na uľahčenie a zrýchlenie zložitých a najmä časovo náročných výpočtov. Vo väčšine prípadov použitia počítača zohráva rozhodujúcu úlohu jeho schopnosť zapamätať si a znovu sprístupniť množstvo informácií. Táto vlastnosť počítača je snáď najcharakteristickejšou črtou. Naproti tomu schopnosť počítača počítať, vykonávať rôzne aritmetické a iné výpočty, sa v mnohých prípadoch stala takmer bezvýznamnou. Vo všetkých uvedených prípadoch veľké množstvá informácií, ktoré je potrebné spracovať, predstavujú určitú abstrakciu reálneho sveta. Informácia vstupujúca do počítača pozostáva z vybranej množiny údajov reálneho sveta, o ktorej sa domnievame, že je podstatná a súčasne postačujúca na vyriešenie daného problému. Údaje predstavujú *abstrakciu* určitej reality i v tom zmysle, že isté vlastnosti a charakteristiky reálnych objektov, ktoré sú však pri riešení problému vedľajšie a bezvýznamné, sa zanedbávajú. Abstrakcia je tým vlastne i zjednodušenie skutočnosti. Ako príklad zoberme kartotéku zamestnancov nejakého podniku. Každý zamestnanec je v tejto kartotéke reprezentovaný (abstrahovaný) množinou údajov, ktoré sú dôležité pre evidenciu a účtovníctvo jeho zamestnávateľa. Tieto údaje obsahujú niektoré identifikačné informácie o pracovníkovi, napr. jeho meno, priezvisko a plat. S najväčšou pravdepodobnosťou sa však medzi týmito údajmi nebudú vyskytovať také informácie, akými sú napr. farba vlasov, váha či výška pracovníka. Tieto informácie sú pre zamestnávateľa bezvýznamné. Pri riešení konkrétneho problému, či už pomocou počítača, alebo bez neho, je dôležité zvoliť abstrakciu skutočnosti, to znamená definovať množinu údajov reprezentujúcu reálnu situáciu. Tento výber sa musí uskutočniť so zreteľom na riešený

problém. Až potom nasleduje voľba reprezentácie vybraných informácií. V tejto etape je už potrebné brať do úvahy aj zariadenie, na ktorom sa budú vybrané informácie spracúvať. Vo väčšine prípadov výber relevantných informácií a voľba ich reprezentácie nie sú celkom nezávislé fázy. *Výber reprezentácie údajov* je pomerne dosť náročný a nebýva vždy určený iba možnosťami prístupného počítača. Pri údajoch treba mať na mysli vždy aj operácie, ktoré sa budú s danými údajmi vykonávať. Dobrým príkladom je reprezentácia čísel, ktoré už samy o sebe predstavujú abstrakcie charakterizovaných vlastností objektov. V prípade, že sčítanie je jedinou (alebo aspoň prevládajúcou) operáciou, ktorá sa má realizovať, vhodným spôsobom ako zobraziť číslo n je zapísať n čiarok. Pravidlo sčítania pre takéto zobrazenie je zrejmé a triviálne; rímske číslice sú postavené na rovnakom princípe a sčítacie pravidlá sú pre malé čísla jednoduché. Zobrazenie arabskými číslicami však vyžaduje pravidlá, ktoré (pre malé čísla) nie sú zrejmé a je potrebné sa ich naučiť. Opačná je však situácia, ak ide o sčítanie veľkých čísel alebo o ich násobenie, či delenie. Dekompozícia týchto operácií na jednoduchšie je pri zobrazeniach arabskými číslicami oveľa jednoduchšia vzhľadom na ich systematický princíp štruktúrovania, ktorý je postavený na pozičnom význame jednotlivých číslic. Je známe, že počítače manipulujú s vnútornou reprezentáciou, založenou na dvojkových čísliciach (bitoch). Takéto zobrazenie je pre človeka nevhodné vzhľadom na veľké množstvo obsiahnutých číslic, ale veľmi výhodné je pre elektronické obvody, pri ktorých sa hodnoty 0 a 1 dajú ľahko a spoľahlivo zobraziť pomocou prítomnosti alebo neprítomnosti elektrického prúdu, elektrického náboja či magnetického poľa. Z tohto príkladu vidieť, že otázka reprezentácie často presahuje niekoľko detailov. Ak máme napr. zobraziť pozíciu nejakého objektu, môžeme si pri prvom rozhodnutí zvoliť dvojicu reálnych čísel v karteziánskych alebo polárnych súradniciach. Druhé rozhodnutie môže viesť k zobrazeniu v pohyblivej rádovej čiarke, kde sa každé reálne číslo x skladá z dvojice $\langle f, e \rangle$ celých čísel. Číslo f predstavuje zlomkovú časť a e je exponent nad určitým základom (napr.: $x = f \,.\, 2^e$). Tretie riešenie vychádza z poznatku, že údaje budú uložené v pamäti počítača, čo môže viesť k dvojkovému pozičnému zobrazeniu celých čísel. Napokon finálne rozhodnutie môže znieť takto: zobraziť dvojkové číslice prostredníctvom smeru

magnetického toku v magnetickej pamäti. Prirodzene, že prvé rozhodnutie je ovplyvnené stanoveným problémom, kým na ďalšie dve riešenia má už podstatný vplyv počítačové zariadenie a jeho technológia. Od programátora však nemožno chcieť, aby rozhodoval o zobrazení použitých čísel alebo dokonca o vlastnostiach pamäťových zariadení. Tieto „rozhodnutia na najnižšej úrovni" môžu byť ponechané tvorcom počítačov, ktorí sú najlepšie informovaní o použitej technológii a na jej základe sú schopní zvoliť najvhodnejšie rozhodnutie prijateľné pre všetky (alebo pre takmer všetky) aplikácie, v rámci ktorých zohrávajú čísla významnú úlohu. V tejto súvislosti sa stáva očividným zmysel *programovacích jazykov*. Programovací jazyk predstavuje abstraktný počítač, ktorý je schopný porozumieť výrazom tohto jazyka. Termíny programovacieho jazyka zobrazujú určitý stupeň abstrakcie objektov spracúvaných skutočným strojom. Z tohto dôvodu sa programátor, používajúci takýto vyšší jazyk, vyhne otázke zobrazenia čísel (navyše informácia o použitom zobrazení čísel je mu nedostupná) v prípade, že číslo je už elementárnym objektom v rámci tohto jazyka.

Použitie jazyka, ktorý poskytuje dostatočnú množinu základných abstrakcií pre väčšinu riešených problémov spracovania údajov, je vzhľadom na spoľahlivosť vytváraných programov veľmi dôležité. Jednoduchšie je vytvoriť program, v ktorom sa manipuluje s pojmami, ako sú čísla, množiny, postupnosti alebo cykly, ako taký, v ktorom sa používajú pojmy bity, slová alebo skoky. Samozrejme, počítač zobrazuje napokon všetky údaje, či už sú to čísla, množiny alebo postupnosti, prostredníctvom dvojkových číslic (bitov). Tento fakt je však pre programátora bezvýznamný, pokiaľ sa nemusí zaujímať o detaily reprezentácie jeho vybraných abstrakcií a pokiaľ si môže byť istý, že zodpovedajúce, počítačom (alebo kompilátorom) zvolené reprezentácie sú správne a optimálne pre jeho ciele. Čím bližšie sú abstrakcie k danému počítaču, tým jednoduchšie je pre inžiniera alebo tvorcu kompilátora urobiť výber zobrazenia pre príslušný jazyk a tým väčšia bude pravdepodobnosť, že urobený výber bude vyhovujúci pre všetky (alebo skoro všetky) mysliteľné aplikácie. Táto realita určuje pevné medze stupňa abstrakcie od skutočného počítača. Bolo by zrejme nezmyselné, zaradiť medzi základné údajové prvky všeobecne použiteľného programovacieho jazyka geometrické objekty, pretože ich presné zobrazenie (vzhľadom na

ich prirodzenú zložitosť) je veľmi závislé od operácií manipulujúcich s týmito objektmi. Tvorca univerzálne použiteľného jazyka a jeho kompilátora nebude nič vedieť o povahe a frekvencii týchto operácií, a preto hocijaký výber zobrazenia ním zvolený môže byť nevyhovujúci pre niektoré z možných aplikácií.

Tieto úvahy stanovili formu zápisu algoritmov a ich údajov. Pochopiteľne, že radšej chceme využívať známe matematické pojmy, ako sú čísla, množiny, postupnosti atď., a nie počítačovo závislé termíny, ako sú napr. bitové reťazce. Okrem toho chceme používať takú formu zápisu, pre ktorú už skutočne existujú použiteľné kompilátory. Napokon je nerozumné použiť silne počítačovo orientovaný a počítačovo závislý jazyk, alebo obrátene, opisovať počítačové programy abstraktnou formou s úplným zanedbaním otázky reprezentácie.

Programovací jazyk *pascal* bol vyvinutý za účelom nájdenia kompromisu medzi týmito extrémami, a preto ho používame v celej knihe [1-3], [1-5]. Tento jazyk bol s úspechom implementovaný na viacerých typoch počítačov a ukázalo sa, že forma zápisu je natoľko blízka reálnemu počítaču, že zvolené prvky a ich reprezentácie sa dajú zrozumiteľne vyjadriť. Jazyk je príbuzný s niektorými inými jazykmi, najmä však s jazykom *algol 60*, a to až natoľko, že lekcie, ktoré budeme v ďalšom preberať, sú priamo aplikovateľné v prostredí jazyka algol 60.

1.2 TYP ÚDAJOV

V matematike je obvyklé klasifikovať premenné podľa určitých význačných vlastností. Dôkladne sa odlišujú reálne premenné od komplexných a logických premenných alebo premenné nadobúdajúce jednoduché hodnoty od tých, ktoré môžu nadobúdať množiny hodnôt či množiny množín. Podobne sa diferencuje medzi funkciami, funkcionálmi, množinami funkcií atď. Takýto spôsob klasifikácie je rovnako dôležitý, ak nie ešte viac, pri spracúvaní údajov. Budeme sa vždy pridŕžiavať princípu, že každá *konštanta*, *premenná*, *výraz* alebo *funkcia* sú určitého typu. Tento typ určuje množinu hodnôt, do ktorej daná konštanta prináleží, alebo ktoré môže daná premenná či výraz nadobudnúť, alebo ktoré možno danou funkciou vypočítať. V prípade matematických zápisov je často možné určiť typ premennej z typu použitého

písma bez detailnejšieho skúmania celého kontextu. Pri počítačových programoch však táto možnosť nie je, pretože počítač obyčajne disponuje iba jedným druhom písma (latinské písmená). Preto sa prijalo pravidlo, že typ konštanty, premennej alebo fukcie sa explicitne stanoví v rámci *deklarácie* príslušného objektu a že najprv musí byť daná konštanta, premenná alebo funkcia deklarovaná (musí byť určený jej typ) a až potom použitá v programe. Toto pravidlo je obzvlášť pochopiteľné, ak zvážime skutočnosť, že kompilátor musí zvoliť reprezentáciu príslušného objektu v pamäti počítača. Prirodzene, veľkosť pamäti pre príslušnú premennú sa vyhradzuje podľa veľkosti rozsahu hodnôt, ktoré môže daná premenná nadobúdať. Pokiaľ má kompilátor možnosť získať takéto informácie, odpadá nevyhnutnosť dynamického vyhradzovania pamäti, čo je často kľúčová otázka efektívnosti použitého algoritmu. Podstatné vlastnosti pojmu *typ údajov*, ako ho budeme v rámci celej knihy používať a ako sa chápe i v programovacom jazyku *pascal*, sa dajú zhrnúť takto [1-2]:

1. Typ údajov určuje množinu hodnôt, do ktorej daná konštanta prináleží, alebo ktoré môže daná premenná či výraz nadobudnúť, alebo ktoré môže daným operátorom alebo funkciou vypočítať.

2. Typ hodnoty konštanty, premennej alebo výrazu sa dá určiť z ich deklarácie alebo zápisu bez nevyhnutnosti uskutočniť vlastný výpočtový proces programu.

3. Každý operátor alebo funkcia predpokladá argumenty presne definovaných typov a poskytuje výsledok tiež presne definovaného typu. V prípade, že operátor pripúšťa argumenty rôznych typov (napr. ak + sa používa na sčítanie celých i reálnych čísel), dá sa typ výsledku určiť podľa špecifických pravidiel príslušného programovacieho jazyka.

Z uvedeného vyplýva, že kompilátor môže informácie získané z použitých typov v rámci deklarácií rôznych objektov využiť pri kontrole kompatibility a prípustnosti jazykových konštrukcií. Napríklad priradenie boolovskej (logickej) hodnoty aritmetickej (reálnej) premennej sa dá zistiť bez realizácie programu. Tento druh redundancie v programoch je veľmi užitočnou pomôckou v rámci procesu tvorby programu a treba ho pokladať za jednu z primárnych predností vyšších programovacích jazykov oproti strojovým kódom (alebo jazykom symbolických inštrukcií). Prirodzene, že v konečnom dôsledku budú údaje repre-

zentované veľkým počtom binárnych číslic bez ohľadu na to, či bol program pôvodne napísaný v jazyku vyššej úrovne, zahrňujúcom pojem typu údajov, alebo v beztypovom jazyku symbolických inštrukcií. Pokiaľ ide o počítač, je jeho pamäť iba obyčajnou homogénnou masou bitov bez viditeľnej štruktúry. Jednako však práve táto bitová masa je jedinou abstraktnou štruktúrou, ktorá umožňuje programátorovi nachádzať zmysel a orientáciu v monotónnej počítačovej pamäti.

Teória uvádzaná v tejto knihe a programovací jazyk *pascal* špecifikujú určité metódy definovania typov údajov. Vo väčšine prípadov sa nové typy údajov definujú pomocou už definovaných typov. Hodnoty takéhoto typu sú obyčajne konglomerátmi *hodnôt prvkov* už definovaných typov. Hovoríme im *štruktúrované typy*. V prípade, že všetky prvky sú toho istého typu, to znamená, že existuje iba jeden typ prvku, potom sa takýto typ nazýva *základný*.

Počet možných hodnôt typu T sa nazýva *kardinalita* typu T. Kardinalita poskytuje mieru veľkosti pamäti potrebnej na reprezentáciu premennej x, ktorá je typu T, čo sa vyjadruje zápisom

$$x : T$$

Vzhľadom na to, že typy jednotlivých prvkov štruktúry môžu byť štruktúrované, je možné vytvárať celé hierarchie štruktúr. Pochopiteľne, prvotné prvky celej štruktúry už nesmú byť deliteľné (šruktúrované). Preto je dôležité, aby existoval nejaký spôsob zápisu takýchto primitívnych, neštruktúrovaných typov. Jednou priamočiarou metódou je *vymenovanie* hodnôt tvoriacich daný typ. V programe manipulujúcom s jednoduchými geometrickými útvarmi, môžeme zadefinovať napr. primitívny typ tvar, ktorého hodnoty označíme identifikátormi *obdĺžnik*, *štvorec*, *elipsa* a *kruh*. Okrem takýchto typov definovaných programátorom musia existovať aj nejaké *štandardné typy*, o ktorých hovoríme, že sú preddefinované. Takéto typy obyčajne zahŕňajú *čísla* a *logické hodnoty*. V prípade, že medzi individuálnymi hodnotami existuje usporiadanie, potom o danom type hovoríme, že je *usporiadaný*, alebo ho nazývame *skalárnym*. V jazyku *pascal* sa všetky neštruktúrované typy považujú za usporiadané, v prípade explicitného vymenovania hodnôt je usporiadanie dané poradím hodnôt v postupnosti. Takto možno definovať primitívne typy a vytvárať konglomeráty, t. j.

štruktúrované typy s ľubovoľnou hĺbkou vnorenia. V praxi však nevystačíme iba s jednou všeobecnou metódou vytvárania štruktúrovaných typov. Vzhľadom na problémy spojené s reprezentáciou a použitím v praxi, musí univerzálny programovací jazyk poskytovať rozličné *metódy štruktúrovania*. V matematickom zmysle môžu byť všetky ekvivalentné, môžu sa líšiť iba v operátoroch umožňujúcich vytváranie ich hodnôt a výber prvkov štruktúry týchto hodnôt.

Základnými metódami štruktúrovania sú **array** (pole), **record** (záznam), **set** (množina) a **postupnosť** (súbor). Zložitejšie štruktúry sa obyčajne nedefinujú ako statické typy, ale v priebehu výpočtu programu sa vytvárajú dynamicky; pričom sa môže meniť ich veľkosť a tvar. O takýchto štruktúrach budeme hovoriť vo štvrtej kapitole, v ktorej rozoberáme problematiku zoznamov, cyklických grafov, stromov a všeobecných konečných grafov.

Premenné a typy údajov zavádzame do programu za účelom ich použitia počas realizácie programu. Okrem nich potrebujeme množstvo *operátorov*. Podobne, ako v prípade typu údajov, programovacie jazyky poskytujú istý počet primitívnych, štandardných operátorov a metód štruktúrovania, na základe ktorých je možné vytvárať zložitejšie operácie. Otázka štruktúrovania operácií sa často považuje za kľúčový problém programátorského umenia. V každom prípade je jasné, že vhodná kompozícia údajov je rovnako základná a dôležitá otázka.

Najdôležitejšie základné operátory sú *porovnanie* a *priradenie*, t. j. *test rovnosti* (a usporiadania v prípade usporiadaných typov) a *príkaz zavedenia rovnosti*. Základný rozdiel medzi týmito operáciami je v ďalšom texte zvýraznený ich odlišným označením (hoci to tak nie je pri tak široko používaných programovacích jazykoch, akými sú *fortran* a *PL/1*, ktoré používajú znak rovnosti ako operátor priradenia).

Test na rovnosť: $x = y$

Priradenie premennej x: $x := y$

Tieto základné operátory sú definované pre všetky typy údajov. Treba však pamätať na to, že ich vykonávanie môže byť často časovo náročné, najmä ak ide o veľké a hlboko štruktúrované údaje.

Okrem testu na rovnosť (alebo usporiadania) a priradenia existujú ešte základné implicitne definované operátory, ktoré sa nazývajú *prevodové operátory*. Zobrazujú isté typy údajov do iných typov údajov

a obzvlášť dôležité sú pri štruktúrovaných typoch údajov. Štruktúrované hodnoty sa vytvárajú z hodnôt prvkov štruktúry prostredníctvom *konštruktorov*. Prístup k hodnotám prvkov štruktúry sa uskutočňuje pomocou *selektorov*. Konštruktory a selektory sú takto prevodovými operátormi, zobrazujúcimi typy prvkov do štruktúrovaných typov a obrátene. Každá metóda štruktúrovania má svoju dvojicu konštruktorov a selektorov, ktoré sa jasne odlišujú na základe svojich zápisov.

Štandardné primitívne typy údajov vyžadujú množinu štandardných primitívnych operátorov. Preto spolu so štandardnými typmi údajov, ako sú čísla a logické hodnoty, zavádzame aj konvenčné operácie z aritmetiky a výrokovej logiky.

1.3 PRIMITÍVNE TYPY ÚDAJOV

V mnohých programoch sa na reprezentáciu údajov používajú celé čísla, aj keď sa na týchto údajoch nevykonávajú žiadne numerické operácie, alebo tieto údaje môžu nadobudnúť iba malý počet hodnôt z možných alternatív. Na tento účel zavedieme nový primitívny, neštruktúrovaný typ údajov T vymenovaním množiny všetkých možných hodnôt $c_1, c_2, \ldots, c_n$.

$$\textbf{type } T = (c_1, c_2, \ldots, c_n) \tag{1.1}$$

Kardinalitu T vyjadrujeme takto: $\text{card}(T) = n$.

Príklady:

type tvar = (obdĺžnik, štvorec, elipsa, kruh)
type farba = (červená, žltá, zelená)
type pohlavie = (mužské, ženské)
type boolean = (false, true)
type deň = (pondelok, utorok, streda, štvrtok, piatok, sobota, nedeľa)
type mena = (frank, marka, libra, dolár, šiling, líra, gulden, koruna, rubel cruzeiro, yen)
type miesto určenia = (peklo, očistec, nebo)

type dopravný prostriedok = (vlak, autobus, automobil, loď, lietadlo)
type hodnosť = (vojak, slobodník, desiatnik, čatár, podporučík, poručík, nadporučík, kapitán, major, podplukovník, plukovník, generál)
type objekt = (konštanta, typ, premenná, proces, funkcia)
type štruktúra = (file, array, record, set)
type stav = (ručný, nezavedený, paritný, chybný)

Definíciou takýchto typov sa nezavádza iba nový identifikátor typu, ale súčasne i množina identifikátorov označujúcich hodnoty nového typu. Tieto identifikátory sa potom v programe dajú použiť ako konštanta, čo zlepšuje zrozumiteľnosť programu. Ak napríklad zavedieme premenné s, d, r, b

var s: pohlavie
var d: deň
var r: hodnosť
var b: boolean

tak sú prípustné tieto príkazy priradenia:

s : = mužské
d: = nedeľa
r : = major
b: = true

Pochopiteľne, že takéto zápisy dávajú viac informácií, ako ich možné alternatívne formulácie

$$s := 1 \quad d := 7 \quad r := 9 \quad b := 2$$

ktoré vychádzajú z predpokladu, že c, d, r, b sú typu *integer* a že konštanty sú reprezentované prirodzenými číslami, usporiadanými v súlade s ich vymenovaním. Okrem toho kompilátor môže kontrolovať oprávnenosť použitia aritmetických operátorov pri údajoch takýchto numerických typov, ako napr. v príkaze

$$s := s + 1$$

V prípade, že uvažujeme o nejakom type ako o usporiadanom, je užitočné zaviesť funkcie, ktoré vracajú ako hodnotu nasledovníka a predchodcu svojich argumentov. Takéto funkcie budeme označovať succ(x) a pred(x). Usporiadanie hodnôt typu T je dané pravidlom

$$(c_i < c_j) \equiv (i < j) \tag{1.2}$$

1.4 ŠTANDARDNÉ PRIMITÍVNE TYPY

Štandardné primitívne typy sú vo väčšine existujúcich počítačov súčasťou ich základného programového vybavenia. Sú to celé čísla, logické pravdivostné hodnoty a množina vytlačiteľných znakov. Na väčších počítačoch bývajú k dispozícii aj reálne čísla so zodpovedajúcou množinou jednoduchých operátorov. Uvedené typy označíme identifikátormi

integer, boolean, char, real

Typ *integer* zahŕňa podmnožinu množiny celých čísel, ktorej veľkosť je rôzna pre rôzne typy počítačov. O všetkých operáciách vykonávaných nad údajmi tohto typu sa predpokladá, že sú presné a spĺňajú bežné aritmetické pravidlá. Okrem toho platí zásada, že výpočet programu sa preruší v okamihu, keď výsledok niektorej z operácií je mimo reprezentovanej podmnožiny. Štandardné operátory sú pre štyri základné aritmetické operácie: sčítanie (+), odčítanie (−), násobenie (∗) a delenie (**div**). Od poslednej operácie (delenie) sa očakáva celočíselný výsledok, prípadný zvyšok po delení sa zanedbáva, neuvažuje sa o ňom. Pre dve kladné čísla m, n teda platí:

$$m - n < (m \textbf{ div } n) * n < = m \tag{1.3}$$

Operátor „modulo" (**mod**) je definovaný prostredníctvom operácie delenia rovnicou

$$(m \textbf{ div } n) * n + (m \textbf{ mod } n) = m \tag{1.4}$$

Takto je m **div** n celočíselný podiel čísel m, n a m **mod** n je príslušný zvyšok.

Typ *real* označuje podmnožinu reálnych čísel. Zatiaľ čo pri celočíselnej aritmetike očakávame presné výsledky, pri aritmetike s hodnotami typu real pripúšťame nepresnosti v rámci limitu chýb spôsobených zaokrúhľovaním nadol. Chyby sú dôsledkom výpočtov vykonávaných s konečným počtom číslic. Toto je zásadný dôvod na presné rozlišovanie typov integer a real vo väčšine programovacích jazykov.

Operáciu delenia reálnych čísel, výsledkom ktorej je reálny podiel, označujeme symbolom (/) na rozdiel od operátora **div** pri delení celých čísel.

Štandardný typ *boolean* obsahuje dve hodnoty, ktoré sa označujú identifikátormi true a false. Boolovskými operátormi sú logický súčin (konjunkcia), logické zjednotenie (disjunkcia) a logická negácia, ktorých hodnoty sú definované v *tab. 1.1*. Logický súčin (konjunkcia) sa označuje symbolom ∧ (alebo **and**), logické zjednotenie (disjunkcia) symbolom ∨ (alebo **or**) a logická negácia symbolom ¬ (alebo **not**).

Tabuľka 1.1

p	q	p **or** q	p **and** q	**not** q
true	true	true	true	false
true	false	true	false	false
false	true	true	false	true
false	false	false	false	true

Poznamenávame, že porovnávania sú operátory, ktoré dávajú výsledok typu boolean. Takto môže byť výsledok porovnávania priradený nejakej premennej, alebo sa môže použiť ako operand logického operátora v boolovskom výraze. Ak sú napr. dané boolovské premenné p a q a celočíselné premenné $x = 5$, $y = 8$, $z = 10$, tak výsledkom príkazov

$$p := x = y$$
$$q := (x < y) \textbf{ and } (y < = z)$$

sú hodnoty p = false a q = true.

Štandardný typ *char* zahŕňa množinu znakov vytlačiteľných na danom počítači. Žiaľ, neexistuje univerzálna množina znakov, ktorá by

bola vytlačiteľná na všetkých typoch počítačov. Preto použitie predikátu štandardný je v tomto prípade klamlivé, musíme ho chápať ako štandardný typ pre počítač, na ktorom sa má realizovať výpočet programu. Najrozšírenejšou je množina znakov definovaná *Medzinárodnou organizáciou pre štandardizáciu* (ISO) a najmä jej americká verzia ASCII (*Americký štandardný kód pre výmenu informácií*). Množina ASCII znakov je z tohto dôvodu uvedená v prílohe A. Pozostáva z 95 vytlačiteľných znakov (grafických) a 33 riadiacich znakov, pričom riadiace znaky sa používajú predovšetkým pri prenosoch údajov a na ovládanie tlačiarne počítača. Veľmi rozšírenou je podmnožina 64 vytlačiteľných znakov (len veľkých písmen) — redukovaná množina ASCII znakov[1].

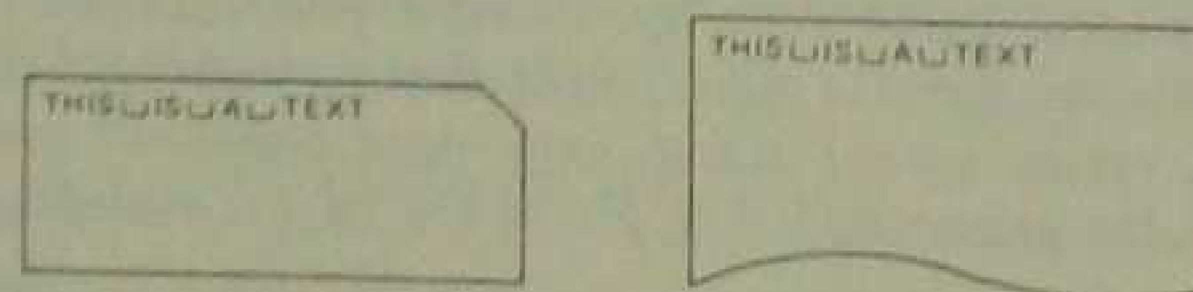
Aby sme mohli vytvárať algoritmy, ktoré manipulujú so znakmi (t. j. hodnotami typu char) a sú pritom počítačovo nezávislé, je potrebné stanoviť niekoľko záväzných zásad, týkajúcich sa množiny znakov:

1. Typ char obsahuje 26 latinských písmen, 10 arabských číslic a určitý počet iných grafických znakov, ako sú napr. interpunkčné znamienka.

2. Podmnožiny písmen a číslic sú usporiadané a súvislé, t. j.

$$\begin{aligned} &(\text{'A'} <= x) \textbf{ and } (x <= \text{'Z'}) \equiv x \text{ je písmeno} \\ &(\text{'0'} <= x) \textbf{ and } (x <= \text{'9'}) \equiv x \text{ je číslica} \end{aligned} \tag{1.5}$$

3. Typ char obsahuje vytlačiteľný prázdny znak, ktorý sa môže používať ako oddeľovač (prázdne znaky sú na *obr. 1.1* zobrazené pomocou symbolu ␣).



Obr. 1.1. Zobrazenie textu na diernom štítku a výstupe z tlačiarne počítača

[1] V ČSSR sú najrozšírenejšími 2 množiny znakov: na počítačoch radu SMEP je to práve spomenutá množina ASCII znakov a na počítačoch radu EC množina EBCDIC znakov (rozšírený dvojkovo kódovaný desiatkový výmenný kód) — pozn. prekl.

Existencia dvoch štandardných prevodových funkcií medzi typmi char a integer je veľmi dôležitá hlavne preto, aby sme mohli písať počítačovo nezávislé programy. Je to funkcia ord(c), ktorej argument c je typu char a výsledok je typu integer, určujúci poradové číslo znaku c v množine znakov char. Druhou funkciou je chr(i), ktorej argument i je celé číslo typu integer a výsledkom je i-tý znak z množiny char. Takto je chr inverzná funkcia vzhľadom na funkciu ord a obrátene, teda

$$\begin{aligned} &\text{ord}(\text{chr}(i)) = i \quad (\text{ak chr}(i) \text{ je definované}) \\ &\text{chr}(\text{ord}(c)) = c \end{aligned} \tag{1.6}$$

Obzvlášť významné sú funkcie

$$\begin{aligned} &f(c) = \text{ord}(c) - \text{ord}(\text{'0'}) = \text{pozícia znaku } c \text{ medzi číslicami} \\ &g(i) = \text{chr}(i + \text{ord}(\text{'0'})) = i\text{-tá číslica} \end{aligned} \tag{1.7}$$

Napríklad platí

$$f(\text{'3'}) = 3 \quad \text{a} \quad g(5) = \text{'5'}$$

kde f je inverzná funkcia vzhľadom na funkciu g a obrátene, t. j.

$$\begin{aligned} &f(g(i)) = i \quad (0 \leq i \leq 9) \\ &g(f(c)) = c \quad (\text{'0'} \leq c \leq \text{'9'}) \end{aligned} \tag{1.8}$$

Uvedené prevodové funkcie sa používajú pri prevode vnútorných reprezentácií čísel na postupnosti číslic a obrátene. Tieto funkcie skutočne vykonávajú konverzie na tej najelementárnejšej úrovni, a to prostredníctvom jednej číslice.

1.5 TYP INTERVAL

Často sa stáva, že nejaká premenná nadobúda hodnoty určitého typu, ale iba v rámci stanoveného intervalu. Túto skutočnosť možno vyjadriť tým, že danú premennú definujeme ako premennú typu *interval*. Definíciu potom možno napísať takto:

$$\textbf{type } T = \text{min} \,..\, \text{max} \tag{1.9}$$

Symboly min a max reprezentujú hranice intervalu.

Príklady:

type rok = 1900 .. 1999
type pismeno = 'A' .. 'Z'
type číslica = '0' .. '9'
type dôstojník = podporučík .. generálporučík

Nech sú definované premenné y a L:

var y: rok
var L: pismeno

Potom sú prípustné priradenia $y := 1973$ a $L :=$ 'W', zatiaľ čo priradenia $y := 1291$ a $L :=$ '9' nie sú prípustné. Kompilátor dokáže skontrolovať správnosť takýchto priraďovacích príkazov iba v tých prípadoch, keď priraďovaná hodnota je označená prostredníctvom konštanty alebo premennej toho istého typu. V ostatných prípadoch prípustnosť priraďovacích príkazov typu

$$y := i, \quad L := c$$

kde i je typu integer a c je typu char, sa dá skontrolovať iba počas realizácie programu. Z praktického hľadiska sú veľmi prospešné systémy, ktoré dokážu vykonávať takéto druhy kontrol; zvlášť výhodné sú pri vytváraní veľkých a zložitých programov. Využitie redundantných informácií pri zisťovaní možných chýb je opäť jednou z hlavných motivácií používania jazyka vyššej úrovne.

1.6 ŠTRUKTÚRA POLE

Pole je pravdepodobne najznámejšia štruktúra údajov, pretože v mnohých jazykoch, vrátane fortranu a algolu 60, je to jediná explicitne prístupná štruktúra. Pole je homogénna štruktúra; pozostáva z prvkov, ktoré sú všetky jediného typu. Tento typ prvku poľa sa nazýva bázový alebo *základný typ*. Pole pokladáme za *štruktúru s náhodným prístupom*; všetky prvky môžu byť vybraté náhodne a sú rovnako sprístupniteľné. Na referenciu individuálneho prvku poľa je potrebné použiť meno celej štruktúry rozšírené o *index*, určujúci vybratý prvok. Index nadobúda hodnoty typu, definovaného ako *typ indexu* poľa.

Definícia typu pole T špecifikuje základný typ T_0 a typ indexu I:

$$\textbf{type } T = \textbf{array } [I] \textbf{ of } T_0 \tag{1.10}$$

Príklady:

type riadok = **array** [1 .. 5] **of** real
type štítok = **array** [1 .. 80] **of** char
type alfa = **array** [1 .. 10] **of** char

Jednotlivé hodnoty prvkov premennej

var x: riadok

ktoré vyhovujú rovnici $x_i = 2^{-i}$, možno zobraziť ako na *obr. 1.2*.

x_1	0.5
x_2	0.25
x_3	0.125
x_4	0.0625
x_5	0.03125

Obr. 1.2. Pole typu riadok

Štruktúrovanú hodnotu x typu T s hodnotami prvkov $c_1, c_2, \ldots, c_n$ možno vyjadriť pomocou *konštruktora* poľa a priraďovacieho príkazu:

$$x := T(c_1, c_2, \ldots, c_n) \tag{1.11}$$

Inverzným operátorom ku konštruktoru poľa je *selektor*, ktorý slúži na vybratie jednotlivých prvkov poľa. Nech x je premenná typu pole. Potom selektor prvku poľa vyjadrujeme prostredníctvom mena poľa rozšíreného o príslušný index i prvku poľa:

$$x[i] \tag{1.12}$$

Najbežnejším spôsobom manipulácie so štruktúrami typu pole, najmä s rozsiahlymi poľami, je aktualizácia ich jednotlivých prvkov. Zriedkavo sa pri poliach stretávame s konštruovaním ich úplne nových štruktú-

rovaných hodnôt. Tento fakt sa vyjadruje tým, že sa premenná typu pole reprezentuje poľom premenných prvkov a umožňuje priradenie do vybratých prvkov.

Príklad:

$$x[i] := 0.125$$

Aj keď sa výberovou aktualizáciou zmení hodnota iba jedného prvku, z koncepčného hľadiska sa mení hodnota celej štruktúry poľa.

Skutočnosť, že indexy polí, t. j. „mená" prvkov polí, musia byť určitého definovaného (skalárneho) typu, je mimoriadne dôležitá. Indexy možno vypočítať; namiesto indexovej konštanty možno dosadiť indexový výraz, vyhodnotí sa a výsledok určuje vybraný prvok poľa. Táto univerzálnosť nielenže predstavuje jednu z najvýznamnejších a najvýkonnejších programovacích stratégií, no súčasne je zdrojom výskytu veľmi častých chýb pri programovaní: vypočítaná hodnota indexu môže byť mimo rozsahu definovaných prípustných hodnôt. My budeme predpokladať, že primerané výpočtové systémy sú schopné zistiť uvedenú chybu a formou varovnej správy signalizovať prístup k neexistujúcemu prvku poľa.

Obyčajne je typ indexu skalárom, t. j. neštruktúrovaným typom, na ktorom je definovaná relácia usporiadania. Ak je základný typ poľa tiež usporiadaný, je daná relácia prirodzeného usporiadania na celom type pole. Prirodzené usporiadanie dvoch polí je vyjadrené pomocou dvoch zodpovedajúcich nezhodných prvkov s najmenšími indexmi, čo sa dá formálne vyjadriť takto:

Ak sú dané dve polia x, y, sú v relácii $x < y$ vtedy a len vtedy, ak existuje index k taký, že platí

$$x[k] < y[k] \quad \text{a súčasne} \quad x[i] = y[i] \tag{1.13}$$

pre všetky $i < k$.

Napríklad:

(2, 3, 5, 7, 9) < (2, 3, 5, 7, 11)
'LABEL' < 'LIBEL'

Vo väčšine aplikácií sa na typoch pole nepredpokladá žiadne usporiadanie. Kardinalita štruktúrovaného typu je súčinom kardinalít jeho

prvkov. Vzhľadom na to, že všetky prvky typu poľa A sú toho istého základného typu B, dostávame, že

$$\text{kardinalita } (A) = (\text{kardinalita } (B))^n \tag{1.14}$$

kde n = kardinalita (I), pričom I je typ indexu poľa.

Nasledujúca časť programu ukazuje použitie selektora poľa. Cieľom programu je nájdenie najmenšieho indexu i prvku poľa s hodnotou x. Vyhľadávanie sa realizuje sekvenčným prezeraním poľa a:

```
var a: array [1 .. N] of T;   {N > 0}
i: = 0;
repeat i: = i + 1 until (a[i] = x) ∨ (i = N)                    (1.15)
if a[i] ≠ x        then „taký prvok" v poli neexistuje
```

Nasledujúci variant tohto programu využíva programovaciu techniku nažývanú *zarážka*, umiestnenú do $(N + 1)$-ého prvku poľa. Zmyslom použitia zarážky je zjednodušenie podmienky ukončenia cyklu.

```
var a: array [1 .. N + 1] of T;
i: = 0; a[N + 1]: = x;
repeat i: = i + 1 until a[i] = x;                                (1.16)
if i > N then „taký prvok v poli a neexistuje"
```

Priradenie $a[N + 1] := x$ je príkladom *výberovej aktualizácie*, t. j. zmeny hodnoty vybraného prvku štruktúrovanej premennej. Základnou podmienkou, ktorá platí bez ohľadu na to, koľkokrát sa opakoval príkaz $i := i + 1$, je

$$a[j] \neq x \quad \text{pre} \quad j = 1, 2, \ldots, i - 1$$

v obidvoch prípadoch (1.15) aj (1.16). Takúto podmienku nazývame *invariant cyklu*.

Vyhľadávanie sa dá, samozrejme, podstatnejšie urýchliť, ak prvky poľa sú už usporiadané (utriedené). V takomto prípade je najzaužívanejšou technikou opakované delenie intervalu na podintervaly, v ktorých sa hľadaný prvok hľadá. Tento spôsob vyhľadávania sa nazýva *bisekcia* alebo *binárne vyhľadávanie*. Jeho algoritmus uvádza (1.17). V rámci každého kroku cyklu sa rozpolí skúmaný interval, ktorý je

určený indexmi i, j. Počet potrebných porovnaní je preto najviac $\lceil \log_2(N) \rceil$.

$$
\begin{aligned}
&i := 1; j := N; \\
&\textbf{repeat } k := (i + j) \textbf{ div } 2; \\
&\quad \textbf{if } x > a[k] \textbf{ then } i := k + 1 \textbf{ else } j := k - 1 \\
&\textbf{until } (a[k] = x) \vee (i > j)
\end{aligned}
\tag{1.17}
$$

(Relevantnou invariantnou podmienkou pri vstupe príkazu cyklu je, aby

$$
\begin{aligned}
a[h] &< x \quad \text{pre} \quad h = 1, \ldots, i - 1 \\
a[h] &\geq x \quad \text{pre} \quad h = j + 1, \ldots, N
\end{aligned}
$$

Ak program skončí tým, že $a[k] \neq x$, tak v danom poli neexistuje hľadaný prvok $a[h] = x$ s $1 \leq h \leq N$.)

Základný typ poľa môže byť takisto štruktúrovaný. Premenná typu pole, ktorej prvky sú opäť typu pole, sa nazýva *matica*. Napríklad premenná

M: **array** [1 .. 10] **of** riadok

je pole pozostávajúce z desiatich prvkov (riadkov), z ktorých každý sa skladá z piatich prvkov typu real. Premennú M nazývame matica stupňa 10×5 s reálnymi prvkami. Selektory poľa je možné zreťaziť, takže

$M[i][j]$

označuje j-tý prvok riadka $M[i]$, čo je zase i-tý prvok štruktúry M. Uvedený zápis sa dá písať i v skrátenej forme:

$M[i, j]$

V podobnom zmysle možno aj deklaráciu poľa

M: **array** [1 .. 10] **of array** [1 .. 5] **of** real

zapísať v stručnejšom tvare:

M: **array** [1 .. 10, 1 .. 5] **of** real

V prípade, že určitá operácia sa má vykonať so všetkými prvkami poľa, alebo aspoň na susedných prvkoch určitej časti poľa, je vhodné použiť

príkaz cyklu **for**. Aplikáciu príkazu **for** možno vidieť na nasledujúcom príklade.

Príklad: Predpokladajme, že zlomok f je reprezentovaný poľom d tak, že

$$f = \sum_{i=1}^{k-1} d_i * 10^{-i}$$

to znamená prostredníctvom jeho desatinného tvaru s $k - 1$ číslicami. Je potrebné f deliť dvoma, čo sa uskutoční opakovaním delenia pre všetkých $k - 1$ číslic d_i, začnúc s $i = 1$. Urobíme to vydelením číslice dvoma, pričom treba brať do úvahy možný prenos z predchádzajúcej pozície a z uchovania možného zvyšku r pre nasledujúci krok (pozri (1.18)):

$$
\begin{aligned}
&r := 10 * r + d[i]; \\
&d[i] = r \textbf{ div } 2; \\
&r := r - 2 * d[i]
\end{aligned}
\tag{1.18}
$$

Tento proces je aplikovaný v programe 1.1, ktorý vypočítava tabuľku záporných mocnín dvojky. Opakovanie procesu rozpoľovania pri výpočte $2^{-1}, 2^{-2}, \ldots, 2^{-n}$ je opätovne vhodne vyjadrené prostredníctvom príkazu **for**, čo vedie k vloženiu jedného príkazu cyklu **for** do druhého.

PROGRAM 1.1. *Výpočet mocnín dvojky*

```
program mocnina (output);
{desatinná reprezentácia záporných mocnín dvojky}
const n = 10;
type číslica = 0..9;
var i, k, r: integer
    d: array [1..n] of číslica;
begin for k := 1 to n do
      begin write ('.'); r := 0;
         for i := 1 to k - 1 do
         begin r := 10 * r + d[i]; d[i] := r div 2;
               r := r - 2 * d[i];
               write (chr (d[i] + ord ('0')))
         end;
```

```
        d[k]: = 5; writeln ('5')
    end
end.
```

Výsledný výstup uvedeného programu pre $n = 10$ je

```
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

1.7 ŠTRUKTÚRA ZÁZNAM

Najbežnejšou metódou vytvárania štruktúrovaných typov je spájanie ľubovoľných prvkov (zložiek), môžu byť aj štruktúrované, do zložených typov. Príkladmi z matematiky sú komplexné čísla zložené z dvoch reálnych čísel alebo súradnice bodov pozostávajúce z dvoch alebo viacerých reálnych čísel podľa rozmernosti priestoru vymedzeného súradnicovou sústavou. Príkladom z oblasti spracovania údajov môže byť opis osoby pomocou niekoľkých podstatných charakteristických údajov, akými sú meno a priezvisko, dátum narodenia, pohlavie a manželský stav. V matematike sa takýto zložený typ nazýva *karteziánsky súčin* jeho zložkových typov. Vychádza sa zo skutočnosti, že množina hodnôt definovaných týmto zloženým typom je tvorená všetkými možnými kombináciami hodnôt braných z jednotlivých množín hodnôt každého typu zložiek. Potom počet takýchto kombinácií, často nazývaných *n*-ticami, sa rovná súčinu počtu prvkov každej množiny zložiek, čo znamená, že kardinalita zloženého typu je daná súčinom kardinalít typov zložiek.

V oblasti spracovania údajov sa zložené typy, akými sú opisy osôb

alebo predmetov, obyčajne vyskytujú v súboroch alebo bankách údajov a zaznamenávajú podstatné charakteristiky osôb alebo predmetov. Termín *záznam* sa takto stal široko zaužívaným pri opisoch zložitých štruktúr údajov takejto povahy a uprednostňuje sa pred pojmom karteziánsky súčin.

Vo všeobecnosti sa typ záznam T definuje takto:

```
type T = record s1: T1;
                s2: T2;
                 :
                sn: Tn
         end                                   (1.19)
```

Kardinalita (T) = kardinality (T_1) * kardinalita (T_2) * ... * kardinalita (T_n)

Príklady:

```
type komplexné číslo = record re: real;
                              im: real
                       end
type dátum = record deň: 1..31;
                    mesiac: 1..12;
                    rok: 1..2000
             end
type osoba = record meno: alfa;
                    priezvisko: alfa;
                    dátumnarodenia: dátum;
                    pohlavie: (muž, žena);
                    stav: (slobodný, ženatý, vdovec,
                          rozvedený)
             end
```

Hodnotu typu T možno vytvoriť pomocou konštruktora záznamu a priradiť premennej tohto typu:

$$x := T(x_1, x_2, \ldots, x_n) \quad (1.20)$$

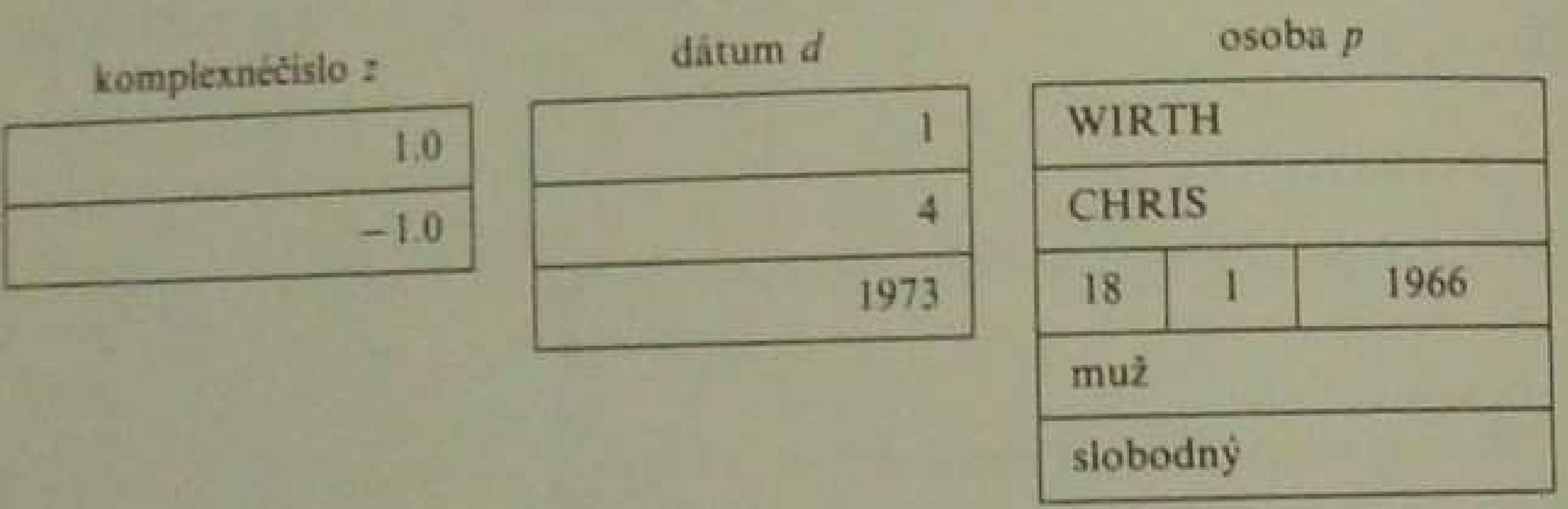
kde x_i sú hodnoty typov zložiek T_i.

Nech sú dané nasledujúce premenné:

z: komplexnéčíslo
d: dátum
p: osoba

Potom sú prípustné napr. nasledujúce priradenia hodnôt (pozri aj *obr. 1.3*):

$z := $ komplexnéčíslo (1.0, −1.0)
$d := $ dátum (1, 4, 1973)
$p := $ osoba ('WIRTH', 'CHRIS', dátum (18, 1, 1966), muž, slobodný)



Obr. 1.3. Záznamy typov komplexnéčíslo, dátum a osoba

Identifikátory $s_1, s_2, \ldots, s_n$, uvedené v rámci definície typu záznam, sú názvy priradené jednotlivým zložkám premenných tohto typu a používajú sa v rámci selektorov záznamu, aplikovaných na štruktúrované premenné typu záznam. Ak máme premennú $x: T$, jej i-tú zložku označujeme

$$x \,.\, s_i \tag{1.21}$$

Výberové aktualizovanie premennej x sa dosiahne použitím toho istého selektorového označenia na ľavej strane priraďovacieho príkazu:

$$x \,.\, s_i := x_i$$

kde x_i je hodnota (výraz) typu T_i.

Nech sú dané tieto premenné typu záznam:

z: komplexnéčíslo
d: dátum
p: osoba

Potom nasledujúce zápisy sú selektory zložiek premenných z, d, p:

z . im	(je typu real)
d . mesiac	(je typu interval 1 . . 12)
p . meno	(je typu alfa)
p . dátumnarodenia	(je typu dátum)
p . dátumnarodenia . deň	(je typu interval 1 . . 31)

Príklad typu osoba ukazuje, že zložka typu záznam môže byť tiež štruktúrovaná. Teda selektory sa môžu reťaziť. Prirodzene, v rámci definícií môžu byť vložené do seba rôzne štruktúrované typy. Napríklad i-tý prvok poľa a, ktorý je zložkou premennej r typu záznam, označujeme

$$r \,.\, a[i]$$

a zložku štruktúry záznam so selektorom s, ktorá je i-tým prvkom poľa a, označujeme

$$a[i] \,.\, s$$

Z charakteristiky karteziánskeho súčinu vyplýva, že obsahuje všetky kombinácie prvkov typov zložiek. Treba však poznamenať, že z hľadiska praktických aplikácií nie všetky musia byť prípustné, t. j. „zmysluplné". Napríklad uvedený typ dátum môže obsahovať prípustné hodnoty (31, 4, 1973) a (29, 2, 1815), ktoré sú však dátumami dvoch dní, ktoré nikdy neexistovali. Teda definícia takéhoto typu neodzrkadľuje reálnu situáciu. Určitý praktický význam to však predsa len má a je povinnosťou programátora, aby zamedzil výskytu bezvýznamných hodnôt v priebehu realizácie programu.

Zmysel použitia premenných typu záznam možno vidieť z nasledujúcej krátkej časti programu. Zámerom programu je zistenie celkového počtu „osôb", reprezentovaných premennou „a" typu „pole", ktoré sú ženského pohlavia a nie sú vydaté.

```
var a: array [1..N] of osoba;
    i, počet: integer;
begin
        počet i: = 0;                                        (1.22)
        for i: = 1 to N do
            if (a[i].pohlavie = žena) ∧ (a[i].stav = slobodný)
            then počet: = počet + 1
end
```

Významným invariantom cyklu je, že počet $= C(i)$, pričom $C(i)$ je počet slobodných ženských príslušníčok podmnožiny $a_1, a_2, \ldots, a_i$.

V nasledujúcom variante uvedeného príkazu cyklu je použitá jazyková konštrukcia nazývaná **with**-príkaz:

```
for i: = 1 to N do
    with a[i] do                                         (1.23)
        if (pohlavie = žena) ∧ (stav = slobodný)
        then počet: = počet + 1
```

Význam príkazu **with** *r* **do** *s* spočíva v tom, že identifikátory selektorov typu premennej *r* môžu byť v rámci príkazu *s* použité bez prefixu a je zrejmé, že sa vzťahujú na premennú *r*. Príkaz **with** takto slúži na skrátenie zápisu textu programu. Okrem toho použitím príkazu **with** odpadá potreba opätovného vypočítavania adresy indexovaného prvku *a*[*i*] v pamäti počítača.

V nasledujúcom príklade predpokladajme (napr. z hľadiska rýchlejšieho vyhľadávania), že určité skupiny osôb sú spolu pospájané v poli *a*. Spojovacia informácia je reprezentovaná dodatočnou zložkou záznamovej štruktúry osoba. Nech sa táto zložka nazýva *spojenie*. Prostredníctvom takýchto spojení sú záznamy pospájané do lineárnej reťaze, takže nájdenie nasledovníka a prechodcu každej osoby sa dá jednoducho uskutočniť. Zaujímavou vlastnosťou takejto techniky spájania je, že reťaz možno prejsť v obidvoch smeroch na základe jediného čísla nachádzajúceho sa v každom zázname. Princíp uvedenej techniky je takýto:

Predpokladajme, že indexy troch po sebe nasledujúcich členov reťaze sú i_{k-1}, i_k, i_{k+1}. Hodnotu zložky spojenia zvolíme tak, aby pre *k*-tý člen bola $i_{k+1} - i_{k-1}$. Pri prechode reťazou smerom dopredu sa hodnota indexu i_{k+1} vypočíta pomocou dvoch okamžitých hodnôt indexových premenných $x = i_{k-1}$ a $y = i_k$ ako $i_{k+1} = x + a[y].spojenie$. Pri prechode reťazou opačným smerom sa i_{k-1} vypočíta pomocou premenných $x = i_{k+1}$ a $y = i_k$ ako $i_{k-1} = x - a[y].spojenie$. V *tab. 1.2* sú pospájané všetky osoby rovnakého pohlavia.

Tabuľka 1.2

Pole s prvkami typu Osoba

	Meno	Pohlavie	Spojenie
1	Katarína	Ž	2
2	Karol	M	2
3	Kristína	Ž	5
4	Jozef	M	3
5	Adam	M	3
6	Eva	Ž	5
7	Peter	M	5
8	Oľga	Ž	3
9	Mária	Ž	1
10	Pavol	M	3

Štruktúry záznam a pole majú jednu spoločnú vlastnosť, ktorou je ľubovoľný prístup k ich jednotlivým prvkom. Štruktúru záznam možno považovať za všeobecnejšiu v tom zmysle, že sa nevyžaduje, aby všetky jej zložkové typy boli identické. Na druhej strane štruktúra pole zasa poskytuje väčšiu flexibilitu tým, že umožňuje, aby jej selektory prvkov boli vypočítanými hodnotami (reprezentovanými prostredníctvom výrazov). Selektory zložiek záznamu sú pevne stanovené identifikátory deklarované v rámci definície typu záznam.

1.8 VARIANTY ŠTRUKTÚR ZÁZNAM

Z praktického hľadiska je často výhodné a prirodzené považovať dva typy jednoducho za varianty toho istého typu. Napríklad typ súradnica, uvedený v predchádzajúcom článku, možno považovať za

zjednotenie jeho dvoch variantov — karteziánskej a polárnej súradnice, ktorých zložkami sú zodpovedajúco a) dve dĺžky, b) dĺžka a uhol. Vzhľadom na možnosť identifikácie konkrétneho variantu danej premennej, sa zavádza tretia zložka, ktorá sa nazýva *diskriminátor typu* alebo *rozlišovacia položka*.

type súradnica =
  record case druh: (karteziánska, polárna) **of**
    karteziánska: (x, y: real);
    polárna: (r: real; φ: uhol)
  end

V uvedenom príklade je druh identifikátorom rozlišovacej položky. Ak má druh hodnotu karteziánska, tak identifikátormi súradníc sú symboly x a y; v prípade, že hodnotou je polárna, súradnice sú identifikovateľné pomocou identifikátorov r, φ.

Množina hodnôt určených typom súradnica je daná zjednotením dvoch typov

$$T_1 = (x, y: \text{real})$$

a

$$T_2 = (r: \text{real}; \varphi: \text{uhol})$$

a jeho kardinalita je daná súčtom kardinalít typov T_1 a T_2.

Vo väčšine prípadov nedochádza k zjednoteniu úplne odlišných typov, ale skôr sa spájajú typy s aspoň čiastočne rovnakými zložkami. Prevládajúci počet výskytov takto štruktúrovaných typov vlastne spôsobil vznik termínu štruktúra *záznam s variantmi*. Typickým príkladom takejto štruktúry je typ osoba, uvedený v predchádzajúcom článku, v rámci ktorého závažné charakteristiky, zaznamenávané v súbore, závisia od pohlavia konkrétnej osoby. Napríklad pre muža môže byť v niektorých situáciách dôležité, koľko váži a či má alebo nemá bradu. Naopak pre ženu môžu byť dôležité tri charakteristické rozmery, pričom jej váha môže byť považovaná za diskrétnu informáciu, a tým i nepodstatnú. Na základe spomenutých úvah teraz uvedieme modifikovanú definíciu typu osoba:

type osoba =
  record *meno*, *priezvisko*: alfa;
    dátumnarodenia: dátum;
    stav: (slobodný, ženatý, vdovec, rozvedený);
  case pohlavie: (muž, žena) **of**
  muž: (*váha*: real;
    bradatý: boolean);
  žena: (*rozmery*: **array** [1 .. 3] **of** integer)
  end

Všeobecný tvar definície typu záznam s variantmi je
type T =

record $s_1: T_1; s_2: T_2; \ldots; s_{n-1}: T_{n-1}$;
  case $s_n: T_n$ **of**
  $c_1: (s_{1,1}: T_{1,1}; s_{1,2}: T_{1,2}; \ldots; s_{1,n_1}: T_{1,n_1})$;
  $c_2: (s_{1,1}: T_{2,1}; s_{2,2}: T_{2,2}; \ldots; s_{2,n_2}: T_{2,n_2})$;   (1.24)
  ⋮
  $c_m: (s_{m,1}: T_{m,1}; s_{m,2}: T_{m,2}; \ldots; s_{m,n_m}: T_{m,n_m})$
  end

Symbolmi s_i a $s_{i,j}$ sú pomenované selektory zložiek záznamu, ktoré sú typov T_i a $T_{i,j}$. Symbol s_n je identifikátorom rozlišovacej položky, ktorej typ je T_n. Konštanty $c_1, c_2, \ldots, c_m$ označujú hodnoty (skalárneho) typu T_n. Premenná x typu T pozostáva zo zložiek $x.s_1, x.s_2, \ldots, x.s_n$, $x.s_{k,1}, \ldots, x.s_{k,n_k}$ práve vtedy, ak (momentálna) hodnota zložky $x.s_n$ je c_k (t.j. $x.s_n = c_k$). Zložky $x.s_1, x.s_2, \ldots, x.s_n$ predstavujú *spoločnú časť* pre m variantov záznamovej štruktúry T.

V dôsledku toho použitie zložkového selektora $x.s_{k,h}$ $(1 \leq h \leq n_k)$ v prípade, že $x.s_n \neq c_k$, treba pokladať za vážnu programátorskú chybu, pretože (napr. pri výberovej aktualizácii súboru údajov typu osoba) by mohlo dôjsť k usporiadaniu žien podľa toho, či nosia bradu alebo nie.

Vzhľadom na to sa pri používaní štruktúr záznam s variantmi vyžaduje maximálna opatrnosť. Odporúča sa, aby príslušné operácie s jednotlivými variantmi štruktúry záznam boli sústredené v rámci príkazu

výberu, nazývaného príkaz **case**. Jeho štruktúra najlepšie odzrkadľuje štruktúru typu záznam s variantmi, uvedenú v rámci definícií typu:

$$
\begin{aligned}
&\textbf{case } x.s_n \textbf{ of}\\
&\quad c_1:\ S_1;\\
&\quad c_2:\ S_2;\\
&\quad \vdots\\
&\quad c_m:\ S_m\\
&\textbf{end}
\end{aligned}
\tag{1.25}
$$

Symbolom S_k sa vyjadruje príkaz, ktorý sa vykoná v prípade, že sa bežná hodnota x vzťahuje na variant k, t.j. príkaz S_k sa vykoná vtedy a len vtedy, ak rozlišovacia položka $x.s_n$ má hodnotu c_k.

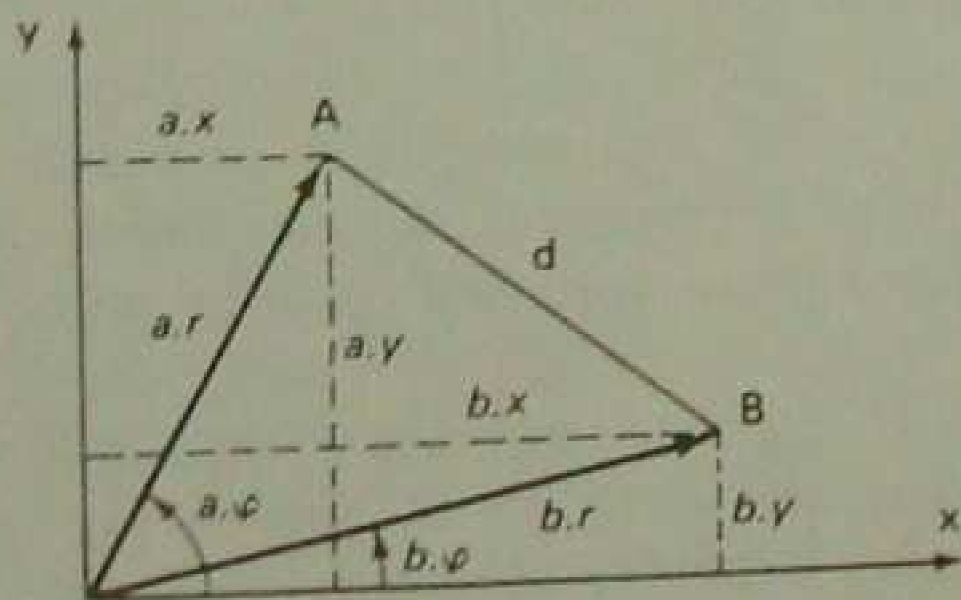
V dôsledku toho sa dá pomerne jednoduchým spôsobom kontrolovať správnosť aplikácie identifikátora selektora. Stačí overiť, či každý príkaz S_k obsahuje iba selektory:

$$x.s_1, \ldots, x.s_{n-1}$$

a

$$x.s_{k,1}, \ldots, x.s_{k,n_k}$$

Cieľom nasledujúcej krátkej časti programu je vypočítať vzdialenosť medzi dvoma bodmi A, B, určenými prostredníctvom dvoch premenných a, b, ktoré sú typu súradnica (t.j. štruktúra záznam s variantmi). Spôsob výpočtu je rozdielny vzhľadom na štyri možné kombinácie karteziánskych a polárnych súradníc (*obr. 1.4*).



Obr. 1.4. Karteziánske a polárne súradnice


```
case a.druh of
  Karteziánska: case b.druh of
                  karteziánska: d: = sqrt(sqr(a.x − b.x) +
                                  + sqr(a.y − b.y));
                  polárna: d: = sqrt(sqr(a.x − b.r * cos(b.φ) +
                                  + sqr(a.y − b.r * sin(b.φ))
                end;
  Polárna: case b.druh of
             karteziánska: d: = sqrt(sqr(a.r * sin(a.φ) − b.x) +
                                 + sqr(a.r * cos(a.φ) − b.y));
             polárna: d: = sqrt(sqr(a.r) + sqr(b.r) −
                              − 2 * a.r * b.r * cos(a.φ − b.φ))
           end
end
```

1.9 ŠTRUKTÚRA MNOŽINA

Treťou základnou štruktúrou údajov, po poli a zázname, je *množina*. Definuje sa

$$\textbf{type } T = \textbf{set of } T_0 \tag{1.26}$$

Možnými hodnotami premennej x, ktorá je typu T, sú množiny prvkov typu T_0. Množinu všetkých podmnožín prvkov množiny T_0 nazývame *potenčná množina* T_0. Typ T teda zahŕňa potenčnú množinu jeho *základného typu* T_0.

Príklady:

```
type celočismnožina = set of 0..30
type množinaznakov = set of char
type stavpásky = set of výnimka
```

V druhom príklade je typ množina znakov definovaný prostredníctvom štandardnej množiny znakov označenej identifikátorom typu char; v treťom príklade je typ stavpásky definovaný na základe množiny výnimočných stavov, ktorá môže byť vyjadrená prostredníctvom skalárneho typu

type výnimka = (nezavedený, ručný, paritný, chybný)

opisujúceho výnimočné stavy, v ktorých sa môže magnetická pásková jednotka počítača nachádzať.

Predpokladajme, že sú definované takéto premenné:

is: celočísmnožina
cs: množinaznakov
t: **array** [1 . . 6] **of** stavpásky

Potom je možné vytvoriť a priradiť napr. nasledujúce hodnoty uvedených typov množina:

is : = [1, 4, 9, 16, 25]
cs : = ['+', '−', '*', '/']
t[3] : = [ručný]
t[5] : = [ ]
t[6] : = [nezavedený . . chybný]

Poznámka: v týchto príkladoch (ako aj v ďalšom texte) sa na označenie množinových konštrukcií používajú hranaté zátvorky, a nie zložené, ako býva zvykom v matematike. Do zložených zátvoriek sa ukladajú poznámky v rámci programového textu.

Priradenia v uvedených príkladoch možno charakterizovať takto: Hodnota priradená prvku *t*[3] predstavuje jednoprvkovú množinu pozostávajúcu z prvku ručný; prvku *t*[5] je priradená prázdna množina, čo znamená, že piata pásková jednotka je vrátená do operačného (nie výnimočného) stavu, zatiaľ čo šiestej páskovej jednotke je priradená množina všetkých štyroch výnimočných stavov.

Kardinalita množinového typu *T* je

$$\text{kardinalita}(T) = 2^{\text{kardinalita}(T_0)} \tag{1.27}$$

čo sa dá jednoducho odvodiť zo skutočnosti, že každý z prvkov T_0, ktorých počet sa rovná kardinalite (T_0), musí byť reprezentovaný jednou z dvoch možných hodnôt „prítomný" alebo „neprítomný" a že všetky prvky sú navzájom nezávislé. Z hľadiska efektívnej a ekonomickej implementácie množinových typov je zrejmé, že základný typ množiny musí byť konečný a súčasne jeho kardinalita čo najmenšia.

Pre každý typ množina sú definované štyri základné operátory:

* množinový prienik
\+ množinové zjednotenie
− množinový rozdiel
in množinová príslušnosť

Operácie prienik, resp. zjednotenie dvoch množín sa niekedy nazývajú násobenie, resp. sčítanie dvoch množín. Priorita množinových operátorov je takáto: najvyššiu má prienik, potom zjednotenie a rozdiel. Operátor množinovej príslušnosti má najnižšiu prioritu a zaraďujeme ho medzi relačné operátory. V nasledujúcich príkladoch sú uvedené množinové výrazy a ich ekvivalenty vyjadrené pomocou zátvoriek:

$$r * s + t = (r * s) + t$$
$$r - s * t = r - (s * t)$$
$$r - s + t = (r - s) + t$$
$$x \textbf{ in } s + t = x \textbf{ in } (s + t)$$

Prvým príkladom aplikácie štruktúry množina je jednoduchý lexikálny analyzátor kompilátora. Predpokladáme, že cieľom lexikálneho analyzátora je preloženie postupnosti znakov do postupnosti textových jednotiek prekladaného jazyka, nazývaných atómy alebo symboly. Lexikálny analyzátor sa realizuje procedúrou, ktorá pri každej výzve prečíta dostatočný počet vstupných znakov na to, aby z nich mohla vygenerovať ďalší výstupný symbol. Jednotlivé pravidlá prekladu sú tieto:

1. Množinu výstupných symbolov tvoria tieto prvky: identifikátor, číslo, menšialeborovnása, väčšialeborovnása, priradenie a iné, ktoré odpovedajú rozličným jednoduchým znakom, akými sú +, −, *, / atď.

2. Symbol *identifikátor* sa generuje po prečítaní postupnosti písmen a číslic začínajúcich písmenom.

3. Symbol *číslo* sa generuje po prečítaní postupnosti číslic.

4. Symboly *menšialeborovnása*, *väčšialeborovnása* a *priradenie* sa vygenerujú po prečítaní príslušných dvojíc znakov < =, > =, : =.

5. Medzery (prázdne znaky) a konce riadkov vstupného textu sa preskočia.

K dispozícii máme jednoduchú procedúru read (*x*), ktorá prečíta ďalší znak zo vstupnej postupnosti a priradí ho premennej *x*. Výsledný

výstupný symbol sa priradí globálnej premennej *sym*. Ďalej máme aj globálne premenné *ident* a *num*, ktorých použitie bude zrejmé z programu 1.2, a premennú *ch*, ktorá obsahuje práve prečítaný znak zo vstupnej postupnosti. Identifikátor *S* označuje zobrazenie znakov do symbolov, t.j. pole symbolov, pri ktorých definičný obor indexov tvoria tie znaky, ktoré nie sú ani číslice, ani písmená. Použitie množín znakov poukazuje na to, ako môže byť lexikálny analyzátor naprogramovaný nezávisle od usporiadania znakov v základnej množine znakov.

V druhom príklade treba riešiť problém vytvorenia školského rozvrhu hodín. Predpokladajme, že *M* študentov sa rozhodlo, že budú navštevovať prednášky z *N* predmetov. Úlohou tvorcu rozvrhu je, aby správnym rozvrhom zamedzil vzniku konfliktných situácií, ktoré by mohli vzniknúť napr. tým, že by niektorí študenti mali naplánované dve prednášky na tú istú hodinu a na ten istý deň [1.1].

Vo všeobecnosti je vytvorenie rozvrhu hodín jedným z najťažších kombinatorických problémov. Treba pritom uvážiť mnohé obmedzujúce faktory a rozhodnúť sa medzi veľkým množstvom rôznych variantov. V našom príklade problém veľmi zjednodušíme a nebudeme sa zaoberať skutočnou situáciou, ktorá pri tvorbe rozvrhu hodín vzniká.

Najprv si uvedomme, že na to, aby sme našli vhodné riešenie problému „paralelných" vyučovaní, musíme naše rozhodnutia budovať na základe množiny údajov získaných počas individuálnych študentských zápisov na začiatku školského roka. Pritom očíslujeme prednášky, ktoré nemôžu prebiehať v tom istom čase. Preto pri implementácii programu na tvorbu rozvrhu najprv naprogramujeme proces redukcie údajov, založený na nasledujúcich deklaráciách a dohode, že študenti sú očíslovaní od 1 po *M*, zatiaľ čo prednášky od 1 po *N*.

PROGRAM 1.2. *Lexikálny analyzátor*

```
var ch: char;
    sym: symbol;
    num: integer;
    ident: record
            k: 0..maxk;
            a: array [1..maxk] of char
          end;

procedure lexanalyzátor;
var ch1 : char;

begin {preskočenie medzier}
  while ch = '␣' do read (ch);
  if ch in ['A'..'Z'] then
    with ident do
    begin sym : = identifikátor; k : = 0;
      repeat if k < maxk then
          begin k : = k + 1; a[k] : = ch
          end;
        read (ch)
      until ¬ (ch in ['A'..'Z', '0'..'9'])
    end else
  if ch in ['0'..'9'] then
    begin sym : = číslo; num : = 0;
      repeat num : = 10 * num + ord (ch) − ord ('0');
        read (ch)
      until ¬ (ch in ['0'..'9'])
    end else
  if ch in ['<', ':', '>'] then
    begin ch1 : = ch; read (ch);
      if ch = '=' then
      begin
        if ch1 = '<' then sym : = menšialeborovnása else
        if ch1 = '>' then sym : = väčšialeborovnása
        else sym : = priradenie;
        read (ch)
      end
    else sym : = S[ch1]
    end else

  begin {ostatné symboly}
      sym : = S[ch]; read (ch)
  end
end {lexanalyzátor}
```

```
type prednáška = 1 .. N;
     študent = 1 .. M;
     výber = set of prednáška;
var s: prednáška;
    i: študent;
    zápis: array [študent] of výber;
    konflikt: array [prednáška] of výber;                    (1.28)
    {Určenie množín konfliktných prenášok
    z individuálnych študentských zápisov}
for s: = 1 to N do konflikt [s]: = [];
for i: = 1 to M do
  for s: = 1 to N do
    if s in zápis [i] then
          konflikt [s]: = konflikt [s] + zápis [i]
```

(Všimnime si, že *s* **in** *konflikt* [*s*] je dôsledkom tohto algoritmu.)

Hlavnou úlohou teraz je zostrojiť rozvrh hodín, t. j. zoznam prednášok, ktoré už nebudú spôsobovať konfliktné situácie. Z množiny všetkých prednášok vyberieme vhodné nekonfliktné podmnožiny prednášok, ktoré odčítavame od premennej *zvyšok* dovtedy, kým nie je množina ostávajúcich prednášok prázdna.

```
var k: integer;
    zvyšok, vyučovanie: výber;
    rozvrhhodín: array [1 .. N] of výber;
begin
    k: = 0; zvyšok: = [1 .. N];
    while zvyšok ≠ [ ] do
      begin
          vyučovanie: = „ďalší vhodný výber";                  (1.29)
          zvyšok: = zvyšok − vyučovanie;
          k: = k + 1;
          rozvrhhodín [k]: = vyučovanie
      end
end
```

Ako realizovať ďalší vhodný výber? Najskôr vyberieme ľubovoľnú prednášku z množiny zvyšujúcich prednášok. Ďalší výber už je obmedzený na množinu tých zvyšujúcich prednášok, ktoré nespôsobujú konflikty s prednáškami vybratými na začiatku. Túto množinu nazývame *skúšobná množina*. Keď hľadáme kandidáta z tejto množiny, vidíme, že jeho výber závisí od toho, či prienik množiny vybratých prednášok s množinou konfliktov kandidáta je prázdna množina. Toto vedie k nasledujúcemu zjemneniu príkazu vyučovanie: = ďalší vhodný výber:

```
var s, t: prednáška:
    skúšmnožina: výber;
begin s: = 1;
  while ¬ (s in zvyšok) do s: = s + 1;
  vyučovanie: = [s];
  skúšmnožina: = zvyšok − konflikt [s];                        (1.30)
  for t: = 1 to N do
      if t in skúšmnožina then
      begin if konflikt [t] * vyučovanie = []
            then vyučovanie: = vyučovanie + [t]
      end
end
```

Zrejmé je, že uvedené riešenie problému výberu vhodných vyučovacích hodín neprispeje k vytvoreniu optimálneho rozvrhu hodín. V nepriaznivých prípadoch môže byť totiž počet vyučovacích hodín taký veľký ako počet prednášok, dokonca aj v prípade prípustnosti simultánneho plánovania.

1.10 REPREZENTÁCIA ŠTRUKTÚR POLE, ZÁZNAM A MNOŽINA

Podstata používania abstrakcií v programovaní spočíva v tom, že program je možné vytvoriť, porozumieť mu a overiť jeho správnosť na základe pravidiel implicitne spojených s abstrakciami. Pritom nie sú potrebné nijaké hlbšie znalosti o implementácii a reprezentácii abstrakcií v konkrétnom počítačovom prostredí. Napriek tomu (najmä pre dobrého programátora) je výhodné, ak máme prehľad o najbežnejších

spôsoboch reprezentácie základných pojmov programovacích abstrakcií, akými sú napr. základné štruktúry údajov. Výhoda je najmä v tom, že programátor ovládajúci spomínané techniky dokáže elegantnejšie navrhnúť štruktúru celého programu, ako aj vybrať vhodné štruktúry údajov, a to nielen z hľadiska abstraktných vlastností štruktúr, ale i vzhľadom na ich realizáciu na skutočnom počítači. Pritom sa berú do úvahy jeho možnosti a kapacitné ohraničenia.

Problém reprezentácie údajov sa transformuje na problém zobrazenia abstraktných štruktúr do pamäti počítača. Pamäť počítača si môžeme — v prvom priblížení — predstaviť ako pole individuálnych pamäťových buniek nazývaných *slová*. Indexy týchto slov sa nazývajú *adresy*.

$$\textbf{var}\ \textit{pamäť}: \textbf{array}\ [\text{adresa}]\ \textbf{of}\ \text{slovo} \tag{1.31}$$

Kardinality typov adresa a slovo sa menia v závislosti od počítačov. Zvláštnym problémom je však veľká variabilita kardinality slova. Jeho logaritmus sa nazýva *dĺžka slova*, pretože vyjadruje počet bitov v jednej pamäťovej bunke.

1.10.1 REPREZENTÁCIA POLÍ

Pod reprezentáciou štruktúry pole rozumieme zobrazenie (abstraktného) poľa, ktorého prvky sú typu T, do pamäti počítača, ktorú predstavuje pole s prvkami typu *slovo*. Pole má byť zobraziteľné takým spôsobom, aby sa adresy jednotlivých prvkov poľa dali vypočítať najjednoduchším (a tým aj najefektívnejším) možným spôsobom. Adresa alebo pamäťový index j-tého prvku poľa sa vypočíta na základe lineárnej funkcie zobrazenia

$$i = i_0 + j * s \tag{1.32}$$

kde i_0 je adresa prvého prvku poľa a s je počet slov, ktoré zaberá jeden prvok poľa.

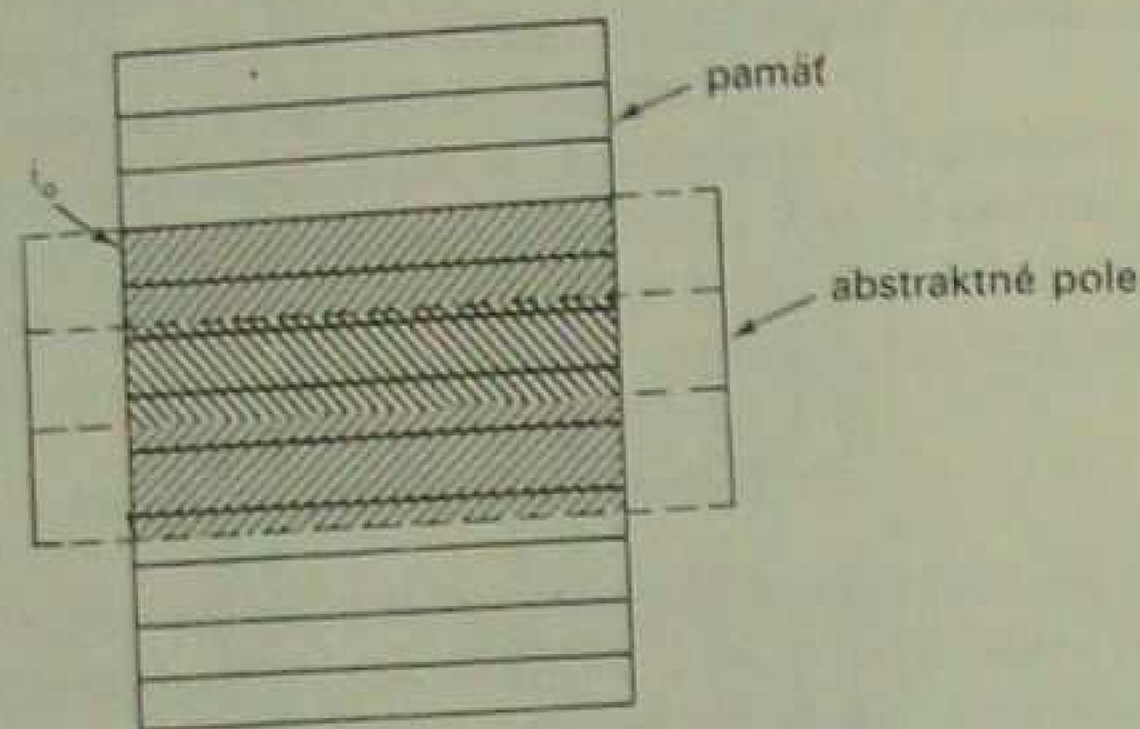
Vzhľadom na to, že podľa definície je slovo najmenšou adresovateľnou jednotkou pamäti počítača, je, samozrejme, veľmi žiadúce, aby s bolo celé číslo, (v najjednoduchšom prípade $s = 1$). Ak s nie je celé číslo (čo je normálne), tak sa obyčajne zaokrúhľuje na najbližšie väčšie celé

Tabuľka 1.3

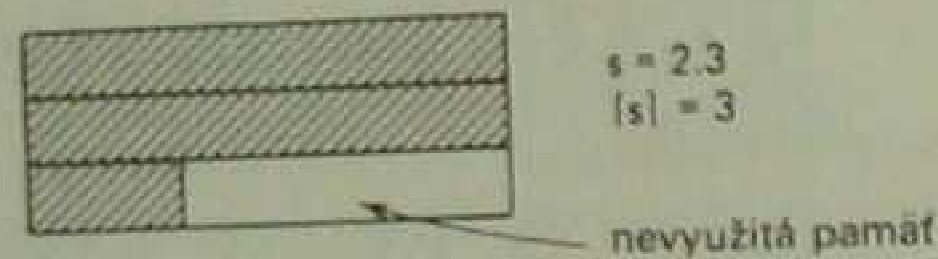
Základné štruktúry údajov

Štruktúra	Deklarácia	Selektor	Prístup k prvkom alebo položkám pomocou	Typy prvkov alebo položiek	Kardinalita
Pole	a: **array** [I] **of** T_0	$a[i]$ $(i \in I)$	selektora s vypočítateľným indexom i	všetky rovnaké (T_0)	$\text{card}(T_0)^{\text{card}(I)}$
Záznam	r: **record** $s_1 : T_1$; $s_2 : T_2$; … $s_n : T_n$ **end**	$r.s$ $(s \in \{s_1, \ldots, s_n\})$	selektora s deklarovaným identifikátorom položky s	môžu byť individuálne odlišné	$\prod_{i=1}^{n} \text{card}(T_i)$
Množina	s: **set of** T_0		testu množinovej príslušnosti pomocou relačného operátora **in**	všetky rovnaké (skalárneho typu T_0)	$2^{\text{card}(T_0)}$

číslo $\lceil s \rceil$. Každý prvok poľa potom zaberá $\lceil s \rceil$ slov, pričom $\lceil s \rceil - s$ slov pamäti počítača ostáva nevyužitých (*obr. 1.5* a *1.6*). Zaokrúhľovanie potrebného počtu slov pamäti smerom k najbližšiemu väčšiemu celému číslu sa nazýva *vypĺňovanie*.



Obr. 1.5. Zobrazenie poľa do pamäti



Obr. 1.6. Reprezentácia štruktúry záznam s výplňou

Koeficient využitia pamäti u je daný podielom minimálnej veľkosti pamäti potrebnej na reprezentáciu danej štruktúry a skutočne použitej veľkosti pamäti:

$$u = \frac{s}{s'} = \frac{s}{\lceil s \rceil} \qquad (1.33)$$

Pretože implementátor sa bude snažiť, aby koeficient využitia pamäti bol, pokiaľ možno, čo najbližšie k jednotke, a vzhľadom na to, že pristupovanie k častiam strojových slov je náročný a relatívne neefektívny proces, bude potrebné, aby sa rozhodol pre kompromisné riešenie. Treba pri tom brať do úvahy tieto skutočnosti:

1. Vypĺňovanie pamäti zníži jej využitie.

2. Vyhnutie sa vypĺňovaniu môže zapríčiniť neefektívny prístup k častiam slov.

3. Prístup k častiam slov môže zase spôsobiť, že vygenerovaný cieľový kód (skompilovaného programu) sa natoľko zväčší, že sa vlastne zneutralizuje zisk z vynechania vypĺňovania.

V skutočnosti sú úvahy 2 a 3 natoľko dominantné, že kompilátory sú konštruované tak, že automaticky vykonávajú vypĺňovanie pamäťového priestoru. Poznamenajme, že hodnota koeficientu využitia pamäti bude vždy

$$u > 0.5, \quad \text{ak} \quad s > 0.5$$

Ale v prípade, že $s \leq 0.5$, môže sa hodnota koeficientu využitia pamäti podstatne zvýšiť tým, že umiestnime viac prvkov poľa do jedného strojového slova. Táto technika sa nazýva *zhusťovanie*. Ak je do jedného slova zhustených n prvkov poľa, tak koeficient využitia pamäti je (*obr. 1.7*)

$$u = \frac{n \,.\, s}{\lceil n \,.\, s \rceil} \qquad (1.34)$$

Obr. 1.7. Zhustenie šiestich prvkov do jedného slova

Prístup k i-tému prvku zhusteného poľa vyžaduje výpočet adresy j slova, v ktorom sa daný prvok nachádza, a výpočet pozície k prvku v rámci tohto slova. Čiže

$$j = i \ \mathbf{div} \ n \qquad (1.35)$$
$$k = i \ \mathbf{mod} \ n = i - j \bullet n$$

Vo väčšine programovacích jazykov programátor nemá možnosť ovplyvniť mechanizmus určovania reprezentácie abstraktných štruktúr údajov. Predsa by však v niektorých prípadoch bolo dobré mať možnosť určenia požadovaného zhustenia, a to najmä vtedy, keď sa viac ako jeden prvok zmestí do jedného strojového slova, t. j. v prípadoch, keď koeficient využitia pamäti bude mať hodnotu 2 a viac. Za tým

účelom zavedieme takúto dohodu: Požadované zhustenie prvkov poľa (alebo zložiek záznamu) vyznačíme tým, že v deklaráciách pred symbol **array** (alebo **record**) uvedieme prefixový symbol **packed**.

Príklad:

type alfa = **packed array** [1 . . n] **of** char

Táto vlastnosť je zvlášť významná pri počítačoch, ktoré majú veľkú dĺžku strojového slova a pomerne vyhovujúci prístup k častiam slova. Základnou vlastnosťou uvedeného prefixu je, že v žiadnom prípade nemení zmysel (alebo správnosť) programu. Z toho vyplýva, že voľbu alternatívnej reprezentácie je možné označiť použitím tohto prefixu so zárukou, že zmysel programu ostane neporušený.

Ak sa zhustenie poľa vykoná naraz, klesnú náklady spojené s adresovaním každého prvku zhusteného poľa. V takomto prípade odpadá vyhodnocovanie zložitej zobrazovacej funkcie, ktoré je potrebné vykonať pri adresovaní každého prvku poľa. Okrem toho sa dá realizovať účinné sekvenčné prehľadávanie celej štruktúry pole. Za týmto účelom sa zavádzajú dve štandardné procedúry *pack* (prevedenie do zhusteného tvaru) a *unpack* (prevedenie do nezhusteného tvaru) definované nasledujúcim spôsobom:

Predpokladajme, že existujú premenné u, p definované takto:

u: **array** [a . . d] **of** T
p: **packed array** [b . . c] **of** T

pričom a, b, c, d sú rovnakého skalárneho typu a platí $a \leqq b \leqq c \leqq d$.

Potom

$$\text{pack}\,(u, i, p), \qquad (a \leqq i \leqq b - c + d) \tag{1.36}$$

je ekvivalentné s príkazom

$$p[j] := u[j + i - b], \qquad j = b \ldots c$$

a

$$\text{unpack}\,(p, u, i), \qquad (a \leqq i \leqq b - c + d) \tag{1.37}$$

je ekvivalentné s

$$u[j + i - b] := p[j], \qquad j = b \ldots c$$

1.10.2 REPREZENTÁCIA ŠTRUKTÚR ZÁZNAM

Záznamy sú zobrazované do pamäti počítača (je im pridelená pamäť) postupným zobrazovaním (ukladaním) ich jednotlivých zložiek. Adresu záznamovej zložky r_i vzhľadom na začiatočnú adresu záznamu r nazývame *posunutie* (alebo *ofset*) k_i zložky r_i. Vypočíta sa podľa vzorca

$$k_i = s_1 + s_2 + \ldots + s_{i-1} \tag{1.38}$$

pričom s_j je veľkosť (v slovách) j-tej zložky záznamu.

Zo skutočnosti, že všetky prvky poľa sú rovnakého typu, vyplýva

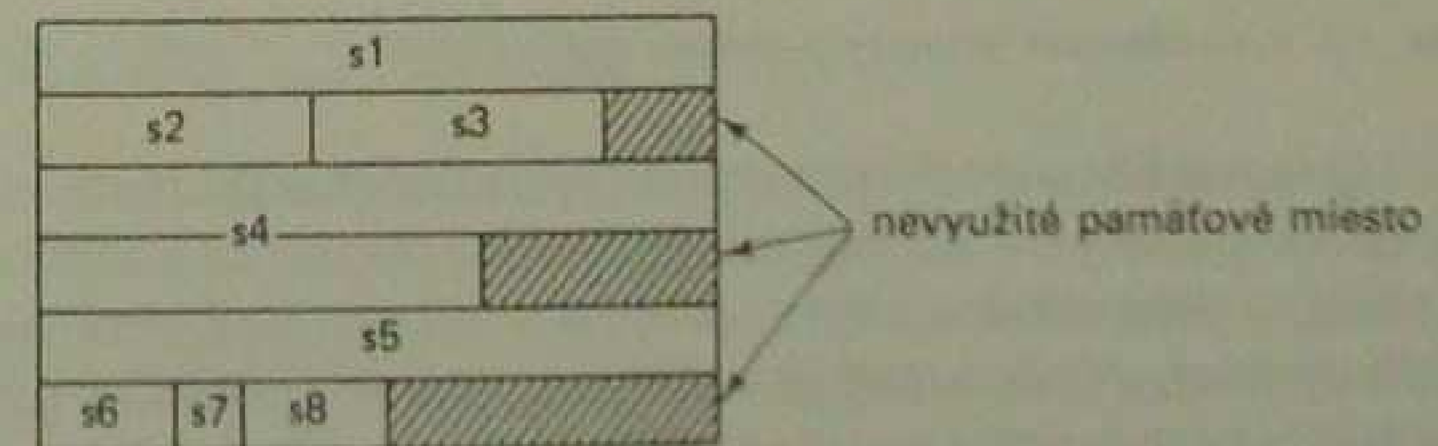
$$s_1 = s_2 = \ldots = s_n$$

a preto

$$k_i = s_1 + s_2 + \ldots + s_{i-1} = (i - 1) \,.\, s$$

Ale vzhľadom na všeobecnú povahu štruktúry záznam takéto zjednodušenie funkcie pre výpočet posunutia zložky záznamu nie je možné. Preto sú zložky záznamu prístupné jedine prostredníctvom presne stanovených identifikátorov. Toto obmedzenie prináša zase jednu výhodu, ktorou je skutočnosť, že posunutia zložiek záznamu sú známe už v čase kompilácie. Preto nie je potrebné generovať pre výpočet posunutia kód (ako napr. v prípade indexov poľa) a tento potom vykonávať v čase realizácie programu. Dosiahne sa dobre známa efektívnosť prístupu k položkám štruktúr typu záznam.

Problém zhusťovania začne vystupovať v tých prípadoch, keď sa niekoľko zložiek záznamu zmestí do jedného strojového slova (*obr. 1.8*). Opäť budeme predpokladať, že potrebu zhusťovania je možné deklarovať prefixovým symbolom **packed** v rámci definície typu **record**.



Obr. 1.8. Reprezentácia zhustenej štruktúry záznam

Vzhľadom na to, že posunutie môže vypočítať kompilátor, dajú sa rovnakým spôsobom vypočítať i posunutia zložiek v rámci slov. To znamená, že pre väčšinu počítačov spôsobí zhusťovanie záznamov podstatne menší pokles efektívnosti sprístupňovania jednotlivých položiek ako v prípade zhusťovania polí.

1.10.3 REPREZENTÁCIA MNOŽÍN

Množinu s možno vhodne reprezentovať v pamäti počítača pomocou jej *charakteristickej funkcie* $C(s)$. Je to pole logických hodnôt, ktorého i-tý prvok vyjadruje prítomnosť alebo neprítomnosť prvku i v danej množine. Veľkosť poľa je určená kardinalitou typu množina.

$$C(s_i) \equiv (i \textbf{ in } s) \qquad (1.39)$$

Napríklad množinu malých celých čísel

$$s = [1, 4, 8, 9]$$

možno reprezentovať postupnosťou logických hodnôt F (false) a T (true)

$$C(s) = (\text{FTFFTFFFTT})$$

ak základným typom množiny s je celočíselný interval 0 . . 9. Postupnosť logických hodnôt je v pamäti počítača reprezentovaná reťazcami bitov (*obr. 1.9*).

Obr. 1.9. Reprezentácia množiny pomocou reťazca bitov

Reprezentácia množín prostredníctvom ich charakteristických funkcií má výhodu v tom, že operácie zjednotenia, prieniku a rozdielu dvoch množín sa dajú vhodne implementovať pomocou elementárnych logických operácií. Nasledujúce ekvivalencie, platné pre všetky prvky i základného typu množín x a y, ukazujú vzťahy medzi logickými a množinovými operáciami:

$$\begin{aligned} i \textbf{ in } (x + y) &\equiv (i \textbf{ in } x) \vee (i \textbf{ in } y) \\ i \textbf{ in } (x * y) &\equiv (i \textbf{ in } x) \wedge (i \textbf{ in } y) \\ i \textbf{ in } (x - y) &\equiv (i \textbf{ in } x) \wedge \neg (i \textbf{ in } y) \end{aligned} \qquad (1.40)$$

Spomenuté logické operácie sú dostupné na všetkých číslicových počítačoch a navyše pracujú súbežne na všetkých prvkoch (bitoch) strojového slova. Ak teda chceme efektívne implementovať základné množinové operácie, je žiadúce, aby množiny boli reprezentované malým počtom slov, na ktorých sa dajú realizovať nielen uvedené logické operácie, ale aj operácie posunu. Operácia zistenia množinovej príslušnosti je potom implementovaná pomocou dvoch inštrukcií: jednoduchého posunu a nasledujúceho testu (znamienkového) bitu. Dôsledkom uvedeného je, že operácia testu v tvare

$$x \textbf{ in } [c_1, c_2, \ldots, c_n]$$

sa dá oveľa efektívnejšie implementovať ako ekvivalentný boolovský výraz

$$(x = c_1) \vee (x = c_2) \vee \ldots \vee (x = c_n)$$

Z doterajších úvah o implementácii množinových typov vyplýva, že štruktúra množina by sa mala používať hlavne v prípade *krátkych základných typov* (v zmysle typovej kardinality). Horná hranica hodnoty kardinality základného typu, pre ktorú je ešte zaručená efektívna implementácia, sa určí z dĺžky strojového slova príslušného počítača. Je zrejmé, že z tohto hľadiska budú výhodnejšie počítače s veľkou dĺžkou strojového slova. Ak je veľkosť slova pomerne malá, tak na implementáciu typu množina sa môže použiť niekoľkoslovná reprezentácia.

1.11 SEKVENČNÝ SÚBOR

Spoločnou vlastnosťou doteraz uvedených štruktúr údajov (poľa, záznamu a množiny) je, že ich *kardinalita je konečná* (samozrejme, za predpokladu, že kardinalita typov ich prvkov je konečná). Z toho dôvodu ani nespôsobujú väčšie implementačné problémy; vhodné reprezentácie sa dajú bez ťažkosti nájsť a realizovať na každom číslicovom počítači.

Väčšina zložitých štruktúr, akými sú napr. postupnosti, stromy, grafy, zoznamy atď., sú charakteristické tým, že ich kardinalita je nekonečná. Tento rozdiel oproti základným štruktúram údajov s konečnou kardinalitou je pomerne významný a má vážne praktické dôsledky. Ako príklad uveďme nasledujúcu definíciu štruktúry postupnosť: Postupnosť so základným typom T_0 je buď prázdnou postupnosťou, alebo zreťazením postupnosti (so základným typom T_0) a hodnoty typu T_0.

Takto definovaný typ postupnosť T obsahuje nekonečne veľa hodnôt. Každá hodnota obsahuje konečný počet prvkov typu T_0, no počet týchto hodnôt je neohraničený, t.j. pre každú postupnosť možno vytvoriť väčšiu (dlhšiu) postupnosť.

Podobné úvahy platia pre všetky ostatné zložité štruktúry údajov. Dôsledok existencie takýchto štruktúr, ktorý treba brať do úvahy i pri voľbe ich reprezentácií, je, že v čase kompilácie nie je známy počet strojových slov, ktoré treba vyhradiť na reprezentáciu hodnôt typov týchto štruktúr. Veľkosť pamäti potrebnej na ich reprezentáciu sa v čase priebehu realizácie programu môže dokonca meniť. To si vyžaduje *dynamické prideľovanie pamäti* pre prípad, že hodnoty narastajú, resp. *uvoľňovanie pamäti* pre opačné prípady. Preto je reprezentácia zložitých štruktúr náročná a ťažká a výsledok jej vyriešenia má rozhodujúci vplyv na efektívnosť a ekonomiku manipulácie s pamäťou počítača. Vhodnú voľbu možno vykonať len na základe znalosti primitívnych operácií, ktoré sa majú na danej štruktúre realizovať, a frekvencie ich vykonávania. Vzhľadom na to, že tvorca programovacieho jazyka a jeho kompilátora nepozná takéto informácie, odporúča sa vylúčiť zložité štruktúry z jazyka (určeného na všeobecné účely). Podobne je rozumné, ak sa programátor vyhýba používaniu zložitých štruktúr najmä v prípade, keď sa jeho problém dá dobre vyriešiť pomocou základných štruktúr. Vo väčšine programovacích jazykov a v ich kompilátoroch sa dilema zavedenia zložitých štruktúr, ale bez informácií o možnostiach ich použitia, rieši zvážením skutočnosti, že všetky zložité štruktúry sa skladajú buď z neštruktúrovaných prvkov, alebo so základných štruktúr. Potom sa za predpokladu, že sú k dispozícii prostriedky pre dynamické prideľovanie pamäti prvkom a pre dynamické spájanie a adresovanie prvkov, dajú vytvárať ľubovoľné štruktúry, a to prostredníctvom

operácií, ktoré špecifikuje programátor. Techniky tvorby zložitých štruktúr a manipulácie s nimi sú podrobne opísané vo štvrtej kapitole.

Spomedzi zložitých štruktúr spomenieme jednu, ktorá má výnimočné postavenie, a to *postupnosť*. Tým, že jej kardinalita je nekonečná, radíme ju medzi zložité, ale vzhľadom na jej veľmi časté a rozsiahle používanie ju takmer prirodzene zaraďujeme k základným štruktúram. Aby sme mohli postupnosť abstraktne opísať, zavedieme nasledujúcu terminológiu:

1. Symbolmi $\langle\,\rangle$ označujeme prázdnu postupnosť.
2. $\langle x_0 \rangle$ označuje postupnosť, ktorá sa skladá z jedného prvku x_0; nazývame ju jednoprvková postupnosť.
3. Ak $x = \langle x_1, x_2, \ldots, x_m \rangle$ a $y = \langle y_1, y_2, \ldots, y_n \rangle$, tak

$$x \,\&\, y = \langle x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n \rangle \tag{1.41}$$

je zreťazenie postupnosti x a y.

4. Ak $x = \langle x_1, x_2, \ldots, x_n \rangle$ je prázdna postupnosť, tak

$$\text{first}\,(x) = x_1 \tag{1.42}$$

označuje prvý prvok postupnosti x.

5. Ak $x = \langle x_1, x_2, \ldots, x_n \rangle$ je neprázdna postupnosť, tak

$$\text{rest}\,(x) = \langle x_2, x_3, \ldots, x_n \rangle \tag{1.43}$$

je postupnosť x bez svojho prvého prvku.

Dôsledkom je invariantná relácia

$$\langle \text{first}\,(x) \rangle \,\&\, \text{rest}\,(x) \equiv x \tag{1.44}$$

Zavedenie týchto zápisov ešte neznamená, že sa budú dať priamo použiť v skutočných programoch realizovateľných na reálnych počítačoch. V skutočnosti je podstatné, že operácia zreťazenia sa vo všeobecnosti nepoužíva. Narábanie s postupnosťami je obmedzené na vybranú množinu operátorov, ktoré zaručujú istú disciplínu pri ich používaní a ktoré sú definované prostredníctvom abstraktných pojmov postupnosť a zreťazenie. Dôsledná voľba množiny operátorov nad postupnosťami uľahčuje implementátorom nájdenie vhodných a efektívnych reprezentácií pre štruktúru postupnosť na ktoromkoľvek pamäťovom médiu. Zaručuje to, že príslušný mechanizmus dynamického prideľova-

nia pamäti môže byť natoľko jednoduchý, že programátor pri práci s ním nemusí venovať pozornosť jeho detailom.

Aby bolo zrejmé, že štruktúra postupnosť zaradená medzi základné štruktúry údajov povoľuje iba aplikáciu tých operácií, ktoré výhradne sekvenčným spôsobom pristupujú k jej jednotlivým prvkom, nazývame ju *sekvenčný súbor* alebo skrátene *súbor*. V súlade s uvedenými definíciami typov pole a množina je typ súbor definovaný pomocou formuly

$$\textbf{type } T = \textbf{file of } T_0 \tag{1.45}$$

z ktorej vyplýva, že každý súbor typu T sa skladá z 0 alebo viacerých prvkov typu T_0.

Príklady:

type text = **file of** char
type sadaštitkov = **file of** štitok

Podstata sekvenčného prístupu spočíva v tom, že v každom okamihu je bezprostredne prístupný iba jediný, špecifický prvok postupnosti, ktorý je určený momentálnou pozíciou prístupového mechanizmu. Zmenu momentálnej pozície možno dosiahnuť pomocou operátorov typu súbor obyčajne tak, aby to bola pozícia nasledujúceho alebo prvého prvku postupnosti. Pozícia v súbore sa formálne vyjadruje tak, že súbor x pokladáme za celok, ktorý pozostáva z dvoch častí, ľavej x_L a pravej x_R. Je zrejmé, že rovnosť

$$x \equiv x_L \,\&\, x_R \tag{1.46}$$

vyjadruje invariantný vzťah.

Druhým dôležitým dôsledkom sekvenčného prístupu je, že procesy vytvárania a prehľadávania postupnosti sú rozdielne a nemôžu sa striedať v ľubovoľnom poradí. Súbor sa vytvára tak, že sa k nemu neustále pridávajú prvky (na jeho koniec). Zisťovanie výskytu konkrétneho prvku sa potom uskutočňuje sekvenčným prehľadávaním súboru (od jeho začiatku). Preto je súbor charakterizovaný jedným z dvoch stavov: buď je v stave jeho vytvárania (zápisu), alebo v stave prehľadávania (čítania).

Výhoda prísne sekvenčného prístupu sa prejavuje najmä pri súboroch, ktoré sa nachádzajú na sekundárnych pamäťových miestach, t. j. takých, ktoré vyžadujú prenos údajov medzi dvoma rôznymi médiami. Metóda sekvenčného prístupu je jediná,ktorá programátorovi umožňuje úspešné ukrytie komplikovaných mechanizmov, potrebných pri takýchto prenosoch. Predovšetkým umožňuje aplikáciu jednoduchých techník ukladania do vyrovnávacej pamäti, ktoré už samy o sebe zaručujú optimálne používanie rozličných systémových prostriedkov zložitého počítačového prostredia.

Existujú pamäťové médiá, pri ktorých je sekvenčná prístupová metóda jedinou možnou metódou prístupu. Medzi takéto médiá patria bezpochyby všetky druhy pások. Dokonca každá záznamová stopa magnetického bubna alebo disku predstavuje pamäťové médium výlučne so sekvenčným prístupom. Sekvenčný prístup je základnou charakteristikou každého počítačového zariadenia, ktoré sa mechanicky pohybuje.

1.11.1 ZÁKLADNÉ OPERÁTORY TYPU SÚBOR

V ďalšom uvedieme formuláciu abstraktného pojmu sekvenčný prístup prostredníctvom množiny základných operátorov typu súbor, ktoré má programátor k dispozícii. Tieto operátory sú definované na základe pojmov postupnosť a zreťazenie. Sú to: operátor inicializácie vytvárania súboru, operátor inicializácie prehľadávania súboru, operátor sprístupňujúci ďalší prvok postupnosti. Posledné dva operátory vyžadujú prítomnosť implicitnej pomocnej premennej, ktorá reprezentuje vyrovnávaciu pamäť. Predpokladáme, že takáto pamäť je automaticky združená s každou premennou x typu súbor, čo označujeme zápisom $x\uparrow$. Je zrejmé, že ak x je typu T, tak $x\uparrow$ je základného typu T_0.

1. Vytvorenie prázdnej postupnosti. Operácia

$$\text{rewrite}(x) \tag{1.47}$$

reprezentuje priraďovací príkaz

$$x := \langle\,\rangle$$

a používa sa na prepisanie bežnej hodnoty x a na inicializovanie proce-

su vytvárania novej postupnosti. V skutočnosti zodpovedá previnutiu pásky.

2. Rozšírenie (zväčšenie) postupnosti. Operácia

$$\text{put}(x) \tag{1.48}$$

zodpovedá priraďovaciemu príkazu

$$x := x \,\&\, \langle x\uparrow \rangle$$

pomocou ktorého sa dá efektívne pridať hodnota $x\uparrow$ k postupnosti x.

3. Inicializácia prehľadávania. Operácia

$$\text{reset}(x) \tag{1.49}$$

zodpovedá súčasným priradeniam

$$x_L := \langle\rangle$$
$$x_R := x$$
$$x\uparrow := \text{first}(x)$$

a inicializuje sa ňou proces čítania (prehľadávania) postupnosti.

Sprístupnenie ďalšieho prvku postupnosti. Operácia

$$\text{get}(x) \tag{1.50}$$

reprezentuje súčasné priradenia

$$x_L := x_L \,\&\, \langle \text{first}(x_R) \rangle$$
$$x_R := \text{rest}(x_R)$$
$$x\uparrow := \text{first}(\text{rest}(x_R))$$

Poznamenávame, že first(s) je definované iba vtedy, keď $s \neq \langle\rangle$.

Operátory rewrite a reset, pochopiteľne, nezávisia od pozície, v ktorej sa súbor nachádza pred ich vykonaním. V každom prípade tieto dve operácie nastavia súbor na jeho začiatok.

Pri prehľadávaní postupnosti je dôležité rozpoznať jej koniec, pretože inač priraďovací príkaz

$$x\uparrow := \text{first}(x_R)$$

môže reprezentovať nedefinovanú operáciu. Dosiahnutie konca súboru vlastne zodpovedá podmienke, že pravá časť súboru x_R je prázdna.

Preto sa zavádza predikát

$$\text{eof}(x) \equiv x_R = \langle\rangle \tag{1.51}$$

vyjadrujúci skutočnosť, že sa dosiahol koniec súboru. Operáciu get(x) možno aplikovať len v tom prípade, ak predikát eof(x) má hodnotu false.

V princípe je možné vyjadriť všetky operácie so súbormi pomocou uvedených štyroch základných operátorov. V praxi sa najčastejšie kombinujú operácie, ktoré menia momentálnu pozíciu súboru (get alebo put) s operáciami prístupu k vyrovnávacej premennej (súboru). Na základe toho definujeme ďalšie dve procedúry, ktoré sa dajú takisto vyjadriť pomocou základných operátorov. Nech v je premenná a e výraz typu T_0, pričom T_0 je typ prvku súboru. Potom

$$\text{read}(x, v)$$

je analogické ako

$$v := x\uparrow; \quad \text{get}(x)$$

a

$$\text{write}(x, e)$$

ako

$$x\uparrow := e; \quad \text{put}(x)$$

Výhoda použitia operácií read a write namiesto operácií get a put je okrem stručnosti zápisu aj v koncepčnej jednoduchosti. Vzhľadom na to môžeme v rámci našich ďalších úvah zabudnúť na existenciu vyrovnávacej premennej $x\uparrow$, ktorej hodnota je v niektorých prípadoch nedefinovateľná. Napriek tomu môže byť vyrovnávacia premenná osožná napr. pri predsnímaní (pri ktorom možno zistiť hodnotu nasledujúceho prvku postupnosti bez jeho bezprostredného sprístupnenia — pozn. prekl.).

Nevyhnutnými podmienkami vykonania uvedených dvoch procedúr sú:

$$\neg\,\text{eof}(x) \quad \text{pre} \quad \text{read}(x, v)$$
$$\text{eof}(x) \quad \text{pre} \quad \text{write}(x, e)$$

Pri prehľadávaní súboru nadobúda predikát eof(x) hodnotu true hneď, ako sa prečíta posledný prvok súboru x. Uvedené úvahy sú prehľadne

obsiahnuté vo dvoch programových schémach pre sekvenčné vytvára-nie a spracovanie súboru x. Príkazy R a S a predikát p predstavujú dodatočné parametre schém:

Zápis do súboru x:

```
rewrite(x);                                   (1.52)
while p do
  begin R(v); write(x, v)
  end
```

Čítanie súboru x:

```
reset(x);
while ¬ eof(x) do                             (1.53)
  begin read(x, v); S(v)
  end
```

1.11.2 ŠTRUKTÚROVANIE SÚBOROV

Väčšina aplikácií vyžaduje, aby veľké súbory boli nejakým spôso-bom štruktúrované. Napríklad aj kniha, ktorú možno považovať za postupnosť znakov, je rozdelená na kapitoly a články. Zmyslom štruk-túrovania (delenia na podštruktúry) je poskytnúť nejaké orientačné body, nejaké súradnice, aby sa uľahčila orientácia v dlhej postupnosti informácií. Existujúce pamäťové zariadenia často poskytujú na repre-zentáciu takýchto orientačných miest určité prostriedky (napr. značky na magnetických páskach) a sú schopné nastaviť médium na určené miesto oveľa rýchlejšie ako postupným čítaním informácií medzi mo-mentálnym a určeným miestom.

V našom ďalšom príklade, v ktorom uvedieme jednoúrovňovú pod-štruktúru, pokladajme súbor za postupnosť prvkov, ktoré sú opäť súbormi. Máme teda na mysli súbor súborov. Ak predpokladáme, že základné prvky (alebo jednotky) sú typu U, tak podštruktúry sú typu

$$T' = \textbf{file of } U$$

a celý súbor je typu

$$T = \textbf{file of } T'$$

Z uvedeného je zrejmé, že takýmto spôsobom môžu byť súbory členené do ľubovoľnej hĺbky (vkladania do seba). Vo všeobecnosti typ T_n môže byť definovaný pomocou rekurzívneho vzťahu

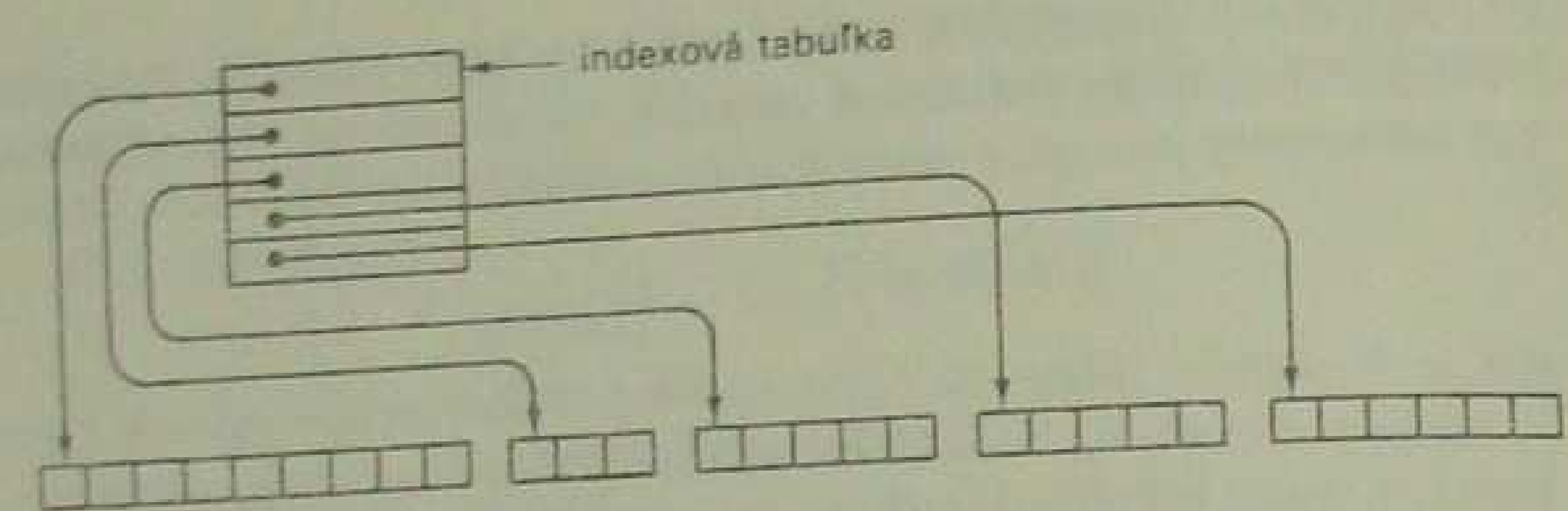
$$T_i = \textbf{file of } T_{i-1}, \qquad i = 1, \ldots, n$$

a $T_0 = U$. Takéto súbory sa často nazývajú *viacúrovňové* a prvok typu T_i *segment* úrovne i. Príkladom takéhoto viacúrovňového súboru je kniha, v rámci ktorej jednotlivým úrovniam segmentácie zodpovedajú kapitoly, články, odseky a riadky. Najbežnejšie sú však súbory s jedno-úrovňovým členením. Takéto súbory však nijako nemožno stotožňovať s pojmom pole súborov. Veď už počet segmentov je premenlivý, navyše súbor možno na konci predlžovať. V súlade s našou doterajšou symbo-likou predpokladajme, že máme definovaný súbor

$$x : \textbf{file of file of } U$$

Potom $x\uparrow$ označuje momentálne prístupný segment a $x\uparrow\uparrow$ momentálne prístupný prvok. Podobne, put($x\uparrow\uparrow$) a get($x\uparrow\uparrow$) sa vzťahujú na prvok segmentu, pričom put($x\uparrow$) a get($x\uparrow$) znamenajú operácie pridania (zápisu) segmentu a prečítania ďalšieho segmentu.

Segmentované súbory sa dajú ľahko implementovať takmer na všet-kých sekvenčných pamäťových zariadeniach, vrátane pások. Segmentá-cia súborov nezmení ich primárnu charakteristiku, ktorou je výlučne sekvenčný prístup, a to buď k jednotlivým prvkom, alebo prostredníc-tvom rýchlejšieho mechanizmu preskoku k jednotlivým segmentom. Iné pamäťové zariadenia, menovite magnetické bubny a disky, obyčaj-ne obsahujú určitý počet stôp, z ktorých každá je sekvenčným zariade-ním, ale často kapacitne nestačí na obsiahnutie celého súboru. Preto sa súbory na diskoch obyčajne nachádzajú na viacerých stopách, pričom obsahujú informácie potrebné na spájanie jednotlivých častí súboru, ktoré sú na separátnych stopách. Je zrejmé, že začiatok každej stopy je vlastne značkou segmentu umožňujúcou priamejší prístup, ako je to v prípadoch čisto sekvenčných médií. Indexová tabuľka umiestnená v primárnej pamäti by sa napr. mohla použiť na adresovanie stôp, na ktorých začínajú segmenty a určenie aktuálnej dĺžky segmentov (*obr. 1.10*).



Obr. 1.10. Indexový súbor s piatimi segmentmi

Na základe takýchto úvah sme dospeli k pojmu *indexový súbor* (niekedy sa tiež nazýva *súbor s priamym prístupom*). Skutočne, bubny a disky sú organizované takým spôsobom, že každá stopa obsahuje viaceré značky, od ktorých môže začať proces čítania alebo zápisu. Vzhľadom na to segment nemusí zaberať celú stopu, pretože to môže viesť k zlému využitiu pamäti najmä vtedy, ak sú segmenty kratšie ako dĺžka stopy. Časť pamäti medzi dvoma značkami sa nazýva *fyzický segment* (alebo *sektor*) na rozdiel od *logického segmentu*, ktorý je významným pojmom medzi štruktúrami údajov programu. Platí, že každý fyzický segment obsahuje najviac jeden logický segment a každý logický segment zaberá aspoň jeden fyzický segment (dokonca aj v prípade, že je prázdny). Bude dobré, ak si uvedomíme, že aj keď sa tieto súbory nazývajú súbory s priamym prístupom, priemerný čas potrebný na nastavenie sa na príslušný segment, tzv. *latentný* čas alebo *čakacia doba*, sa rovná polovičnej dobe jednej otáčky disku.

Indexové súbory zachovávajú základnú charakteristiku, t. j. proces zápisu postupuje sekvenčným spôsobom až na jeho koniec. Preto sú výhodné najmä pri tých aplikáciách, v rámci ktorých sa zmeny vykonávajú v relatívne veľkých časových intervaloch. Zmeny sa uskutočňujú buď zväčšením, alebo prekopírovaním a aktualizovaním celého súboru. Kontroly možno vykonávať oveľa kvalitnejšie a rýchlejšie prostredníctvom indexových bodov, čo je typické v prípade *banky údajov*.

Systémy, ktoré umožňujú výberový zápis úsekov do stredu súboru, sú vo všeobecnosti zložité a ich použitie je riskantné, pretože nový záznam musí byť rovnako dlhý ako ten, ktorý sa prepisuje. Okrem toho v aplikáciách, týkajúcich sa veľkých súborov údajov, sa výberové aktualizovanie neodporúča. Dôvodom je, že po každom zlyhaní, či už na základe chybného programu, alebo kvôli nefungujúcemu technickému vybaveniu treba obnoviť stav údajov a zopakovať proces, ktorý zlyhal. Vzhľadom na to sa aktualizácia súboru robí vcelku, a to takým spôsobom, že starý súbor sa nahradí novým, zaktualizovaným, až vtedy, keď sa overí jeho bezchybnosť. Z hľadiska spoľahlivosti je sekvenčná organizácia oveľa lepšia na účely aktualizácie. Jej používanie odporúčame i napriek tomu, že existujú mnohé dômyselnejšie organizácie veľkých súborov údajov, ktoré môžu byť aj efektívnejšie, ale v prípade zlyhania technického vybavenia často spôsobujú stratu údajov.

1.11.3 TEXTY

Súbory s prvkami typu char zohrávajú pri výpočtoch a spracovaní údajov zvlášť dôležitú úlohu. Predstavujú prepojenie medzi počítačmi a ich používateľmi. Čitateľný vstup vytvorený programátormi, ako aj čitateľný výstup reprezentujúci výsledky počítačového spracovania sú postupnosťami znakov. Takémuto typu údajov dáme štandardný názov

type text = file of char

Komunikáciu medzi výpočtovým procesom a jeho používateľom tvorí prepojenie, ktoré môže byť reprezentované dvoma textovými súbormi. Prvý z nich obsahuje *vstup* do výpočtového procesu (súbor nazveme *input*), druhý — *výstup* — výsledky výpočtu (nazveme ho *output*). V našich ďalších úvahách budeme vo všetkých programoch predpokladať existenciu týchto dvoch súborov a ich deklaráciu:

var input, output: text

V súlade s predpokladom, že tieto súbory reprezentujú štandardné vstupné a výstupné médium počítačového systému (ako napr. snímač diernych štítkov a riadkovú tlačiareň), budeme predpokladať, že na súbor input bude možné aplikovať iba operácie čítania a na súbor output operácie zápisu.

Vzhľadom na to, že sa tieto dva štandardné súbory používajú najčastejšie, urobíme nasledujúcu dohodu: Ak prvý parameter procedúr read a write nebude premenná typu súbor, automaticky budeme predpokla-

dať súbory input a output. Naviac pri týchto dvoch procedúrach dovolime ľubovoľný počet parametrov. Spomenuté dohody zápisu môžeme zhrnúť nasledujúcim spôsobom:

read $(x_1, x_2, \ldots, x_n)$ je ekvivalentné s read (input, $x_1, x_2, \ldots, x_n$)
write $(x_1, x_2, \ldots, x_n)$ je ekvivalentné s write (output, $x_1, x_2, \ldots, x_n$)
read $(f, x_1, x_2, \ldots, x_n)$ je ekvivalentné s
begin read (f, x_1); read (f, x_2); ... read (f, x_n) **end**
write $(f, x_1, x_2, \ldots, x_n)$ je ekvivalentné s
begin write (f, x_1); write (f, x_2); ... write (f, x_n) **end**

Texty sú typickými príkladmi postupnosti zobrazujúcich podštruktúru. Zvyčajnými jednotkami podštruktúry sú kapitoly, články, odseky a riadky. Na zobrazenie podštruktúry v textových súboroch sa často používa metóda špeciálnych oddeľovacích znakov. Prázdny znak (medzera) je najznámejším príkladom, ale podobné oddeľovacie znaky sa dajú použiť aj na označenie koncov riadkov, odsekov, článkov a kapitol. Napríklad niekoľko takých znakov, nazývaných *riadiace znaky* (pozri prílohu A), obsahuje najpoužívanejšia množina znakov ISO (vrátane jej americkej verzie ASCII).

V tejto knihe nebudeme používať špeciálne oddeľovacie znaky ani špecifikácie presnej metódy reprezentácie podštruktúry. Namiesto toho budeme uvažovať o textoch ako o súboroch obsahujúcich postupnosti znakových postupností reprezentujúcich jednotlivé riadky. Podobne našu diskusiu obmedzíme iba na jednu úroveň podštruktúry, konkrétne na riadok. Pretože sa na texty nepozeráme ako na súbory súborov tlačiteľných znakov, ale ako na súbory znakov, zavádzame ďalšie operátory a predikáty slúžiace na manipuláciu, t. j. označenie a rozpoznanie riadkov. Ich efekt možno najlepšie pochopiť, ak si predstavíme, že riadky sú oddelené (hypotetickými) oddeľovacími znakmi (ktoré nepatria do typu char) a ich úlohou je vložiť a rozpoznať (nájsť) takýto oddeľovač.

Ďalšími operátormi sú:

writeln (f) Zapíš znak konca riadka do súboru f.
readln (f) Preskoč znaky súboru f až po najbližší znak, ktorý nasleduje po poslednom znaku riadka.

eoln (f) Boolovská funkcia. Vráti hodnotu true, ak sa prečítal posledný znak riadka, v opačnom prípade vráti hodnotu false. (Tvrdíme, že ak eoln (f) je true, tak $f\uparrow$ je prázdny znak.)

Uveďme teraz dve programové schémy pre zápis a čítanie textov, ktoré sú podobné schémam na zápis a čítanie všeobecných súborov (pozri vzťahy (1.52) a (1.53)). V rámci týchto schém predpokladáme existenciu textového súboru f a sústredíme sa na tvorbu a rozpoznávanie štruktúry riadka. Nech $R(x)$ je príkaz priraďujúci hodnotu premennej x (ktorá je typu char) a p a q sú podmienky znamenajúce: toto bol posledný znak v riadku a toto bol posledný znak súboru. Nech U je príkaz, ktorý sa má vykonať pred začatím čítania jednotlivých znakov riadka, $S(x)$ je príkaz, ktorý sa má uskutočniť po prečítaní každého znaku súboru, a V je príkaz, ktorý sa vykoná po prečítaní posledného znaku riadka.

Zápis do textového súboru f:

```
rewrite (f);
while ¬ q do
  begin
    while ¬ p do
      begin R(x); write (f, x)                    (1.54)
      end;
    writeln (f)
  end
```

Čítanie textového súboru f:

```
reset (f);
while ¬ eof (f) do
  begin U;
    while ¬ eoln (f) do                           (1.55)
      begin read (f, x); S(x)
      end
    V; readln (f)
  end
```

Existujú prípady, keď riadková štruktúra textu nereprezentuje žiadnu dôležitú informáciu. V takýchto situáciách je možné použiť jednoduchú programovú schému vytvorenú na základe našich predpokladov, týkajúcich sa hodnoty vyrovnávacej premennej pri prečítaní posledného znaku riadka (pozri definíciu eoln(f)). Uvedomme si, že v súlade s definíciou eoln sa každý koniec riadka prejaví ako dodatočný prázdny znak.

```
while ¬ eof(f) do
  begin read(f, x); S(x)                    (1.56)
  end
```

Väčšina programovacích jazykov dovoľuje v procedúrach read a write použiť argumenty typu integer i real. Takéto zovšeobecnenie by bolo priamočiare, ak by typy integer a real boli definované ako polia znakov, ktorých prvky by reprezentovali jednotlivé číslice čísel. Jazyky, ktoré sa používajú v komerčných aplikáciách, skutočne tieto definície dodržiavajú a vyžadujú, aby čísla boli reprezentované desiatkovými číslicami a desiatkovou číselnou sústavou. Významnou výhodou zaradenia typov integer a real medzi základné typy je, že takéto podrobné špecifikácie sa dajú vynechať a výpočtový systém môže použiť rôzne reprezentácie pre čísla (také, ktoré sú najvhodnejšie pre dané aplikácie). Napríklad systémy orientované na vedecko-technické výpočty používajú výhradne dvojkovú reprezentáciu, ktorá je z viacerých hľadísk výhodnejšia ako desiatková. Z uvedeného však vyplýva, že programátor nemôže predpokladať možnosť priameho čítania čísel z textových súborov, resp. ich zápis do textov bez príslušných konverzných operácií. Zvyčajne sa tento problém rieši tak, že konverzné operácie tvoria súčasť procedúr read a write, ak ich argumenty sú numerických typov. Profesionálny programátor, samozrejme, vie, že takéto príkazy (nazývame ich V/V príkazy, t.j. *vstupno-výstupné príkazy*) realizujú dve rozdielne funkcie: prenos údajov medzi rôznymi pamäťovými médiami a konverziu reprezentácií údajov. Druhá funkcia môže byť oveľa zložitejšia a časovo náročnejšia.

V ďalších kapitolách tejto knihy budeme procedúry read a write pre numerické argumenty používať podľa pravidiel definovaných programovacím jazykom pascal. Tieto pravidlá dovoľujú, pre istý druh špeci-

fikácií formátu, riadiť proces transformácie údajov. Špecifikáciou formátu sa určuje napr. počet požadovaných číslic v prípade príkazu write. Tento počet znakov, nazývaný aj „rozsah" alebo „šírka poľa", treba uviesť hneď za parametrom určujúcim hodnotu, ktorá sa má zapísať do súboru, a to nasledujúcim spôsobom:

$$\text{write}(f,\ x : n)$$

Do súboru f zapisujeme hodnotu argumentu x; jeho hodnota je premenná na postupnosť (aspoň) n znakov a ak treba, tak sa pred číslice zaradí znamienko a potrebný počet prázdnych znakov. Zmieňovať sa o ďalších detailoch nasledujúcich programov (z hľadiska zrozumiteľnosti) nie je potrebné. Najprv uvedieme dva príklady procedúr, ktoré realizujú konverzie medzi číselnými reprezentáciami (sú to programy 1.3 a 1.4). Chceme pomocou nich poukázať na zložitosť takých operácií, ktoré sú implicitne predpokladané v procedúre write. Tieto procedúry uskutočňujú konverzie reálnych čísel z desiatkového tvaru do vhodného „vnútorného" tvaru a obrátene. (Konštanty uvedené v záhlaví programu sú určené na základe formátu čísel s pohyblivou rádovou čiarkou počítača CDC 6000, t.j. 11 bitov pre dvojkový exponent a 48 bitov pre mantisu. Funkcia expo(x) znamená exponent premennej x.)

PROGRAM 1.3. *Prečítanie reálneho čísla*

```
procedure čítajreal (var f: text; var x: real);
  {prečíta reálne číslo x zo súboru f}
  {nasledujúce konštanty sú závislé od počítačového systému}
  const t48 = 281474976710656;   {= 2 ** 48}
        limit = 56294995342131;  {= t48 div 5}
        z = 27;                  {= ord('0')}
        lim1 = 322;              {najväčší exponent}
        lim2 = -292;             {najmenší exponent}
  type možint = 0..323;
  var ch: char; y: real; a, i, e: integer;
      s, ss: boolean;            {znamienka}
  function desať(e: možint): real;  {= 10 ** e, 0 < e < 322}
    var i: integer; t: real;
```

```
begin i: = 0; t: = 1.0;
  repeat if odd (e) then
    case i of
    0: t: = t * 1.0E1;
    1: t: = t * 1.0E2;
    2: t: = t * 1.0E4;
    3: t: = t * 1.0E8;
    4: t: = t * 1.0E16;
    5: t: = t * 1.0E32;
    6: t: = t * 1.0E64;
    7: t: = t * 1.0E128;
    8: t: = t * 1.0E256
    end
    e: = e div 2; i: = i + 1
  until e = 0;
  desať: = t
end;
begin
  {preskoč prázdne znaky}
  while f↑ = ' ' do get (f);
  ch: = f↑;
  if ch = '−' then
    begin s: = true; get (f); ch: = f↑
    end else
    begin s: = false;
      if ch = '+' then
      begin get (f); ch: = f↑
      end
    end;
  if ¬ (ch in ['0' .. '9']) then
  begin správa ('očakáva sa číslica'); halt
  end
  a: = 0; e: = 0;
  repeat if a < limit then a: = 10 * a + ord (ch) − z else e: = e + 1;
    get (f); ch: = f↑
  until ¬ (ch in ['0' .. '9']);
```



```
if ch = '.' then
begin {prečítaj desatinnú časť} get (f); ch: = f↑;
  while ch in ['0' .. '9'] do
  begin if a < limit then
        begin a: = 10 * a + ord (ch) − z; e: = e − 1
        end;
      get (f); ch: = f↑
  end
end;
  if ch = 'E' then
  begin {prečítaj mierku} get (f); ch: = f↑;
    i: = 0;
    if ch = '−' then
    begin ss: = true; get (f); ch: = f↑
    end else
    begin ss: = false; if ch = '+' then
      begin get (f); ch: = f↑
      end
    end;
    while ch in ['0' .. '9'] do
    begin if i < limit then begin i: = 10 * i + ord (ch) − z
                           end;
      get (f); ch: = f↑
    end;
    if ss then e: = e − i else e: = e + i
    end;
    if e < lim 2 then
      begin a: = 0; e: = 0
      end else
    if e > lim1 then
    begin správa ('príliš veľké číslo'); halt
    end;
    {0 < a < 2 ** 49}
    if a ≧ t48 then y: = ((a + 1) div 2) * 2.0 else y: = a;
    if s then y: = −y;
    if e < 0 then x: = y/desať (−e) else
```

```
    if e ≠ 0 then x := y * desaf(e) else x := y;
    while (f↑ = ' ') ∧ (¬ eof(f)) do get(f);
end {čítajreal}
```

PROGRAM 1.4. *Zápis reálneho čísla*

```
procedure zapíšreal (var f: text; x: real; n: integer);
  {zapíše reálne číslo x s dĺžkou n číslic
   vo formáte pohyblivej rádovej čiarky}
  {nasledujúce konštanty závisia od reprezentácie
   reálnych čísel s pohyblivou rádovou čiarkou}
  const t48 = 281474976710656; {= 2 ** 48; 48 = veľkosť mantisy}
        z = 27;              {ord('0')}
  type možint = 0..323; {rozsah desiatkového exponenta}
  var c, d, e, e0, e1, e2, i: integer;
  function desaf (e: možint): real; {10 ** e, 0 < e < 322}
    var i: integer; t: real;
  begin i := 0; t := 1.0;
    repeat if odd(e) then
      case i of
      0: t := t * 1.0E1;
      1: t := t * 1.0E2;
      2: t := t * 1.0E4;
      3: t := t * 1.0E8;
      4: t := t * 1.0E16;
      5: t := t * 1.0E32;
      6: t := t * 1.0E64;
      7: t := t * 1.0E128;
      8: t := t * 1.0E256;
      end
      e := e div 2; i := i + 1
    until e = 0;
    desaf := t
  end {desaf};
  begin {treba aspoň 10 znakov: b + 9.9E + 999}
    if x = 0 then
    begin repeat write(f, ' '); n := n - 1
          until n ≦ 1;
    write(f, '0')
    end else
    begin
      if n ≦ 10 then n := 3 else n := n - 7
      repeat write(f, ' '); n := n - 1
      until n ≦ 15;
      {1 < n ≦ 15, počet číslic, ktoré sa budú tlačiť}
      begin {zistenie znamienka, určenie exponenta}
        if x < 0 then
        begin write(f, '-'); x := -x
        end else write(f, ' ');
      e := expo(x); {e = entier(log2(abs(x)))}
      if e ≧ 0 then
        begin e := e * 77 div 256 + 1; x := x/desaf(e);
          if x ≧ 1.0 then
            begin x := x/10.0; e := e + 1
            end
        end else
        begin e := (e + 1) * 77 div 256; x := desaf(-e) * x;
          if x < 0.1 then
            begin x := 10.0 * x; e := e - 1
            end
        end;
      {0.1 ≦ x < 1.0}
      case n of {zaokrúhlenie}
        2: x := x + 0.5E - 2;
        3: x := x + 0.5E - 3;
        4: x := x + 0.5E - 4;
        5: x := x + 0.5E - 5;
        6: x := x + 0.5E - 6;
        7: x := x + 0.5E - 7;
        8: x := x + 0.5E - 8;
        9: x := x + 0.5E - 9;
       10: x := x + 0.5E - 10;
```

```
        11: x: = x + 0.5E − 11;
        12: x: = x + 0.5E − 12;
        13: x: = x + 0.5E − 13;
        14: x: = x + 0.5E − 14;
        15: x: = x + 0.5E − 15
      end;
      if x ≧ 1.0 then
          begin x: = x * 0.1; e: = e + 1;
          end;
      c: = trunc(x, 48); {= trunc(x * (2 ** 48))}
      c: = 10 * c; d: = c div t48;
      write(f, chr(d + z), '.');
      for i: = 2 to n do
      begin c: = (c − d * t48) * 10; d: = c div t48;
          write(f, chr(d + z))
      end;
      write(f, 'E'); e: = e − 1;
      if e < 0 then
          begin write(f, '−'); e: = −e
          end else write(f, '+');
      e1: = e * 205 div 2048; e2: = e − 10 * e1;
      e0: = e1 * 205 div 2048; e1: = e1 − 10 * e0;
      write(f, chr(e0 + z), chr(e1 + z), chr(e2 + z))
    end
  end
end {zapíšreal}
```

1.11.4 PROGRAM NA EDITOVANIE SÚBORU

Použitie sekvenčných štruktúr ukážeme na ďalšom príklade, ktorý nám súčasne poslúži aj ako ukážka metódy tvorby a vysvetlenia zmyslu programov. Táto metóda sa nazýva *postupné zjemňovanie* [1.4, 1.6] a v tejto knihe ju použijeme na objasnenie mnohých algoritmov.

Program, ktorý chceme vytvoriť, by mal riešiť problematiku úpravy textu *x* na text *y*. Úprava súboru (editovanie) znamená rušenie alebo

nahradenie určitých riadkov alebo vkladanie nových riadkov textu. Proces úpravy súboru je riadený postupnosťou *editovacích inštrukcií*, ktoré sú umiestnené na štandardnom vstupnom textovom súbore input v tomto tvare:

I, m.	Vloženie textu za *m*-tý riadok.
D, m, n.	Zrušenie riadkov *m* až *n*.
R, m, n.	Nahradenie riadkov *m* až *n*.
E.	Ukončenie procesu úpravy súboru.

Každá inštrukcia sa nachádza na samostatnom riadku štandardného vstupného súboru, ktorý nazývame súbor inštrukcií; *m* a *n* sú desiatkové čísla riadkov upravovaného súboru. Vkladané texty nasledujú bezprostredne za inštrukciami *I* a *R*, ukončené sú prázdnym riadkom.

Pretože predpokladáme prísne *sekvenčné spracovanie* vstupného textového súboru *x*, požadujeme, aby upravovacie inštrukcie mali vzostupne usporiadané čísla riadkov. Je jasné, že stav spracovania musí byť charakterizovaný momentálnou pozíciou súboru *x*, určenou číslom riadka, ktorý sa práve upravuje.

Predpokladajme, že upravovací program (editor) má byť použitý v interaktívnom režime práce, a preto editovacie inštrukcie (t. j. súbor inštrukcií) vytvára programátor prostredníctvom klávesnice terminálu, čo, samozrejme, vyžaduje, aby programátor dostával spätné informácie od systému, s ktorým pracuje (v tomto prípade od editora). V našom prípade je vhodnou a užitočnou spätnou informáciou riadok zdrojového textu programu, ktorý práve editujeme. Tento riadok nazývame momentálny riadok. Zo systémového hľadiska sa na reprezentáciu momentálneho riadka používa explicitná premenná, ktorá po každej inštrukcii obsahuje práve načítaný riadok súboru a umožňuje jednak jeho okamžité zobrazenie na obrazovke terminálu, a tiež nasledujúci zápis do súboru *y*. Túto techniku nazývame predsnímanie. Formálne môžeme algoritmus editovacieho programu uviesť takto:

```
program editor(x, y, input, output);
var lno: integer;   {číslo bežného riadku}
    cl: riadok;     {bežný riadok}
    x, y:           text;
```

```
begin
    prečítaj-inštrukciu;                                  (1.57)
  repeat
      interpretuj-inštrukciu;
      zapiš-riadok;
      prečítaj-inštrukciu
  until inštrukcia = 'E'
end
```

Pristúpme teraz k detailnejším špecifikáciám jednotlivých príkazov uvedeného algoritmu. Príkazy typu prečítaj-inštrukciu a interpretuj-inštrukciu sa vo všeobecnosti skladajú z troch častí: z kódu inštrukcie a z dvoch parametrov. Zavedieme preto tri premenné, *kód*, *m* a *n*, ktoré v ďalšom využijeme na komunikáciu medzi dvoma procedúrami.

```
var kód, ch: char;
    m, n: integer
```

Príkaz prečítaj-inštrukciu bude po zjemnení vyzerať takto:

```
read (kód, ch);
if ch = ',' then read (m, ch)  else m: = lno;          (1.58)
if ch = ',' then read (n)      else n: = m
```

Všimnime si premenlivý počet parametrov procedúry read (0, 1 alebo 2). Chýbajúci parameter označujúci vstupný súbor dostane automaticky štandardnú hodnotu input.

Príkaz interpretuj-inštrukciu bude po zjemnení vyzerať takto:

```
skopíruj-riadky;
if kód = 'I' then
begin zapiš-riadok-do-y;   {do súboru y}
      vlož riadky;         {do súboru y}
end else
if kód = 'D' then preskoč-riadky else                   (1.59)
begin
    vlož-riadky;
    preskoč-riadky
end else
if kód = 'E' then skopíruj-zvyšné-riadky else chyba.
```


V druhom kroku zjemnenia vyjadríme príkazy skopíruj-riadky, vlož-riadky a preskoč-riadky, uvedené v (1.59), pomocou operácií manipulujúcich už len s riadkami. Sú to operácie prečítaj-riadok-z-*x* a zapiš-riadok-do-*y*. Ich spoločnou charakteristikou je riadiaca štruktúra cyklus. Operácia skopíruj-riadky kopíruje riadok po riadku zo súboru *x* do súboru *y*. Začína sa od momentálneho riadka a pokračuje až po *m*-tý riadok. Operácia preskoč-riadky číta riadky zo súboru *x* bez zápisu do súboru *y* až po *n*-tý riadok.

```
skopíruj-riadky: while lno < m do
                 begin
                     zapiš-riadok-do-y;
                     prečítaj-riadok-z-x
                 end
preskoč-riadky:  while lno < n do prečítaj-riadok-z-x
vlož-riadky:     prečítaj-riadok-z-terminálu;
                 while nie-je-koniec do                       (1.60)
                   begin
                       zapiš-riadok-do-y;
                       prečítaj-riadok-z-terminálu
                   end;
                 prečítaj-riadok-z-x;
skopíruj-zvyšné-riadky: while ¬ eof(x) do
                        begin
                            zapiš-riadok-do-y;
                            prečítaj-riadok-z-x
                        end;
                        zapiš-riadok-do-y
```

V treťom a poslednom, kroku zjemňovania vyjadríme operácie prečítaj-riadok-z-*x*, zapiš-riadok-do-*y*, prečítaj-riadok-z-terminálu a zapiš-riadok-na-terminál prostredníctvom operácií s jednotlivými znakmi. V rámci doterajších úvah sa všetky operácie týkali celých riadkov textu a nerobili sme žiadne špeciálne predpoklady vzhľadom na podštruktúru riadku. Vieme, že riadky sú tvorené postupnosťami znakov. Bolo by preto prirodzené deklarovať premennú *cl* (obsahujú znaky momentálneho riadka) ako postupnosť znakov, t. j.

```
var cl: file of char
```

My sa však budeme riadiť pravidlom, ktoré tvrdí, že pokiaľ sa dá použiť vhodná základná štruktúra údajov (ako napr. pole), nikdy nepoužívame štruktúru s nekonečnou kardinalitou. Pre náš prípad bude štruktúra pole plne vyhovujúca. Ak obmedzíme dĺžku riadka na 80 znakov, tak môžeme premennú *cl* definovať takto:

```
var cl: array [1..80] of char
```

Štyri spomenuté operácie používajú na manipuláciu s poľom *cl* indexovú premennú *i*, ktorá vystupuje ako lokálna premenná v rámci všetkých štyroch operácií a mohla by byť lokálne deklarovaná v každej z nich. Okrem premennej *i* potrebujeme definovať aj globálnu premennú *L*, ktorá označuje dĺžku momentálneho riadka.

```
prečítaj-riadok-z-x:  i := 0; lno := lno + 1;
                      while ¬ eoln(x) do
                        begin
                           i := i + 1; read(x, cl[i])
                        end;
                      L := i; readln(x)
zapíš-riadok-do-y:    i := 0
                      while i < L do
                        begin
                           i := i + 1; write(y, cl[i])
                        end;
                      writeln(y)                                  (1.61)
prečítaj-riadok-z-terminálu:  i := 0;
                              while ¬ eoln(input) do
                                begin
                              i := i + 1; read(cl[i])
                                end;
                              readln
zapíš-riadok-na-terminál:     i := 0; write(lno);
                              while i < L do
                                begin
                                   i := i + 1; write(cl[i])
                                end;
                              writeln
```


Podmienku nie-je-koniec v operácii vlož-riadky možno jednoducho vyjadriť ako $L \neq 0$.

Toto bol súčasne posledný krok v rámci procesu programu na editovanie textu.

Cvičenia

1.1. Predpokladajte, že kardinality štandardných typov integer, real a char sú označené symbolmi c_I, c_R a c_C. Aké budú hodnoty kardinalít typov údajov pohlavie, boolean, deň, písmeno, číslica, dôstojník, riadok, alfa, komplexnéčíslo, dátum, osoba, súradnica, znakmnožina, stavpásky, ktoré boli priebežne uvádzané v rámci príkladov tejto kapitoly?

1.2 Zvoľte reprezentáciu pre premenné typov, uvedených v cvičení 1.1
   a) v pamäti vášho počítača,
   b) v programovacom jazyku fortran,
   c) vo vašom obľúbenom programovacom jazyku.

1.3. Zistite, aké budú postupnosti inštrukcií (na vašom počítači) pre
   a) operácie vybratia a uloženia prvkov zhustených záznamov a polí,
   b) množinové operácie vrátane zistenia množinovej príslušnosti.

1.4. Dá sa skontrolovať správnosť použitia záznamovej štruktúry s variantmi v čase realizácie programu? Možno ju dokonca overiť v čase kompilácie?

1.5. Aké sú dôvody definovania niektorých množín údajov ako sekvenčných súborov namiesto polí?

1.6. Predpokladajte, že máte implementovať sekvenčné súbory, v súlade s ich definíciou v článku 1.11, na počítači s veľmi veľkou primárnou pamäťou. Nech súčasne platí, že dĺžka súborov nikdy neprekročí dĺžku *L*. Na ich reprezentáciu teda môžete použiť polia. Opíšte možnú implementáciu, ktorá má zahŕňať zvolené

reprezentácie údajov a procedúry pre základné operátory so súbormi get, put, reset a rewrite, ktoré sú definované pomocou množiny axióm v článku 1.11.

1.7. Aplikujte cvičenie 1.6 na segmentované súbory.

1.8. Máme cestovný poriadok pre železnice, ktorý udáva denné spoje medzi jednotlivými stanicami železničného systému. Nájdite vhodné štruktúry údajov (napr. pole, záznam alebo súbor), ktoré by sa dali použiť na reprezentáciu pojmov súvisiacich so železničným systémom (odchody a príchody vlakov z/do konkrétnych staníc v závislosti od zvolených vlakových tratí).

1.9. Daný je textový súbor T a dve polia A a B obsahujúce malý počet slov. Predpokladajte, že tieto slová sú reprezentované krátkymi poľami znakov, ktorých maximálna dĺžka je malá a pevná (nemenná). Napíšte program, ktorý zmení text T na text S tak, že každý výskyt slova A_i nahradí zodpovedajúcim slovom B_i.

1.10. Aké úpravy (napr. zmeny hodnôt niektorých konštánt atď.) treba urobiť v programoch 1.3 a 1.4, ak by ste ich chceli prispôsobiť na váš počítač?

1.11. Napíšte procedúru podobnú programu 1.4, ktorej záhlavie je takéto:

```
procedure zapíšreal (var f: text; x: real; n, m: integer);
```

Predpokladá sa, že táto procedúra má realizovať konverziu hodnoty x na postupnosť aspoň n znakov (tieto sa majú zapísať do súboru f) reprezentujúcich x v desiatkovom tvare s pevnou rádovou čiarkou. Počet číslic za desatinnou bodkou je určený hodnotou parametra m. V prípade potreby doplňte (zaraďte pred číslo) potrebný počet prázdnych znakov a znamienko.

1.12. Prepíšte program na úpravu súborov (editor), uvedený v odseku 1.11.4, na tvar úplného programu.

1.13. Porovnajte nasledujúce tri verzie binárneho vyhľadávania s programom (1.17). Ktorý z týchto troch programov je správny? Ktorý z nich je najefektívnejší? Predpokladajte nasledujúce premenné a konštantu $N > 0$:


```
var i, j, k: integer;
    a: array [1 .. N] of T;
    x: T
```

Program A:

```
i := 1; j := N;
repeat
    k := (i + j) div 2;
    if a[k] < x then i := k
                else j := k
until (a[k] = x) ∨ (i ≧ j)
```

Program B:

```
i := 1; j := N;
repeat
    k := (i + j) div 2;
    if x ≦ a[k] then j := k − 1;
    if a[k] ≦ x then i := k + 1
until i > j
```

Program C:

```
i := 1; j := N;
repeat
    k := (i + j) div 2;
    if x < a[k] then j := k
                else i := k + 1
until i ≧ j
```

Pomôcka: Všetky programy musia skončiť buď na základe podmienky $a[k] = x$, t. j. v prípade, že taký prvok existuje, alebo na základe podmienky $a[k] \neq x$, t. j. vtedy, keď sa v poli nenachádza prvok, ktorého hodnota je x.

1.14. Výrobný podnik organizuje prieskum verejnej mienky na zistenie úspešnosti svojich výrobkov. Vyrába gramofónové platne a magnetické pásky s hitmi, z ktorých najúspešnejšie sa odvysielajú v rámci hitparády. Skúmané obyvateľstvo sa rozdelí na štyri kategórie podľa pohlavia a veku. Prvú vekovú kategóriu budú

tvoriť dvadsaťroční a mladší, druhú ostatní. Každá osoba môže uviesť päť hitov, ktoré sú označené číslami 1 až N (pričom $N = 30$). Výsledky prieskumu sú reprezentované súborom.

```
type hit = 1 .. N;
     pohlavie = (muž, žena);
     odpoveď =
       record meno, priezvisko: alfa;
               p: pohlavie;
               vek: integer;
               výber: array [1 .. 5] of hit
       end;
var prieskum: file of odpoveď
```

Každý prvok súboru reprezentuje jednu odpovedajúcu osobu, charakterizovanú menom, priezviskom, pohlavím, vekom a piatimi hitmi zoradenými podľa priority. Tento súbor je potom vstupom do programu, ktorý má vypočítať tieto výsledky:

1. Zoznam hitov zoradených podľa popularity. Pri každom bude uvedené jeho číslo a celkový počet bodov, ktoré získal prieskumom. Hity, ktoré nedostali ani jediný bod, sa uvádzať nebudú.

2. Štyri samostatné zoznamy s menami a priezviskami tých osôb, ktoré uviedli na prvom mieste jeden z troch najpopulárnejších hitov v ich kategórii.

Do záhlavia každého zoznamu zaraďte vhodný text.

Zoznam použitej literatúry

1-1. DAHL, O. J. — DIJKSTRA, E. W. — HOARE, C. A. R.: Structured Programming. New York, Academic Press (1972), s. 155—165.

1-2. HOARE, C. A. R.: Notes on Data Structuring. v Structured Programming. Dahl, Dijkstra a Hoare, s. 83—174.

1-3. JENSEN, K. — WIRTH, N.: PASCAL, User Manual and Report, Lecture Notes in Computer Science, Vol. 18, Berlin, Springer Verlag (1974).

1-4. WIRTH, N.: Program Development by Stepwise Refinement, Comm. ACM, 14, No. 4 (1971), s. 221—227.

1-5. WIRTH, N.: The Programming Language PASCAL. Acta Informatica 1, No. 1 (1971), 35—63.

1-6. WIRTH, N.: On the Composition of Well-Structured Programs, Computing Surveys, 6, No. 4 (1974), s. 247—259.

2 TRIEDENIE

2.1 ÚVOD

Hlavným cieľom tejto kapitoly je poskytnúť rozsiahly počet príkladov ilustrujúcich použitie údajových štruktúr, o ktorých sme hovorili v predchádzajúcej kapitole, a ukázať, ako výber štruktúry pre príslušné údaje významne ovplyvňuje algoritmy riešiace danú úlohu. Aj triedenie je dobrým príkladom na to, aby sme mohli ukázať, že sa takáto úloha môže vykonať podľa viacerých odlišných algoritmov. Každý z nich má určité výhody a nevýhody, ktoré treba zvážiť v súvislosti s konkrétnou aplikáciou.

Pod *triedením* zvyčajne rozumieme proces preusporiadania danej množiny objektov v špecifickom *poradí*. Účelom triedenia je uľahčiť neskoršie vyhľadávanie prvkov triedenej množiny. Triedenie je ako také temer univerzálne vykonávanou základnou činnosťou. Objekty sa triedia v telefónnych zoznamoch, registroch daní z príjmu, obsahoch kníh, knižniciach, slovníkoch, v skladoch a temer všade tam, kde sa majú uchovávané objekty vyhľadávať a vyberať. Aj malé deti učíme, aby si dávali svoje veci „do poriadku", a tak sa stretli s určitým druhom triedenia dávno predtým, než sa niečo dozvedia o aritmetike.

Triedenie je teda dôležitou a základnou činnosťou najmä v spracovaní údajov. Čo sa dá triediť ľahšie než „údaje"! Napriek tomu sa však pri triedení budeme v prvom rade zaujímať o najzákladnejšie metódy, používané pri konštrukcii algoritmov. Nie je veľa metód, ktoré by sa niekde nevyskytovali v súvislosti s algoritmami triedenia. Triedenie využívame najmä na to, aby sme ukázali veľkú rozmanitosť algoritmov, pričom všetky majú ten istý účel, mnohé z nich sú v istom zmysle optimálne a väčšina z nich má výhody pred inými algoritmami. Preto je ideálne na demonštrovanie nevyhnutnosti analýzy výkonnosti algoritmov. Príklad triedenia sa viac-menej hodí na to, aby sme ukázali, ako významne môže stúpnuť výkonnosť vývojom dômyselných algoritmov, keď sú k dispozícii vhodné metódy.

Závislosť výberu algoritmu od štruktúry údajov, ktoré sa majú spracovať (všadeprítomný jav), je v prípade triedenia taká veľká, že metódy triedenia sa obvykle rozdeľujú na dve kategórie, a to na *triedenie polí* a *triedenie* (sekvenčných) *súborov*. Tieto dve triedy sa často nazývajú *vnútorné* a *vonkajšie triedenie*, pretože polia sa uchovávajú vo veľmi rýchlych vnútorných pamätiach počítačov s náhodným prístupom a súbory sú vhodne umiestnené na pomalších, ale priestrannejších vonkajších pamätiach založených na mechanicky sa pohybujúcich zariadeniach (disky a pásky). O tom, aké je dôležité toto rozlíšenie, svedčí príklad triedenia očíslovaných kariet. Štruktúrovanie kariet ako poľa zodpovedá ich rozloženiu pred triedičom tak, aby každá karta bola viditeľná a individuálne prístupná (*obr. 2.1*).

Obr. 2.1. Triedenie polí

Štruktúrovanie kariet ako súboru však znamená, že z každej kôpky je viditeľná iba jedna karta navrchu (*obr. 2.2*). Takéto obmedzenie bude mať zrejme vážne dôsledky pre metódu triedenia, ktorá sa má použiť. Je však nevyhnutné vtedy, ak je počet kariet, ktoré sa majú rozložiť, väčší než stôl, ktorý máme k dispozícii.

Obr. 2.2. Triedenie súborov

Skôr než budeme pokračovať, zavedieme niektoré pojmy a zápis, ktorý budeme v tejto kapitole používať. Máme prvky

$$a_1, a_2, \ldots, a_n$$

Triedenie spočíva v permutovaní týchto prvkov na také poradie

$$a_{k1}, a_{k2}, \ldots, a_{kn}$$

že, ak je daná *funkcia usporiadania f*, platí

$$f(a_{k1}) \leqq f(a_{k2}) \leqq \ldots \leqq f(a_{kn}) \tag{2.1}$$

Zvyčajne sa funkcia usporiadania nevyhodnocuje podľa špecifikovaného pravidla výpočtu, ale je uložená ako explicitná zložka (časť) každého prvku. Jej hodnota sa nazýva *kľúč* prvku. V dôsledku toho sa štruktúra záznamu mimoriadne dobre hodí na zobrazenie prvkov a_i. Preto definujeme typ prvok, ktorý budeme používať vo všetkých nasledujúcich triediacich algoritmoch:

```
type prvok = record kľúč: integer;
                {deklarácia iných zložiek}        (2.2)
         end
```


Iné zložky predstavujú konkrétne údaje o prvkoch: kľúč iba predpokladá úmysel identifikovať prvky. Pokiaľ však ide o naše triediace algoritmy, kľúč je *jedinou* podstatnou zložkou a nie je potrebné definovať nijaké zvyšné zložky. Výber celého čísla ako typu kľúča môže byť ľubovoľný. Je zrejmé, že tak isto by sa mohol použiť akýkoľvek typ, pre ktorý je definovaná relácia úplného usporiadania.

Metóda triedenia je *stabilná*, ak relatívne poradie prvkov s rovnakými kľúčmi zostáva v procese triedenia nezmenené. Stabilita triedenia je často žiadúca vtedy, ak sú prvky už usporiadané (utriedené) podľa určitých sekundárnych kľúčov, t. j. vlastností, ktoré samotný (primárny) kľúč neodráža.

Táto kapitola nie je vyčerpávajúcim prehľadom metód triedenia. Sú tu skôr podrobnejšie vysvetlené niektoré vybrané špecifické metódy. Čitateľom, ktorých zaujíma dôkladný výklad triedenia, odporúčame výbornú a obsažnú príručku D. E. KNUTHA [2-7] (pozri aj LORIN [2-10]).

2.2 TRIEDENIE POLÍ

Od metód triedenia poľa by sme mali predovšetkým požadovať úsporné využívanie pamäti, ktorú máme k dispozícii. To znamená, že permutácia, ktorou sa prvky usporadúvajú, sa má vykonávať na mieste[1] a metódy, ktoré prenášajú prvky z poľa *a* do výsledného poľa *b*, sú svojou povahou menej zaujímavé. Keď sme takto vymedzili náš výber metód spomedzi mnohých možných riešení kritériom úspory pamäti, prejdeme k prvej klasifikácii podľa efektivnosti metód, t. j. podľa ich časovej náročnosti. Dobré *meradlo efektivnosti* dostaneme spočítaním počtov *C* potrebných *porovnaní kľúčov* a *M presunov prvkov*. Tieto počty sú funkciami počtu *n* prvkov, ktoré sa majú triediť. Pretože dobré triediace algoritmy vyžadujú rádovo $n \cdot \log n$ porovnaní, rozoberieme najprv niekoľko jednoduchých a zrejmých triediacich metód nazývaných *priame metódy*, z ktorých všetky vyžadujú rádovo n^2 porovnaní

[1] Pojem „na mieste" najvýstižnejšie charakterizuje spôsoby triedenia bez pomocnej pamäti. Je odvodený z originálneho pojmu „in situ" — pozn. prekl.

kľúčov. Máme tri pádne dôvody na to, aby sme uviedli priame metódy skôr, než prejdeme k rýchlejším algoritmom.

1. Priame metódy sú osobitne vhodné na objasnenie charakteristických čŕt hlavných zásad triedenia.
2. Ich programy sú ľahko pochopiteľné a krátke. Nezabúdajte, že aj programy zaberajú miesto v pamäti!
3. Hoci dômyselné metódy vyžadujú menej operácií, sú tieto operácie zvyčajne zložitejšie v podrobnostiach; teda priame metódy sú rýchlejšie pre dostatočne malé n, nepoužívajú sa však pre veľké n.

Metódy triedenia, ktoré triedia prvky na mieste, možno rozdeliť na tri základné kategórie podľa toho, z ktorej metódy vychádzajú:

1. Triedenie vkladaním.
2. Triedenie výberom.
3. Triedenie výmenou.

V ďalšom sa budeme podrobnejšie zaoberať jednotlivými metódami triedenia prvkov na mieste (t. j. bez pomocnej pamäti), pričom objasníme a porovnáme uvedené tri metódy triedenia.

Predpokladajme, že triediaci program má utriediť pole prvkov, reprezentované premennou a, na mieste. Prvky poľa sú typu prvok, definovaného v (2.2). Okrem toho majme definovaný typ index:

```
type index = 0..n;                                    (2.3)
var a: array 1..n of prvok
```

2.2.1 TRIEDENIE PRIAMYM VKLADANÍM

Túto metódu požívajú hráči kariet. Prvky (v tomto prípade hracie karty) sa rozdelia na dve skupiny — postupnosti: zdrojovú a cieľovú. Z prvej — zdrojovej — sa prvky vyberajú, do druhej — cieľovej — sa ukladajú. Zdrojovú postupnosť tvoria prvky $a_i, \ldots, a_n$, cieľovú prvky $a_1, \ldots, a_{i-1}$. V rámci každého kroku algoritmu sa zo zdrojovej postupnosti vyberie i-tý prvok a vloží sa na patričné miesto cieľovej postupnosti. Algoritmus sa začína hodnotou indexu $i = 2$ a v rámci každého kroku sa jeho hodnota zväčší o jednotku. Príklad triedenia znázorňuje *tab. 2.1*. V prvom riadku sa nachádza zdrojová neutriedená postupnosť, v poslednom riadku je utriedená cieľová postupnosť.

V príklade procesu triedenia vkladaním uvedenom v *tab. 2.1* sme utriedili osem náhodne vybratých čísel. Algoritmus triedenia priamym vkladaním má takýto tvar:

```
for i := 2 to n do
    begin x := a[i];
        „vlož x na patričné miesto
        v postupnosti a1 ... ai"
    end
```

Príklad procesu triedenia priamym vkladaním *Tabuľka 2.1*

Začiatočné hodnoty triediacich kľúčov	44	55	12	42	94	18	06	67
$i = 2$	44	55	12	42	94	18	06	67
$i = 3$	12	44	55	42	94	18	06	67
$i = 4$	12	42	44	55	94	18	06	67
$i = 5$	12	42	44	55	94	18	06	67
$i = 6$	12	18	42	44	55	94	06	67
$i = 7$	06	12	18	42	44	55	94	67
$i = 8$	06	12	18	42	44	55	67	94

Pri hľadaní vhodného miesta v cieľovej postupnosti je účelné striedať porovnania prvkov s ich presunmi. Prvok x sa dostane na svoje miesto v cieľovej postupnosti tak, že ho porovnáme s jeho ľavým susedom a. Ak je väčší, tak tohto suseda posunieme o jedno miesto vpravo. Proces hľadania vhodného miesta pre prvok x končí vtedy, ak:

1. Kľúč prvku a_j je menší ako kľúč prvku x.
2. Dosiahneme ľavý koniec cieľovej postupnosti.

Vzhľadom na to, že ide o typický prípad cyklu s dvoma ukončovacími podmienkami, je výhodné použiť techniku zarážky. Dá sa veľmi jedno-

ducho aplikovať tým, že prvku a_0 priradíme hodnotu prvku x. (Samozrejme, že sa to dá urobiť jedine v tom prípade, ak interval indexovej premennej v deklarácii poľa a zväčšíme na $0..n$.) Úplný algoritmus triedenia priamym vkladaním uvádza program 2.1.

PROGRAM 2.1. *Triedenie priamym vkladaním*

```
procedure triedenievkladanim;
  var i, j: index; x: prvok;
begin
  for i: = 2 to n do
  begin x: = a[i]; a[0]: = x; j: = i − 1;
    while x.kľúč < a[j].kľúč do
      begin a[j + 1]: = a[j]; j: = j − 1
      end;
    a[j + 1]: = x
  end
end
```

Analýza triedenia priamym vkladaním

Počet porovnaní kľúčov C_i v i-tom prechode je najviac $i - 1$ a najmenej 1. Za predpokladu, že všetky permutácie n kľúčov sú rovnako pravdepodobné, môžeme C_i v priemere pokladať za rovnajúce sa $i/2$. Počet presunov M_i (priradením jednotlivým prvkom) je $C_i + 2$ vrátane zarážky. Celkový počet porovnaní a presunov potom bude

$$\begin{aligned} C_{min} &= n - 1 & M_{min} &= 2(n-1) \\ C_{priem} &= \frac{1}{4}(n^2 + n - 2) & M_{priem} &= \frac{1}{4}(n^2 + 9n - 10) \\ C_{max} &= \frac{1}{2}(n^2 + n) - 1 & M_{max} &= \frac{1}{2}(n^2 + 3n - 4) \end{aligned} \tag{2.4}$$

Najmenšie hodnoty C a M sú v prípade, ak zdrojová postupnosť obsahuje usporiadané prvky. Tento prípad nazývame najlepší. Opakom je najhorší prípad, ktorý nastane vtedy, keď sú prvky zdrojovej postupnosti v obrátenom poradí (opačne usporiadané). V tomto zmysle triedenie vkladaním dokumentuje skutočne reálnu situáciu. Okrem

toho je jasné, že uvedený algoritmus opisuje stabilnú metódu triedenia, t. j. takú, pri ktorej sa zachováva aj relatívne usporiadanie rovnakých prvkov, aké bolo vo vstupnej postupnosti.

Keď si uvedomíme, že cieľová postupnosť $a_1 \dots a_{i-1}$, t. j. tá, do ktorej sa vybratý prvok ukladá, je utriedená, tak je možné algoritmus triedenia vkladaním veľmi jednoducho vylepšiť. Zlepšenie sa týka rýchlejšej metódy určenia miesta v cieľovej postupnosti, do ktorého sa má prvok uložiť. Obyčajne sa volí metóda binárneho vyhľadávania, ktorá hľadané miesto nájde postupným delením postupnosti na dve podpostupnosti. Modifikovaný algoritmus triedenia sa nazýva binárne vkladanie a je uvedený v programe 2.2.

PROGRAM 2.2. *Triedenie binárnym vkladaním*

```
procedure binárnevkladanie;
  var i, j, l, r, m: index; x: prvok;
begin
  for i: = 2 to n do
  begin x: = a[i]; l: = 1; r: = i − 1;
    while l ≦ r do
    begin m: = (l + r) div 2;
      if x.kľúč < a[m].kľúč then r: = m − 1
                             else l: = m + 1
    end;
    for j: = i − 1 downto 1 do a[j + 1]: = a[j];
    a[l]: = x
  end
end
```

Analýza binárneho vkladania

Miesto pre uloženie prvku sa nájde vtedy, keď platí:

$$a_j.kľúč \leqq x.kľúč < a_{j+1}.kľúč$$

Teda skúmaný interval sa nakoniec rovná 1; čiže interval pozostávajúci z i kľúčov sa rozpoľuje $\lceil \log_2 i \rceil$ -krát. Počet porovnaní potom bude

$$C = \sum_{i=1}^{n} \lceil \log_2 i \rceil$$

Aproximáciou tejto sumy pomocou integrálu dostávame:

$$\int_1^n \log x \, dx = x(\log x - c)\Big|_1^n = n(\log n - c) + c \qquad (2.5)$$

pričom $c = \log e = 1/\ln 2 = 1.44269\ldots$. Počet porovnaní je v podstate nezávislý od začiatočného usporiadania prvkov. Ale vzhľadom na skracovací charakter delenia zahrnutého v rozpoľovaní skúmaného intervalu môže byť skutočný počet porovnaní potrebných pri i kľúčoch až o 1 vyšší, ako sa očakávalo. Podstata tejto systematickej odchýlky spočíva v tom, že miesto na uloženie prvku sa v priemere rýchlejšie zisti na ľavom konci postupnosti ako na pravom konci. Preto sa uprednost-ňujú tie prípady, kedy sú začiatočné postupnosti značne neusporiada-né. Najmenší počet porovnaní je potrebný vtedy, keď sú prvky v zdro-jovej postupnosti opačne usporiadané, najviac porovnaní treba pri usporiadanej zdrojovej postupnosti. Tieto črty charakterizujú nepriro-dzené správanie algoritmu triedenia.

$$C \doteq n(\log n - \log e \pm 0.5)$$

Žiaľ, vylepšenie algoritmu triedenia binárnym vyhľadávaním sa týka iba počtu porovnaní, a nie počtu potrebných presunov. Presuny prv-kov, t. j. kľúčov a s nimi združených informácií, vyžadujú vo všeobec-nosti oveľa viac času ako porovnanie dvoch kľúčov. Uvedeným vylep-šením algoritmu sa výrazne nevylepší hodnota ukazovateľa M: tento naďalej zostáva rádu n^2. V skutočnosti triedenie už utriedeného poľa zaberá viac času ako metóda priameho vkladania prostredníctvom sekvenčného vyhľadávania! Tento príklad ukazuje, že často môže dôjsť k situácii, keď prirodzené vylepšenie algoritmu má nakoniec menší efekt, ako sa pôvodne očakávalo, a v niektorých prípadoch (ktoré skutočne existujú) vylepšenie môže byť zhoršením. Napokon triedenie vkladaním nepatrí medzi najvýhodnejšie metódy na číslicových počíta-čoch; vloženie prvku s nasledujúcim posunom všetkých prvkov o jednu pozíciu je neekonomické. Lepšie výsledky by sme očakávali od takej metódy, pri ktorej sa jednotlivé prvky iba presúvajú na dlhšie vzdiale-nosti. Táto úvaha nás vedie k druhej metóde triedenia, ktorá sa nazýva triedenie metódou výberu.

2.2.2 TRIEDENIE PRIAMYM VÝBEROM

Táto metóda je založená na nasledujúcom princípe:

1. Vyber prvok s najmenším kľúčom.
2. Vymeň ho s prvým prvkom a_1.
3. Potom tieto operácie opakuj s ostávajúcimi $n - 1$ prvkami, s $n - 2$ prvkami atď., až zostane jediný najväčší prvok. Princíp znázor-níme na tých istých ôsmich kľúčoch, ktoré sme použili v *tab. 2.2.*

Príklad procesu triedenia priamym výberom *Tabuľka 2.2*

Začiatočné hodnoty triediacich kľúčov:								
	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67

Program môžeme napísať takto:

```
for i := 1 to n - 1 do
    begin „priraď index najmenšieho prvku postupnosti
          a[i] ... a[n] do k;“
          „vymeň prvky a_i a a_k“
    end
```

Táto metóda sa nazýva priamy výber a je v istom zmysle opakom metódy priameho vkladania. Pri priamom vkladaní sa v rámci každého kroku uvažuje jedna položka zdrojovej postupnosti a všetky prvky cieľovej postupnosti na vyhľadanie miesta vloženia. Pri priamom výbe-re sa v každom kroku berie do úvahy celá zdrojová postupnosť (t. j. jej všetky prvky), z ktorej sa vyberie prvok s najmenším kľúčom a uloží sa ako ďalší prvok do cieľovej postupnosti. Úplný program triedenia priamym výberom dokumentuje program 2.3.

PROGRAM 2.3. *Triedenie priamym výberom*

```
procedure triedenievýberom;
  var i, j, k: index; x: prvok;
begin for i: = 1 to n − 1 do
    begin k: = i; x: = a[i];
      for j: = i + 1 to n do
        if a[j].kľúč < x.kľúč then
        begin k: = j; x: = a[j]
        end
      a[k]: = a[i]; a[i]: = x
    end
end
```

Analýza triedenia priamym výberom

Je zrejmé, že počet porovnaní kľúčov C nezávisí od začiatočného usporiadania kľúčov. V tomto zmysle možno túto metódu považovať za menej prirodzenú ako triedenie vkladaním. Počet porovnaní C je

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

Počet presunov M je minimálne

$$M_{min} = 3(n - 1) \tag{2.6}$$

ak sú kľúče v zdrojovej postupnosti usporiadané, a maximálne

$$M_{max} = \text{trunc}\left(\frac{n^2}{4}\right) + 3(n - 1)$$

ak sú kľúče v zdrojovej postupnosti usporiadané opačne. Priemerný počet presunov M_{prem} sa dá ťažko určiť aj napriek jednoduchosti algoritmu. Závisí od toho, koľkokrát sa nájde kľúč k_j menší ako všetky predchádzajúce kľúče $k_1 \dots k_{j-1}$ v postupnosti $k_1 \dots k_n$. Táto hodnota sa preto berie ako priemer všetkých $n!$ permutácií n-kľúčov a je určená vzťahom

$$H_n - 1$$

kde H_n je n-té harmonické číslo

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \tag{2.7}$$

(pozri aj KNUTH, vol 1, s. 95—99).

H_n možno vyjadriť vzťahom

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \dots \tag{2.8}$$

pričom $\gamma = 0.577\,216\dots$ je Eulerova konštanta. Pre dostatočne veľké n môžeme zanedbať zlomkové časti a priemerný počet priradení v i-tom prechode aproximovať ako

$$F_i = \ln i + \gamma + 1$$

Priemerný počet presunov M_{prem} pri triedení priamym výberom je potom daný sumou F_i pre $i = 1$ až n.

$$M_{prem} = \sum_{i=1}^{n} F_i = n(\gamma + 1) + \sum_{i=1}^{n} \ln i$$

Ďalšou aproximáciou sumy diskrétnych výrazov pomocou integrálu

$$\int_1^n \ln x \, dx = x(\ln x - 1)\Big|_1^n = n \ln n - n + 1$$

dostávame približnú hodnotu

$$M_{prem} \doteq n(\ln n + \gamma) \tag{2.9}$$

Záverom môžeme konštatovať, že vo všeobecnosti je triedenie priamym výberom efektívnejšie ako triedenie priamym vkladaním, hoci v prípadoch, keď sú kľúče v zdrojovej postupnosti usporiadané alebo takmer usporiadané, je triedenie priamym vkladaním predsa len o niečo rýchlejšie.

2.2.3 TRIEDENIE PRIAMOU VÝMENOU

Klasifikácia metódy triedenia je len zriedka úplne jasná. Napríklad obidve predchádzajúce metódy môžu byť klasifikované aj ako metódy

triedenia výmenou. V tomto odseku však uvedieme algoritmus triedenia, v rámci ktorého je výmena dvoch prvkov dominantnou charakteristikou. Nasledujúci algoritmus triedenia výmenou je založený na princípe postupného porovnávania a zámeny dvojíc susedných prvkov až dovtedy, kým všetky prvky postupnosti nie sú utriedené.

Podobne ako v predchádzajúcom prípade triedenia i táto metóda spočíva v cyklickom spracúvaní poľa. V rámci každého prechodu sa vyberie najmenší prvok skúmanej podmnožiny, ktorý sa presunie na ľavý koniec poľa. Ak by sme si sledované pole predstavili namiesto v horizontálnej polohe vo vertikálnej polohe a, s tou istou dávkou predstavivosti, triedené prvky ako bubliny vo vodnej nádrži (so špecifickou váhou úmernou ich kľúčom), tak v rámci každého prechodu poľom dôjde k „vybublaniu" vybratého prvku na úroveň zodpovedajúcu jeho váhe (*tab. 2.3*).

Tabuľka 2.3

Príklad bublinového triedenia

Zdrojová postupnosť	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

Táto metóda je známa pod pojmom bublinové triedenie. Najjednoduchší algoritmus bublinového triedenia dokumentuje program 2.4.

PROGRAM 2.4. *Bublinové triedenie*

```
procedure bublinovétriedenie;
  var i, j: index; x: položka;
begin for i := 2 to n do
    begin for j := n downto i do
      if a[j − 1].kľúč > a[j].kľúč then
        begin x := a[j − 1];
          a[j − 1] := a[j];
          a[j] := x
        end
    end
end {bublinovétriedenie}
```


Zrejme, že tento algoritmus sa dá zlepšiť. Keď si všimneme príklad uvedený v *tab. 2.3*, uvidíme, že v posledných troch prechodoch sa prvky nepremiestňovali, pretože už boli utriedené. Ak si v rámci prechodov dokážeme zapamätať, či nastala alebo nenastala nejaká výmena prvkov, je možné realizovať prvé zlepšenie algoritmu, ktoré spočíva v presnom stanovení konca triediaceho procesu. Algoritmus končí v tom okamihu, keď v rámci posledného prechodu nenastala operácia výmeny prvkov. Toto zlepšenie sa dá ďalej vylepšiť jednoducho tým, že okrem registrovania poslednej výmeny si zapamätáme aj polohu (index) poslednej výmeny. Napríklad je jasné, že všetky dvojice susedných prvkov pod takýmto indexom k sú v žiadanom poradí. Nasledujúci prechod môže preto končiť pri tomto indexe namiesto toho, aby pokračoval až po vopred určený dolný index i. Šikovný programátor si iste všimne určitú asymetriu: nesprávne umiestnená bublinka na „ťažkom" konci ináč utriedeného poľa sa dostane do svojej polohy jediným prechodom, kým neumiestnená bublinka na opačnom konci bude putovať k svojej správnej pozícii v každom prechode iba o jeden krok. Napríklad pole

12 18 42 44 55 67 94 06

sa utriedi vylepšeným bublinovým triedením jediným prechodom, kým pole

94 06 12 18 42 44 55 67

bude vyžadovať až sedem prechodov. Táto neprirodzená asymetria nás nabáda k tretiemu vylepšeniu: striedať smer prehľadávania v jednotlivých (po sebe idúcich) prechodoch. Takáto stratégia sa nazýva triedenie pretriasaním (shakersort). Jeho činnosť dokumentuje *tab. 2.4*. Majme opäť tie isté triediace kľúče ako v prípade *tab. 2.3*.

Príklad triedenia pretriasaním *Tabuľka 2.4*

$l = 2$ $r = 8$ ↑	3 8 ↓	3 7 ↑	4 7 ↓	4 4 ↑
44	06	06	06	06
55	44	44	12	12
12	55	12	44	18
42	12	42	18	42
94	42	55	42	44
18	94	18	55	55
06	18	67	67	67
67	67	94	94	94

PROGRAM 2.5. *Shakersort*

```
procedure shakersort;
  var j, k, l, r: index; x: prvok;
begin l: = 2; r: = n; k: = n;
  repeat
    for j: = r downto l do
      if a[j - 1].kľúč > a[j].kľúč then
      begin x: = a[j - 1]; a[j - 1]: = a[j]; a[j]: = x;
            k: = j
      end;
    l: = k + 1;
    for j: = l to r do
      if a[j - 1].kľúč > a[j].kľúč then
      begin x: = a[j - 1]; a[j - 1]: = a[j]; a[j]: = x;
            k: = j
      end;
    r: = k - 1
  until l > r
end {shakersort}
```

Analýza bublinového triedenia a triedenia pretriasaním

Počet porovnaní kľúčov pri triedení priamou výmenou je

$$C = \frac{1}{2}(n^2 - n) \qquad (2.10)$$

a minimálne, priemerné a maximálne počty presunov (priradení medzi prvkami) sú

$$M_{min} = 0, \quad M_{priem} = \frac{3}{4}(n^2 - n), \quad M_{max} = \frac{3}{2}(n^2 - n) \qquad (2.11)$$

Uvedené výsledky zohľadňujú aj spomenuté vylepšenia algoritmu (shakersort). Minimálny počet porovnaní je $C_{min} = n - 1$. Pokiaľ ide o zlepšené bublinové triedenie, Knuth zistil, že priemerný počet prechodov je úmerný $n - k_1\sqrt{n}$ a priemerný počet porovnaní $\frac{1}{2}[n^2 - n(k_2 + \ln n)]$. Poznamenávame však, že všetky uvedené zlepšenia nijako neovplyvňujú počet výmeny, ale len zmenšujú počet nadbytočných dvojnásobných kontrol. Zámena dvoch prvkov je však väčšinou oveľa nákladnejšou operáciou než porovnanie kľúčov; naše vynikajúce zlepšenia majú preto oveľa menší účinok, než by človek intuitívne očakával.

Táto analýza ukazuje, že triedenie výmenou a jeho malé zlepšenia nedosahujú kvalitu triedenia priamym vkladaním ani triedenia priamym výberom a bublinové triedenie môže v skutočnosti sotva čo ponúknuť okrem príťažlivého názvu. Algoritmus triedenia pretriasaním sa používa s výhodou v tých prípadoch, v ktorých je známe, že prvky sú už temer usporiadané. Je to však v praxi zriedkavý prípad.

Môžeme ukázať, že priemerná vzdialenosť, ktorú má každý z n prvkov prejsť počas triedenia, je $n/3$ miest. Toto číslo poskytuje kľúč pri hľadaní lepších, t.j. efektívnejších, metód triedenia. Všetky priame metódy triedenia v podstate posúvajú každý prvok o jedno miesto v každom základnom kroku, preto si vyžadujú rádovo n^2 takých krokov. Každé zlepšenie sa musí zakladať na zásade presunu prvkov na väčšie vzdialenosti v jednotlivých krokoch.

V ďalšom budeme hovoriť o troch zlepšených metódach, a to vždy o jednej pre každú základnú metódu triedenia vkladaním, výberom a výmenou.

2.2.4 TRIEDENIE VKLADANÍM SO ZMENŠOVANÍM KROKU (SHELLOVO TRIEDENIE)

Zlepšenie triedenia priamym vkladaním navrhol D. L. Shell v r. 1959. Metóda je vysvetlená a demonštrovaná na našom štandardnom príklade ôsmich prvkov (*tab. 2.5*). Najskôr sa všetky prvky, ktoré sú od seba vzdialené o 4 miesta, zoskupia a triedia jednotlivo. Tento proces sa nazýva triedenie s krokom 4. V príklade s ôsmimi prvkami obsahuje každá skupina presne dva prvky. Po tomto prvom prechode sa prvky opäť zoradia do skupín s prvkami, vzdialenými od seba o 2 miesta, a potom sa triedia znova. Tento proces sa nazýva triedenie s krokom 2. Napokon pri treťom prechode sa všetky prvky triedia obyčajným triedením alebo triedením s krokom 1.

Triedenie vkladaním so zmenšovaním kroku *Tabuľka 2.5*

	44	55	12	42	94	18	06	67
Výsledok triedenia s krokom 4:	44	18	06	42	94	55	12	67
Výsledok triedenia s krokom 2:	06	18	12	42	44	55	94	67
Výsledok triedenia s krokom 1:	06	12	18	42	44	55	67	94

Mohli by sme spočiatku pochybovať o nevyhnutnosti niekoľkých triediacich prechodov, z ktorých každý zahŕňa všetky prvky, pretože vynaložená práca by mohla byť väčšia ako celkový úžitok. Každý triediaci krok však buď zahŕňa relatívne málo prvkov, alebo sú už prvky celkom dobre usporiadané a požaduje sa len pomerne málo nových zmien usporiadania.

Zrejme, že výsledkom metódy je usporiadané pole a každý prechod získa „profit" z predchádzajúcich prechodov (vzhľadom na to, že každé

triedenie s krokom i kombinuje dve skupiny triedené v predchádzajúcom triedení s krokom $2i$). Je tiež zrejmé, že akákoľvek postupnosť krokov bude prijateľná, pokiaľ je však posledná jednotková, pretože v najhoršom prípade posledný prechod urobí všetku triediacu prácu.

Menej samozrejmé však je, že metóda so zmenšovaním kroku prináša lepšie výsledky s inými krokmi, než sú mocniny dvojky. Triediaci program preto nekladie žiadne špecifické požiadavky na postupnosť krokov. Nech t je počet krokov, ktoré označíme

$$h_1, h_2, \ldots, h_t$$

a nech platí

$$h_t = 1, \quad h_{i+1} < h_i \tag{2.12}$$

Potom sa každé triedenie s krokom h programuje ako priame vkladanie s využitím techniky zarážky na stanovenie podmienky ukončenia procesu vyhľadávania pozície pre vloženie prvku. Je samozrejmé, že každý krok triedenia potrebuje svoju vlastnú zarážku. Z hľadiska efektívnosti programu je dôležité, aby algoritmus, ktorý určuje tieto zarážky, bol čo najjednoduchší. Ak teda chceme pole a utriediť uvedenou metódou, treba zväčšiť toto pole nielen o prvok $a[0]$, ale aj o ďalších h_1 prvkov. Jeho deklarácia bude potom vyzerať takto:

$$a\text{: } \mathbf{array}\ [-h_1 \,..\, n]\ \mathbf{of}\ \text{prvok}$$

Algoritmus triedenia vkladaním so zmenšovaním kroku (nazývame ho podľa jeho autora shellsort [2-11]) pre $t = 4$ je znázornený v programe 2.6.

Program 2.6. *Shellov triediacj algoritmus*

```
procedure shellsort;
  const t = 4;
  var i, j, k, s: index; x: prvok; m: 1..t;
      h: array [1..t] of integer;
begin h[1]: = 9; h[2]: = 5; h[3]: = 3; h[4]: = 1;
  for m: = 1 to t do
  begin k: = h[m]; s: = -k; {pozicia zarážky}
    for i: = k + 1 to n do
```

```
        begin x: = a[i]; j: = i - k;
          if s = 0 then s: = -k; s: = s + 1; a[s]: = x;
          while x.kľúč < a[j].kľúč do
          begin a[j + k]: = a[j]; j: = j - k
          end
          a[j + k]: = x
        end
    end
end
```

Analýza triedenia vkladaním so zmenšovaním kroku

Analýza tohto algoritmu vedie k zložitým matematickým problémom, z ktorých mnohé neboli dodnes vyriešené. Napríklad nie je známe, aká voľba krokov prináša najlepšie výsledky. Zaujímavé je však zistenie, že by nemali byť navzájom násobkami. Tým sa vyhneme úkazu, zjavnému z predchádzajúceho príkladu, v ktorom sa v rámci každého prechodu triedenia vzájomne kombinovali také podpostupnosti prvkov, ktoré predtým nijako spolu nesúviseli. Dôležité je, aby k interakcii medzi rôznymi podpostupnosťami dochádzalo, pokiaľ možno, čo najčastejšie a aby platila táto veta:

Ak triedime s krokom i postupnosť, ktorá bola utriedená s krokom k, tak táto ostáva utriedená s krokom k. KNUTH [2-8] odporúča takúto postupnosť krokov (písanú od konca):

$$1, 4, 13, 40, 121, \ldots$$

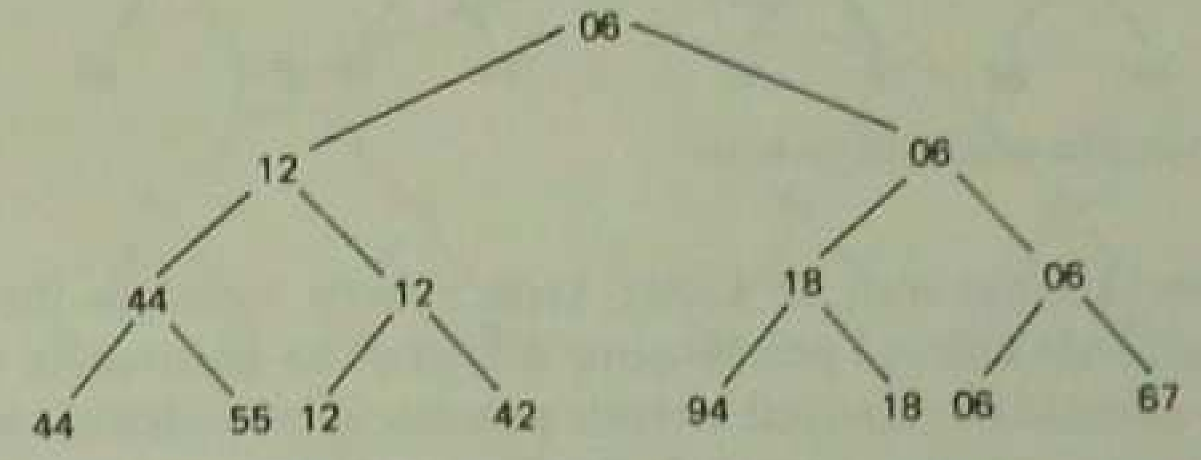
pričom $h_{k-1} = 3h_k + 1$, $h_t = 1$ a $t = \lfloor \log_3 n \rfloor - 1$. Odporúča aj postupnosť

$$1, 3, 7, 15, 31, \ldots$$

kde $h_{k-1} = 2h_k + 1$, $h_t = 1$ a $t = \lfloor \log_2 n \rfloor - 1$. Matematickou analýzou druhej alternatívy Shellovho algoritmu triedenia n-prvkovej postupnosti sa zistilo, že jeho zložitosť je úmerná $n^{1,2}$. Aj keď tento výsledok je podstatne lepší ako n^2, nebudeme sa touto metódou ďalej zaoberať, pretože poznáme ešte lepšie algoritmy triedenia.

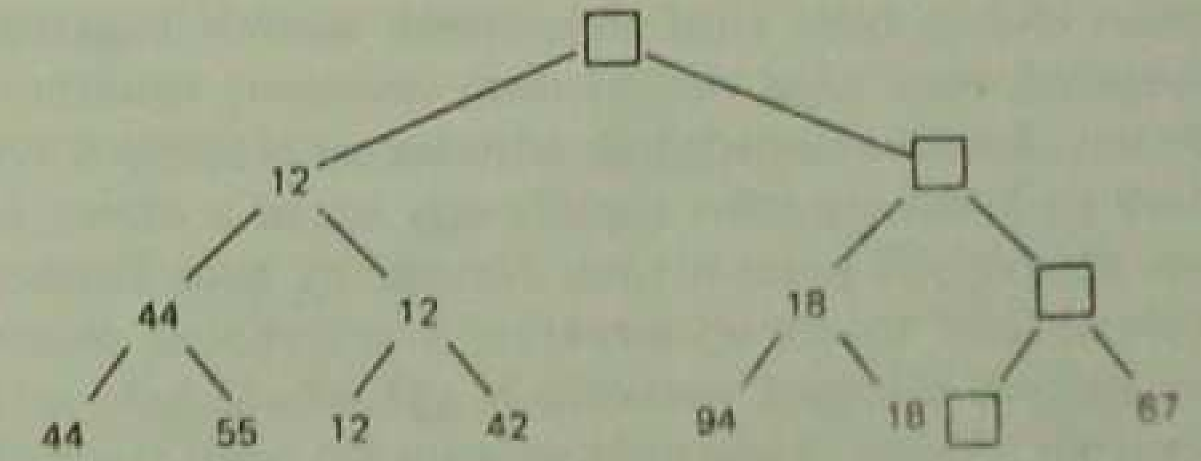
2.2.5 STROMOVÉ TRIEDENIE

Triedenie metódou výberu je založené na opakujúcom sa výbere najmenšieho kľúča spomedzi n prvkov, potom spomedzi $n - 1$ prvkov atď. Preto vyhľadanie najmenšieho kľúča spomedzi n prvkov vyžaduje $n - 1$ porovnaní, spomedzi $n - 1$ prvkov $n - 2$ porovnaní atď. Čo potrebujeme na to, aby sme mohli zlepšiť toto triedenie výberom? Predovšetkým väčšie množstvo informácií získaných z prehľadávania v rámci každého prechodu, než je iba určenie najmenšieho prvku. Napríklad pomocou $n/2$ porovnaní sa dá určiť menší kľúč každej dvojice prvkov, ďalšími $n/4$ porovnaniami menší kľúč každej dvojice menších kľúčov atď. Nakoniec pomocou $n - 1$ porovnaní sa dá zostrojiť strom výberu znázornený na *obr. 2.3*, ktorého koreň obsahuje najmenší prvok [2-2].



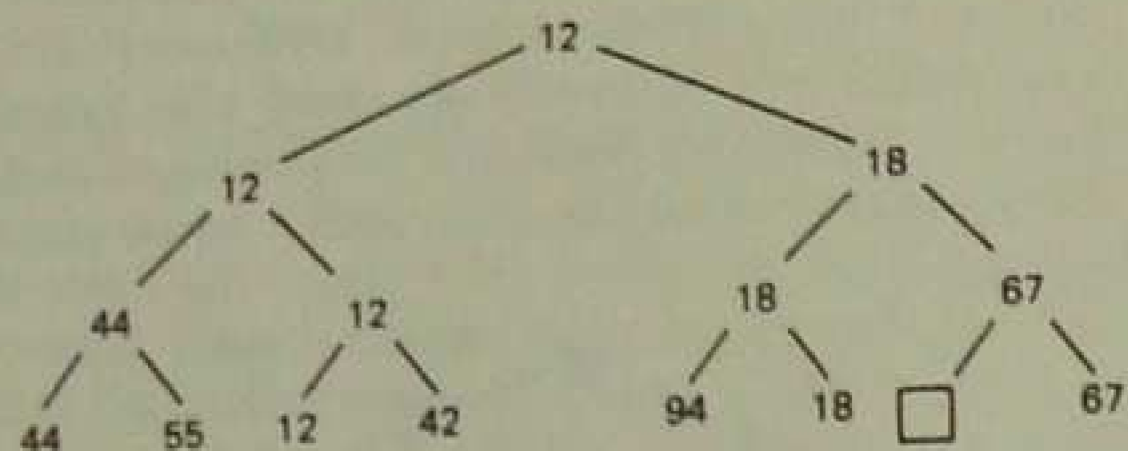
Obr. 2.3. Opakovaný výber medzi dvoma kľúčmi

V ďalšom kroku odstránime najmenší prvok z koreňa stromu a súčasne tento prvok postupne nahrádzame v strome buď prázdnym vrcholom (kľúčom s hodnotou ∞) v prípade listu, alebo zodpovedajúcim



Obr. 2.4. Výber menšieho kľúča

druhým (t. j. väčším) prvkom z každej porovnávanej dvojice, v ktorej sa tento najmenší prvok vyskytoval. Postupujeme pritom od koreňa stromu k jeho listom (*obr. 2.4* a *obr. 2.5*). Zopakovaním uvedeného postupu získame v koreni stromu druhý najmenší prvok, ktorý obdobne vylúčime. Tento proces sa potom opakuje, pokiaľ strom nie je prázdny (t. j. obsahuje samé prázdne vrcholy). Triediaci proces teda končí v okamihu, keď sa koreňu priradí hodnota $-\infty$.



Obr. 2.5. Zaplnenie prázdnych vrcholov

Môžeme konštatovať, že každý krok výberu vyžaduje iba $\log_2 n$ porovnaní. Teda celkove potrebujeme n krokov na zostrojenie stromu a $n \cdot \log n$ základných operácií v rámci procesu výberu. Tento výsledok je výrazným zlepšením oproti doterajším metódam vyžadujúcim n^2 krokov, dokonca aj vzhľadom na Shellov algoritmus, ktorý potreboval $n^{1.2}$ krokov.

Ďalšie zdokonalenie stromového triedenia vyžaduje zlepšenie techniky manipulovania s informáciami, ktoré získavame z predchádzajúcich porovnaní. Tým sa však zväčší zložitosť jednotlivých krokov triedenia. Našou ďalšou úlohou bude nájsť efektívnejšie metódy organizovania týchto informácií, resp. nájsť vhodný druh stromovej štruktúry.

Prvé, čo nás okamžite napadne, je odstránenie prázdnych vrcholov ($-\infty$), ktoré na konci triedenia zaplnia celý strom a okrem toho sú predmetom zbytočných porovnávaní. Navyše by bolo žiadúce nájsť spôsob reprezentácie stromovej štruktúry s n vrcholmi prostredníctvom n pamäťových jednotiek namiesto $2n - 1$, ako to bolo pri uvedenom stromovom triedení. Tieto ciele predstavujú základné myšlienky triediacej metódy, ktorú vynašiel J. WILLIAMS [2-14] a nazval ju triedenie haldou. Je zrejmé, že táto metóda znamená výrazné zlepšenie stromového triedenia.

Halda je definovaná ako postupnosť kľúčov

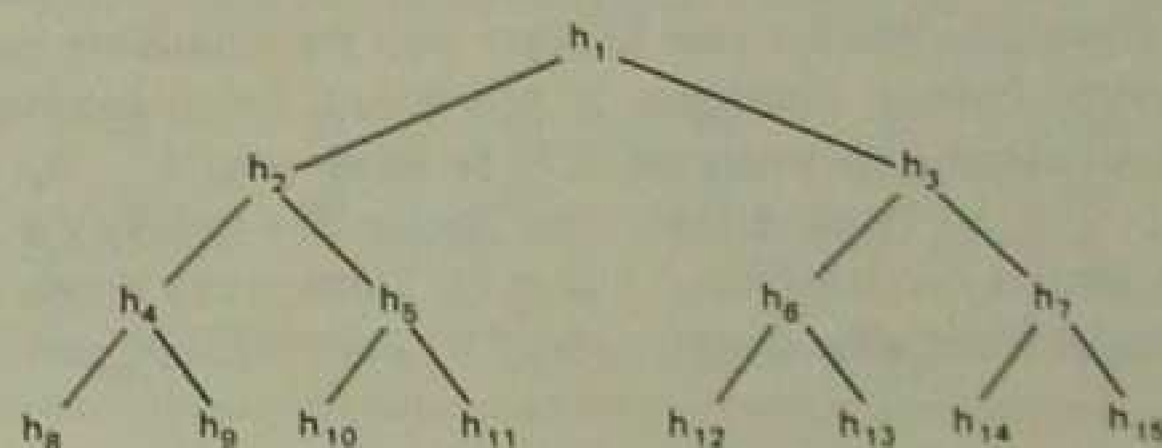
$$h_l, h_{l+1}, \ldots, h_r$$

takých, že platí

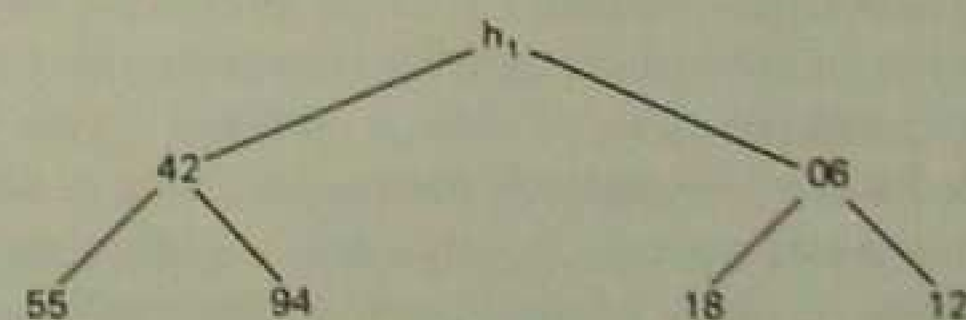
$$h_i \leqq h_{2i} \qquad h_i \leqq h_{2i+1} \qquad (2.13)$$

pre všetky $i = l \ldots r/2$. Ak je binárny strom reprezentovaný poľom znázorneným na *obr. 2.6*, tak triediace stromy na *obr. 2.7* a *2.8* sú haldy a platí: prvok h_1 je najmenší prvok haldy

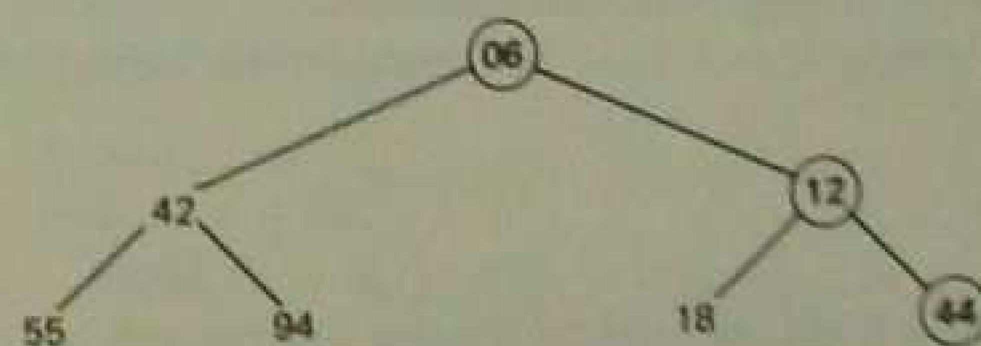
$$h_1 = \min(h_1 \ldots h_n)$$



Obr. 2.6. Pole h zobrazené ako binárny strom



Obr. 2.7. Halda so siedmimi prvkami



Obr. 2.8. Halda rozšírená o prvok s kľúčom 44

Predpokladajme, že máme danú haldu s prvkami $h_{l+1} \dots h_r$ (pre nejaké l a r) a že máme do nej pridať nový prvok x tak, aby sme dostali rozšírenú haldu $h_l \dots h_r$. Ako príklad zoberme haldu $h_1 \dots h_7$ znázornenú na *obr. 2.7* a rozšírime ju doľava o prvok $h_1 = 44$. Novú haldu zostrojíme takto: Najprv priradíme prvok x koreňu stromu, a potom ho postupne porovnávame a vymieňame vždy s menším prvkom. V uvedenom príklade je prvok s hodnotou kľúča 44 najprv vymenený s kľúčom 06, potom s kľúčom 12, až vznikne halda znázornená na *obr. 2.8*.

Teraz sformulujme algoritmus konštrukcie haldy. Predpokladajme, že i a j sú indexy označujúce prvky, ktoré treba v rámci každého prechodu vymeniť. Upozorňujeme čitateľa, aby sa sám presvedčil, či predkladaný algoritmus zachováva podmienky (2.13) definujúce haldu. Autorom algoritmu konštrukcie haldy v poli v minimálnej pamäti je R. W. Floyd. Jadrom algoritmu je procedúra, ktorú nazývame vytvorhaldu, uvedená ako program 2.7. Je dané pole $h_1 \dots h_n$. Pritom prvky $h_{n/2+1} \dots h_n$ už tvoria haldu, lebo žiadne dva indexy i a j nie sú také, že by platilo $j = 2i$ (alebo $j = 2i + 1$). Tieto prvky tvoria spodný rad príslušného binárneho stromu (*obr. 2.6*) a nevyžadujú žiadne usporiadanie.

PROGRAM 2.7. *Vytvorenie haldy*

```
procedure vytvorhaldu (l, r: index);
  label 13;
  var i, j: index; x: prvok;
begin i: = l; j: = 2 * i; x: = a[i];
  while j ≦ r do
  begin if j < r then
          if a[j].kľúč > a[j + 1].kľúč then j: = j + 1;
        if x.kľúč ≦ a[j].kľúč then goto 13;
        a[i]: = a[j]; i: = j; j: = 2 * i {vytvorenie haldy}
  end;
13: a[i]: = x
end
```

Halda sa teraz každým krokom rozšíri doľava o nový prvok, ktorý sa prostredníctvom procedúry vytvorhaldu dostane na správnu pozíciu. Tento proces je zachytený v *tab. 2.6*. Algoritmicky môžeme opísať proces generovania n-prvkovej haldy $h_1 \dots h_n$ v minimálnej pamäti takto:

```
l: = n(div 2) + 1;
while l > 1 do
  begin l: = l − 1; vytvorhaldu (l, n)
  end
```

Ak chceme získať prvky v usporiadanom tvare, musíme vykonať n krokov vytvorenia haldy, pričom po každom kroku môžeme vybrať z koreňa haldy nasledujúci prvok.

Konštrukcia haldy *Tabuľka 2.6*

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
44	55	06	42	94	18	12	67
44	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

Poslednými otázkami, na ktoré musíme nájsť vhodné odpovede, sú: kde umiestniť jednotlivé prvky z koreňa haldy a či je možné triedenie na mieste, t. j. bez pomocnej pamäti. Pochopiteľne, že existuje prijateľné riešenie. V rámci každého kroku vyberieme posledný prvok z haldy (označme ho x) a na jeho miesto uložíme prvok z koreňa haldy. Potom použijeme procedúru vytvorhaldu, pomocou ktorej sa dostane prvok x na svoju správnu pozíciu. Potrebujeme na to $n - 1$ krokov, ktoré sú zachytené prostredníctvom haldy v *tab. 2.7*.

Formálne môžeme uvedený proces opísať pomocou procedúry vytvorhaldu (program 2.7) takto:

```
r: = n;
while r > 1 do
  begin x: = a[1]; a[1]: = a[r]; a[r]: = x;
        r: = r − 1; Vytvorhaldu (1, r)
  end
```

Tabuľka 2.7

Príklad triedenia haldou

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44	06
18	42	44	55	94	67	12	06
42	55	44	67	94	18	12	06
44	55	94	67	42	18	12	06
55	67	94	44	42	18	12	06
67	94	55	44	42	18	12	06
94	67	55	44	42	18	12	06

Z príkladu v *tab. 2.7* vidno, že výsledná postupnosť je v skutočnosti usporiadaná obrátene. Dá sa to ľahko odstrániť tým, že v procedúre vytvorhaldu obrátime reláciu usporiadania. Výsledkom je kompletný algoritmus triedenia haldou, uvedený v programe 2.8.

PROGRAM 2.8. *Algoritmus triedenia haldou*

```
procedure triedeniehaldou;
  var l, r: index; x: prvok;
    procedure vytvorhaldu;
      label 13;
      var i, j: index;
    begin i: = l; j: = 2 * i; x: = a[i];
      while j ≦ r do
      begin if j < r then
                 if a[j].kľúč < a[j + 1].kľúč
                 then j: = j + 1;
                 if x.kľúč ≧ a[j].kľúč then goto 13;
                 a[i]: = a[j]; i: = j; j: = 2 * i
      end;
    13: a[i]: = x
    end;
begin l: = (n div 2) + 1; r: = n;
  while l > 1 do
    begin l = l − 1; vytvorhaldu
    end;
  while r > 1 do
    begin x: = a[1]; a[1]: = a[r]; a[r]: = x;
          r: = r − 1; vytvorhaldu
    end
end {triedeniehaldou}
```


Analýza triedenia haldou

Na prvý pohľad iste nie je jasné, či táto metóda prináša dobré výsledky. Veď veľké prvky sa najprv dostanú na ľavú stranu haldy a až potom sa začnú presúvať na konečnú, vzdialenejšiu stranu na pravom konci haldy. Preto sa táto metóda neodporúča pre malý počet triedených prvkov, ako to bolo v našom príklade. Naproti tomu pre veľké n je triedenie haldou veľmi efektívna metóda a čím väčšie je n, tým lepšie výsledky dosiahneme, dokonca aj v porovnaní so Shellovým triedením.

V najhoršom prípade potrebujeme $n/2$ prechodov, t. j. vyvolaní procedúry vytvorhaldu, ktorou sa prvky premiestnia cez $\log(n/2)$, $\log(n/2 - 1)$, ..., $\log(n - 1)$ pozícií, pričom logaritmus má základ 2 a zaokrúhľujeme na najbližšie menšie celé číslo. Triediaca fáza potrebuje $n - 1$ prechodov a vyžaduje najviac $\log(n - 1)$, $\log(n - 2)$, ..., 1 presunov. Nakoniec ešte treba pripočítať $n - 1$ presunov, ktorými sa dostane prvok z koreňa na pravú stranu haldy. Teda celková cena algoritmu triedenia haldou bude v najhoršom prípade $n \cdot \log(n)$. Tento výborný výsledok pre najhorší prípad je jednou z najlepších vlastností triedenia haldou.

Nie je vôbec jasné, v ktorých prípadoch možno očakávať najhoršie a v ktorých najlepšie výsledky. Všeobecne sa zdá, že triedeniu haldou vyhovujú zdrojové postupnosti, ktoré sú viac-menej usporiadané v opačnom poradí. Vieme, že toto je vlastnosť neprirodzeného správania sa. Ak sú prvky zdrojovej postupnosti opačne usporiadané, je zjavné, že fáza vytvorenia haldy nevyžaduje žiadne presuny. Záverom ešte ukážeme dve začiatočné postupnosti, ktoré majú za následok

minimálny a maximálny počet presunov. V oboch prípadoch použijeme naše dobre známe prvky z predchádzajúcich príkladov:

$M_{min} = 13$ pre postupnosť

94 67 44 55 12 42 18 06

$M_{max} = 24$ pre postupnosť

18 42 12 44 06 55 67 94

Uvedený počet presunov je približne $\frac{1}{2} n . \log n$, pričom odchýlky od tejto hodnoty sú pomerne malé.

2.2.6 TRIEDENIE ROZDEĽOVANÍM

Po prebratí dvoch efektívnejších metód triedenia, postavených na princípoch vkladania a výberu uvedieme tretiu, vylepšenú metódu, vychádzajúcu z princípov triedenia výmenou. Vzhľadom na to, že z uvedených troch algoritmov triedenia bolo bublinové triedenie v priemere najmenej efektívne, očakávali by sme výrazný faktor zlepšenia. Výsledok je skutočne prekvapujúci. Vylepšená metóda triedenia výmenou, o ktorej budeme podrobne hovoriť v tomto odseku, predstavuje najlepšiu metódu triedenia v poli. Vzhľadom na jej vynikajúce parametre ju jej autor C. A. Hoare nazval rýchle triedenie.[1] Princíp algoritmu je založený na skutočnosti, že najefektívnejšie sú tie výmeny, ktoré sa vykonávajú na veľké vzdialenosti. Predpokladajme, že máme n prvkov s obráteným poradím ich kľúčov. Tieto je možné utriediť pomocou $n/2$ výmen. Najprv sa pritom porovnajú prvky na ľavom a pravom konci postupnosti, a potom sa postupne pokračuje z oboch strán smerom k opačnému koncu. Pochopiteľne, že je to možné iba vtedy, keď vieme, že ich poradie je obrátené. No napriek tomu máme z toho príkladu predsa len nejaký osoh. Pokúsime sa ho zužitkovať pri konštrukcii algoritmu, ktorý by bol schopný realizovať nasledujúce úkony: vybrať ľubovoľný prvok x triedeného poľa; prehľadať prvky zľava, pokiaľ sa nenájde prvok $a_i > x$, potom prehľadať prvky sprava, pokiaľ sa nenájde

[1] Pretože tento algoritmus je všeobecne známy pod pojmom Quicksort [2-5] a [2-6], budeme v ďalšom používať tento termín — pozn. prekl.

prvok $a_j < x$. Ďalej treba tieto dva prvky vymeniť a pokračovať v procese prehľadávania a výmeny dovtedy, kým sa ľavé a pravé prehľadávania nestretnú v strede poľa. Výsledkom týchto úkonov je rozdelenie poľa na dva úseky; v ľavom sú všetky prvky menšie ako prvok x, v pravom sú všetky prvky väčšie ako x. Proces rozdelenia poľa na úseky vzhľadom na prvok x je vyjadrený vo forme procedúry v programe 2.9. Všimnime si, že relácie $>$ a $<$ sú nahradené reláciami $\geq$ a $\leq$, ktorých negácie v príkaze cyklu **while** sú $<$ a $>$. Vzhľadom na tieto zmeny prvok x vystupuje ako zarážka pre obidve prehľadávania.

PROGRAM 2.9. *Rozdelenie poľa na úseky*

```
procedure rozdeľ;
  var w, x: prvok;
begin i: = 1; j: = n;
    vyber náhodne prvok x;
    repeat
      while a[i].kľúč < x.kľúč do i: = i + 1;
      while x.kľúč < a[j].kľúč do j: = j − 1;
      if i ≦ j then
        begin w: = a[i]; a[i]: = a[j]; a[j]: = w;
              i: = i + 1; j: = j − 1
        end
    until i > j
end
```

Ak máme napr. pole kľúčov

44 55 12 42 94 06 18 67

a za prvok x zvolíme kľúč s hodnotou 42, tak potrebujeme dve výmeny medzi prvkami, aby sme rozdelili pole na dva úseky. Výsledné hodnoty indexov sú: $i = 5$ a $j = 3$. Kľúče $a_1, \ldots, a_{i-1}$ sú menšie, alebo sa rovnajú kľúču prvku x, kľúče $a_{j+1}, \ldots, a_n$ sú väčšie, alebo sa rovnajú prvku x. Dostávame teda dva úseky poľa:

$$a_k.kľúč \leq x.kľúč \quad \text{pre} \quad k = 1, \ldots, i-1 \qquad (2.14)$$
$$a_k.kľúč \geq x.kľúč \quad \text{pre} \quad k = j+1, \ldots, n$$

pričom

$$a_k.kľúč = x.kľúč \quad \text{pre} \quad k = j+1, ..., i-1$$

Tento algoritmus je veľmi priamočiary a efektívny, pretože najviac porovnávané premenné i, j a prvok x možno v priebehu prehľadávania umiestniť do rýchlych registrov počítača. Môže byť však aj ťažkopádny, a to v prípade n rovnakých kľúčov, ktoré vyžadujú $n/2$ výmen. Tieto nepotrebné výmeny možno ľahko odstrániť zmenou prehľadávacích príkazov za príkazy

```
while a[i].kľúč ≦ x.kľúč do i: = i + 1;
while x.kľúč ≦ a[j].kľúč do j: = j − 1;
```

V tomto prípade však zvolený prvok x triedeného poľa prestáva plniť funkciu zarážky prehľadávania, a tak môže dôjsť k prekročeniu hraníc poľa. Aby sme sa tomuto nebezpečenstvu vyhli, je potrebné stanoviť zložitejšie podmienky ukončenia procesu prehľadávania. Jednoduchosť podmienok použitých v programe 2.9 má za následok dodatočné výmeny, ktoré sa v priemerných prípadoch vyskytujú pomerne zriedkavo. Nepatrnú úsporu môžeme dosiahnuť zmenou podmienky určujúcej ukončenie jednotlivých prechodov algoritmu, a to z podmienky $i \leqq j$ na podmienku $i < j$. Túto zmenu však treba ešte zohľadniť v rámci príkazov

$$i := i + 1 \quad \text{a} \quad j := j - 1$$

ktoré budú potom obsahovať zvláštnu podmienkovú klauzulu. Na ďalšom príklade ukážeme opodstatnenosť tejto klauzuly. Zvoľme prvok $x = 2$:

1 1 1 2 1 1 1

Po prvom prechode (prehľadaní a výmenách) pole nadobudne tvar

1 1 1 1 1 1 2

Premenné i a j budú mať hodnoty 5 a 6. Druhým prechodom algoritmu sa pole nezmení a premenné i a j nadobudnú hodnoty 7 a 6. Všimnime si, že ak by sme nebrali do úvahy podmienku $i \leqq j$, došlo by k chybnej výmene prvkov a_6 a a_7.

Správnosť uvedeného algoritmu rozdelenia poľa overíme tak, že zistíme, či dve tvrdenia, uvedené v (2.14), sú invariantmi cyklu **repeat**.

Pri vstupe do cyklu, t. j. v prípade, že $i = 1$ a $j = n$, sú tvrdenia pravdivé, pri ukončení cyklu, t. j. pri $i > j$, implikujú požadovaný výsledok.

Pripomeňme si, že naším cieľom nebolo iba rozdeliť pole na úseky vzhľadom na prvok x, ale ho aj utriediť. Ale od rozdeleného poľa k utriedenému je už iba krôčik, najmä keď si uvedomíme, že ten istý rozdeľovací proces môžeme v ďalšom aplikovať na jednotlivé úseky, potom na úseky úsekov atď., až získame len jednoprvkové úseky. Tento postup dokumentuje program 2.10.

PROGRAM 2.10. *Triedenie rozdeľovaním. Quicksort (rekurzívna verzia)*

```
procedure quicksort;
  procedure trieď (l, r: index);
    var i, j: index; x, w: prvok;
  begin i: = l; j: = r;
       x: = a[(l + r) div 2];
    repeat
      while a[i].kľúč < x.kľúč do i: = i + 1;
      while x.kľúč < a[j].kľúč do j: = j − 1;
      if i ≦ j then
      begin w: = a[i]; a[i]: = a[j]; a[j]: = w;
            i: = i + 1; j: = j − 1
      end
    until i > j;
    if l < j then trieď (l, j);
    if i < r then trieď (i, r)
  end;
begin trieď (1, n)
end {quicksort}
```

Procedúra trieď je volaná rekurzívne. Použitie rekurzie v algoritmoch predstavuje veľmi účinnú programovaciu techniku, o ktorej budeme podrobne hovoriť v kapitole 3. My teraz ukážeme, ako sa rekurzívny algoritmus dá vyjadriť v nerekurzívnom tvare. Najbežnejším spôsobom odstránenia rekurzie je jej transformácia na iteráciu. Táto vyžaduje navyše pomocnú pamäť a určité prídavné operácie, o ktorých sa zmienime v ďalšom.

Ako uskutočniť proces odstránenia rekurzie? Háčik je v tom, že v priebehu realizácie algoritmu vytvárame a udržujeme zoznam tých úsekov poľa, ktoré treba ešte spracovať. V rámci každého kroku algoritmu vznikajú dva nové úseky, z ktorých iba jeden môže byť vzápäti spracovaný. Identifikačné údaje o druhom úseku musíme pridať k uvedenému zoznamu. Zrejme, že tieto údaje sa budú v zozname nachádzať v opačnom poradí, v akom budú im zodpovedajúce úseky poľa spracúvané. To znamená, že úsek, ktorý bol uložený do zoznamu (vo forme svojich identifikačných údajov) ako prvý, bude spracovaný ako posledný. Zoznam sa teda správa ako pulzujúci zásobník. V programe 2.11, ktorý predstavuje iteratívnu verziu triediaceho algoritmu quicksort, sú identifikačné údaje úsekov poľa reprezentované dvojicou indexov, t.j. ľavým a pravým indexom úseku poľa, ktorý sa bude v ďalšom spracúvať. Zavedieme preto premennú typu pole, ktorú nazývame zásobník, a index s, ktorý označuje dvojicu indexov naposledy pridanú do zásobníka. Voľbe vhodnej veľkosti zásobníka m sa budeme venovať v rámci analýzy algoritmu quicksort.

PROGRAM 2.11. *Nerekurzívna verzia algoritmu quicksort*

```
procedure quicksort − 1;
  const m = 12;
  var i, j, l, r: index;
      x, w: prvok;
      s: 0..m;
      zásobník: array [1..m] of
                record l, r: index end;
begin s: = 1; zásobník [1].l: = 1; zásobník [1].r: = n;
  repeat {zober ďalší úsek z vrcholu zásobníka}
    l: = zásobník [s].l; r: = zásobník [s].r; s: = s − 1;
    repeat {rozdeľ a[l] ... a[r]}
      i: = l; j: = r; x: = a[(l + r) div 2];
      repeat
        while a[i].kľúč < x.kľúč do i: = i + 1;
        while x.kľúč < a[j].kľúč do j: = j − 1;
        if i ≦ j then
          begin w: = a[i]; a[i]: = a[j]; a[j]: = w;
                i: = i + 1; j: = j − 1
          end
      until i > j;
      if i < r then
      begin {ulož pravý úsek do zásobníka}
        s: = s + 1; zásobník [s].l: = i; zásobník [s].r: = r
      end;
      r: = j
    until l ≧ r
  until s = 0
end {quicksort − 1}
```


Analýza algoritmu quicksort

Aby sme mohli uskutočniť analýzu algoritmu quicksort, musíme najprv preskúmať proces rozdelenia poľa na úseky. Voľbou prvku x dôjde k preusporiadaniu celého poľa, takže potrebujeme presne n porovnaní. Počet výmen prvkov poľa určíme na základe pravdepodobnosti nasledujúcou úvahou. Predpokladajme, že prvky poľa, ktoré máme rozdeliť na dva úseky, obsahujú n kľúčov: 1, ..., n. Vyberme nejaký prvok x. Po rozdelení poľa na úseky sa bude prvok x nachádzať v poli na x-tej pozícii. Počet potrebných výmen sa bude rovnať súčinu počtu prvkov v ľavom úseku $(x - 1)$ a pravdepodobnosti výskytu kľúča, ktorý treba vymeniť. Kľúč sa vymení, ak nie je menší ako prvok x. Táto pravdepodobnosť je $(n - x + 1)/n$. Očakávaný počet výmen pre všetky možné voľby prvku x je potom daný týmto vzťahom:

$$M = \frac{1}{n} \sum_{x=1}^{n} \frac{x-1}{n} \cdot (n - x + 1) = \frac{n}{6} - \frac{1}{6n} \tag{2.15}$$

Vidíme, že očakávaný počet výmen je približne $n/6$.

Predpokladajme ďalej, že pri voľbe prvku x budeme mať šťastie a vždy vyberieme medián poľa. Každým rozdelením poľa na úseky potom dostaneme dve polovice, čím bude počet potrebných prechodov triedenia $\log n$. Celkový počet porovnaní bude $n . \log n$ a počet potrebných výmen $n/6 . \log n$.

Nemožno, pravdaže, očakávať, že sa nám zakaždým podarí vybrať medián. V skutočnosti je pravdepodobnosť výberu iba $1/n$. Je však prekvapujúce, že ak je hranica vybraná náhodne, priemerná účinnosť programu quicksort je v porovnaní s optimálnym prípadom horšia iba o faktor $2 . \ln 2$.

Ale quicksort má aj svoje úskalia. Predovšetkým funguje dobre pre malé hodnoty n ako i všetky prepracovanejšie metódy. Jeho výhoda pred inými prepracovanejšími metódami triedenia spočíva v ľahkosti, s akou sa môže priama metóda triedenia použiť na spracovanie malých úsekov. Výhodné je to obzvlášť vtedy, ak berieme do úvahy rekurzívnu verziu programu.

Ešte však zostáva otázka najhoršieho prípadu. Ako pracuje quicksort vtedy? Odpoveď nás, žiaľ, sklame a odhalí jednu slabinu rýchleho triedenia (z ktorého sa v týchto prípadoch stane pomalé triedenie). Vezmime napr. nešťastný prípad, v ktorom sa zakaždým najväčšia hodnota úseku náhodou vyberie ako prvok x. Potom sa v každom kroku rozpolí segment n prvkov na ľavý úsek s $n - 1$ a na pravý úsek s jediným prvkom. Výsledkom je, že namiesto (iba) $\log n$ rozdelení potrebujeme až n rozdelení a účinnosť v najhoršom prípade je rádu n^2.

Zrejme, že rozhodujúcim krokom je výber prvku x. V našom vzorovom programe je vybraný ako prostredný prvok. Všimnite si, že skoro tak dobre by sa mohol vybrať buď prvý, alebo posledný prvok $a[l]$ alebo $a[r]$. Najhorším prípadom je tu utriedené zdrojové pole; program quicksort potom vykazuje určitý odpor k triedeniu a uprednostneniu usporiadaných polí. Pri vyberaní prostredného prvku je zvláštna charakteristika programu quicksort menej zrejmá, pretože utriedené zdrojové pole sa stane optimálnym prípadom! V skutočnosti je priemerný výkon o niečo lepší, ak sa zvolí prostredný prvok. Hoare navrhuje, aby sa výber prvku x urobil náhodne, alebo aby sa zvolil ako medián malej vzorky, pozostávajúcej povedzme z troch kľúčov [2.12] a [2.13]. Takýto uvážený výber sotva ovplyvní priemerný výkon programu quicksort, ale značne zlepší výkon v najhoršom prípade. Prejavom toho je, že triedenie na základe programu quicksort je niečo podobné ako hazardná hra, pri ktorej by si človek mal byť vedomý toho, koľko si môže dovoliť stratiť, ak sa mu prestane dariť.

Z tejto skúsenosti možno zobrať jedno dôležité ponaučenie, ktoré sa

týka priamo programátora. Aké sú dôsledky správania sa v najhoršom prípade na účinnosť programu 2.11? Uvedomili sme si, že výsledkom každého rozdelenia je, že pravý úsek má iba jeden prvok; požiadavku na triedenie tohto úseku odložíme do zásobníka na neskoršie spracovanie. Teda maximálny počet požiadaviek, a tým aj celková požadovaná veľkosť zásobníka je n, čo je, samozrejme, úplne neprijateľné. (Poznamenávame, že rekurzívnou verziou algoritmu sa situácia nezlepší, naopak skôr zhorší, pretože systém umožňujúci rekurzívne volanie procedúr bude musieť automaticky uchovávať hodnoty lokálnych premenných a parametrov v implicitnom zásobníku, a to pre každé volanie procedúry.) Zlepšenie dosiahneme tým, že do zásobníka uložíme údaje o väčšom úseku (ktorý spracujeme neskôr) a pokračujeme ďalším delením menších úsekov. V takomto prípade bude veľkosť zásobníka ohraničená na $m = \log_2 n$.

Na základe uvedených úvah bude potrebné modikovať program 2.11. Zmeny sa budú týkať tých miest algoritmu, kde sa určujú indexy ešte neutriedených úsekov poľa. Znázorňuje ich nasledujúca časť programu:

```
if j - l < r - i then
begin if i < r then
      begin {ulož údaje v pravom úseku do zásobníka}
            s: = s + 1; zásobník [s] . l: = i;
            zásobník [s] . r: = r
      end;
      r: = j {pokračuj v triedení ľavého úseku}
end else                                                      (2.16)
begin if l < j then
      begin {ulož údaje v ľavom úseku do zásobníka}
            s: = s + 1; zásobník [s] . l: = l;
            zásobník [s] . r: = j
      end;
      l: = i {pokračuj v triedení pravého úseku}
end
```

2.2.7 HĽADANIE MEDIÁNA

Medián postupnosti n prvkov je definovaný ako prvok, ktorý je menší (alebo sa rovná) ako jedna polovica všetkých prvkov a súčasne väčší (alebo sa rovná) ako druhá polovica všetkých prvkov. Napríklad mediánom postupnosti

16 12 99 95 18 87 10

je prvok 18.

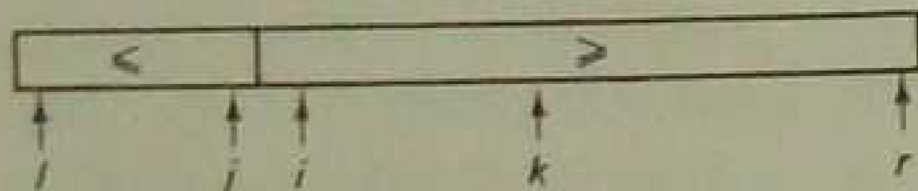
Problém nájdenia mediána sa obyčajne dáva do súvislosti s triedením, pretože jednou zaručenou metódou vyhľadávania mediána je takýto postup: Postupnosť najprv utriedime, a potom jednoducho vyberieme prostredný prvok. Je zrejmé, že to asi nebude najoptimálnejší spôsob zistenia mediána. Delenie postupnosti na úseky, ako sme uviedli v programe 2.9, predstavuje oveľa rýchlejšiu metódu. Univerzálna metóda, o ktorej budeme v ďalšom hovoriť, sa zaoberá problémom nájdenia k-tého najmenšieho prvku postupnosti. Medián je potom špeciálnym prípadom takéhoto prvku, pre ktorý platí $k = n/2$.

Algoritmus, ktorý vynašiel C. A. R. HOARE [2-4] pracuje takto: Najprv sa použije procedúra rozdelenia poľa na úseky z algoritmu quicksort pre indexy s hodnotami $l = 1$ a $r = n$. Za prvok x zvolíme prvok $a[k]$. Výsledné hodnoty indexov i a j budú také, že platí:

1. $a[h] \leqq x$ pre všetky $h < i$.
2. $a[h] \geqq x$ pre všetky $h > j$. (2.17)
3. $i > j$.

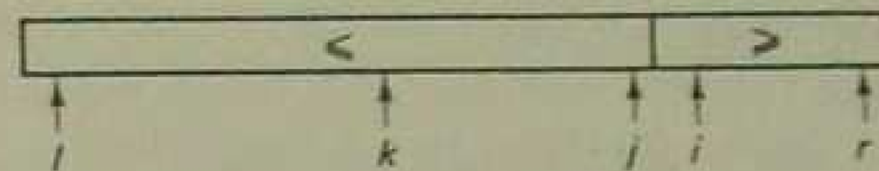
Rozdelením poľa na úseky môžu nastať tieto tri prípady:

1. Prvok x bol príliš malý. Hranica medzi dvoma úsekmi je pod požadovanou hodnotou k a proces rozdelenia na úseky treba zopakovať v rámci úseku $a[i], \ldots, a[r]$ (*obr. 2.9*).



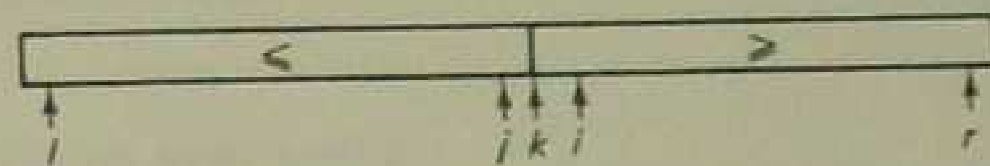
Obr. 2.9. Malý prvok x

2. Prvok x bol príliš veľký. Operáciu rozdelenia na úseky treba zopakovať v rámci úseku $a[l], \ldots, a[j]$ (*obr. 2.10*).



Obr. 2.10. Veľký prvok x

3. $j < k < i$: prvok $a[k]$ delí pole na dva úseky v súlade s požadovanými vlastnosťami, a preto je jeho k-tým najmenším prvkom (*obr. 2.11*).



Obr. 2.11. Správne zvolený prvok x

Rozdeľovací proces treba opakovať dovtedy, pokiaľ nevznikne tretí z uvedených prípadov. Túto iteráciu možno vyjadriť nasledujúcim algoritmom:

```
l: = 1; r: = n;
while l < r do
  begin x: = a[k];
    rozdeľ (a[l] ... a[r]);                (2.18)
    if j < k then l: = i;
    if k < i then r: = j
  end
```

Formálny dôkaz správnosti tohto algoritmu je uvedený v originálnom Hoarovom článku. Kompletný program na nájdenie k-tého najmenšieho prvku postupnosti (nazývame ho nájdi) sa odvodí z toho algoritmu a je uvedený v programe 2.12.

PROGRAM 2.12. *Nájdenie k-tého prvku*

```
procedure nájdi (k : integer);
  var l, r, i, j, w, x: integer;
begin l: = 1; r: = n;
  while l < r do
  begin x: = a[k]; i: = l; j: = r;
    repeat {rozdeľ}
      while a[i] < x do i: = i + 1;
```

```
      while x < a[j] do j: = j - 1;
      if i ≦ j then
        begin w: = a[i]; a[i]: = a[j]; a[j]: = w;
              i: = i + 1; j: = j - 1
        end
    until i > j;
    if j < k then l: = i;
    if k < i then r: = j
  end
end {nájdi}
```

Ak predpokladáme, že sa v priemere každým rozdelením rozpolí dĺžka úseku, v ktorom sa nachádza hľadaný prvok, tak počet potrebných porovnaní je

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \doteq 2n \qquad (2.19)$$

Vidíme, že počet porovnaní je úmerný celkovému počtu prvkov. Tento vzťah najlepšie dokumentuje efektívnosť programu nájdi na vyhľadávanie mediánov a podobných prvkov. Súčasne zvýrazňuje jeho prednosť pred priamočiarou metódou, ktorou je utriedenie všetkých kandidátov a náväzné vybratie k-tého prvku (najlepšia z týchto metód má zložitosť $n \,.\, \log n$). V najhoršom prípade, keď sa v rámci každého rozdeľovacieho kroku zmenšuje počet kandidátov iba o jednotku, je počet potrebných porovnaní rádovo n^2. Ale v prípade malého počtu prvkov (menšieho ako desať) ťažko nájdeme pri použití tohto algoritmu nejakú výhodu.

2.2.8 POROVNANIE METÓD VNÚTORNÉHO TRIEDENIA

Na záver prehľadu metód vnútorného triedenia sa pokúsime porovnať ich efektívnosť. Skôr ako uvedieme zložitosti jednotlivých algoritmov, zopakujme zavedenú symboliku: symbolom n označujeme počet triedených prvkov, symboly C a M vyjadrujú celkové počty potrebných porovnaní a presunov prvkov. Pomocou týchto symbolov vyjadríme

analytické vzorce pre všetky tri priame triediace metódy, ktoré sú v *tab. 2.8*. Jednotlivé stĺpce tejto tabuľky obsahujú údaje o minimálnych (min), priemerných (priem) a maximálnych (max) počtoch porovnaní a presunoch prvkov uvažovaných pre všetky $n!$ permutácií prvkov. Pre prepracované metódy vnútorného triedenia nie sú známe také jednoduché a presné vzorce. Z analýz týchto algoritmov vyplýva, že výpočtová zložitosť Shellovho triedenia je $c_1 \,.\, n^{1.2}$ a $c_i \,.\, n \,.\, \log(n)$ v prípade triedenia haldou a quicksortom.

Porovnanie priamych metód triedenia *Tabuľka 2.8*

	min	*priem*	*max*
Priame vkladanie	$C = n - 1$ $M = 2(n - 1)$	$(n^2 + n - 2)/4$ $(n^2 - 9n - 10)/4$	$(n^2 - n)/2 - 1$ $(n^2 + 3n - 4)/2$
Priamy výber	$C = (n^2 - n)/2$ $M = 3(n - 1)$	$(n^2 - n)/2$ $n(\ln n + 0.57)$	$(n^2 - n)/2$ $n^2/4 + 3(n - 1)$
Priama zmena (bublinové triedenie)	$C = (n^2 - n)/2$ $M = 0$	$(n^2 - n)/2$ $(n^2 - n) * 0.75$	$(n^2 - n)/2$ $(n^2 - n) * 1.5$

Uvedené vzorce poskytujú iba približnú mieru účinnosti algoritmov triedenia. Udávajú efektívnosť ako funkciu celkového počtu prvkov n a umožňujú delenie triediacich algoritmov na jednoduché priame metódy (n^2) a prepracované alebo logaritmické metódy ($n \,.\, \log n$). Pre praktické účely je užitočné mať k dispozícii experimentálne údaje, na základe ktorých sa dajú presnejšie určiť koeficienty c_i, detailnejšie charakterizujúce jednotlivé triediace metódy. Navyše sme pri analýze algoritmov nebrali do úvahy zložitosti iných operácií ako napr. porovnania kľúčov a presuny prvkov a riadenie cyklu. Prirodzene, že tieto faktory vo veľkej miere závisia od konkrétnych počítačových systémov. V *tab. 2.9* sú uvedené časy výpočtov jednotlivých algoritmov triedenia (v milisekundách), implementovaných v jazyku pascal na počítači CDC 6400. Nachádzajú sa v nej časové údaje pre triedenie už utriedeného poľa, náhodne usporiadaného a obrátene usporiadaného poľa.

Tabuľka 2.9

Časy algoritmov triedenia

	Utriedené		Náhodne usporiadané		Obrátene usporiadané	
Priame vkladanie	12	23	366	1444	704	2836
Binárne vkladanie	56	125	373	1327	662	2490
Priamy výber	489	1907	509	1956	695	2675
Bublinové triedenie	540	2165	1026	4054	1492	5931
Bublinové triedenie s indikátorom*	5	8	1104	4270	1645	6542
Triedenie pretriasaním	5	9	961	3642	1619	6520
Shellovo triedenie	58	116	127	349	157	492
Triedenie haldou	116	253	110	241	104	226
Quicksort	31	69	60	146	37	79
Triedenie zlučovaním**	99	234	102	242	99	232

* Pozri odsek 2.3.1.

** Indikátor registruje poslednú výmenu prvkov v poli. Jeho zavedením sa zvýšila celková účinnosť bublinového triedenia.

Každý stĺpec v tabuľke sa navyše skladá z dvoch podstĺpcov, z ktorých ľavý udáva časy pre 256 triedených prvkov, pravý pre 512 prvkov. Uvedené časové údaje očividne oddeľujú priame metódy (n^2) od logaritmických metód ($n \, . \log n$). Pozoruhodné sú nasledujúce výsledky:

1. Výhody metódy binárneho vkladania sú oproti priamemu vkladaniu skutočne nepatrné a v prípade už utriedenej postupnosti sa výhoda zmení dokonca na nevýhodu.

2. Bublinové triedenie je jasne najefektívnejšou metódou. Jeho vylepšená verzia — triedenie pretriasaním — je takisto horšia ako priame vkladanie alebo priamy výber (okrem prípadu utriedeného poľa).

3. Quicksort je dvoj- až trojnásobne efektívnejšia metóda než triedenie haldou. Jeho triediaca rýchlosť je prakticky rovnaká pre obidva patologické prípady, t. j. pre obrátene usporiadané pole, ako aj pre už utriedené pole.

Musíme však dodať, že triedené údaje boli reprezentované iba prostredníctvom ich triediacich kľúčov, t. j. bez ďalších relevantných informácií. To, samozrejme, nie je najreálnejšia situácia, a preto v *tab. 2.10*, kde máme opäť uvedené časy pre jednotlivé metódy triedenia, sme brali navyše do úvahy aj zložitejšie štruktúry triedených prvkov. V našom prípade sme za veľkosť triedeného prvku zvolili sedemnásobok veľkosti

Tabuľka 2.10

Časy algoritmov triedenia pre prvky s prídavnými informáciami

	Utriedené		Náhodne usporiadané		Obrátene usporiadané	
Priame vkladanie	12	46	366	1129	704	2150
Binárne vkladanie	56	76	373	1105	662	2070
Priamy výber	489	547	509	607	695	1430
Bublinové triedenie	540	610	1026	3212	1492	5599
Bublinové triedenie s indikátorom	5	5	1104	3237	1645	5762
Triedenie pretriasaním	5	5	961	3071	1619	5757
Shellovo triedenie	58	186	127	373	157	435
Triedenie haldou	116	264	110	246	104	227
Quicksort	30	55	60	137	37	75
Triedenie zlučovaním*	99	196	102	195	99	187

* Pozri odsek 2.3.1.

pamäti, ktorú zaberá triediaci kľúč. Ľavý podstĺpec každého stĺpca tabuľky udáva časy triedenia pre prvky bez prídavných informácií, pravý podstĺpec časy pre zväčšené prvky. Počet triedených prvkov bol 256.

Na základe údajov uvedených v tejto tabuľke môžeme konštatovať, že:

1. Triedenie metódou výberu je najlepšou metódou spomedzi priamych algoritmov triedenia.

2. Bublinové triedenie je najhoršou metódou triedenia a iba jeho „vylepšená" verzia — triedenie pretriasaním — je o niečo horšia (aj to len v prípade obrátene usporiadaného poľa).

3. Quicksort si ešte upevnil pozíciu najrýchlejšej metódy triedenia, preto ho považujeme za najlepší algoritmus vnútorného triedenia.

2.3 TRIEDENIE SEKVENČNÝCH SÚBOROV

2.3.1 PRIAME ZLUČOVANIE

Uvedené metódy vnútorného triedenia sa nedajú použiť v prípadoch, keď je súbor triedených údajov taký veľký, že sa nezmestí do operačnej

pamäti počítača. Vtedy zvyčajne pracujeme s údajmi, ktoré sú uložené na rôznych periférnych sekvenčných pamäťových zariadeniach, akými sú napr. magnetické pásky. Charakteristickou vlastnosťou sekvenčných súborov je, že v každom okamihu je priamo prístupný práve jeden prvok súboru. Vzhľadom na takéto pomerne silné obmedzenie (oproti poľu) musíme pri triedení súborov používať iné techniky triedenia. Jednou z najzaužívanejších a najdôležitejších je metóda triedenia zlučovaním. Zlučovanie znamená spájanie dvoch (alebo viacerých) usporiadaných postupností do jednej usporiadanej postupnosti prostredníctvom opakovaného výberu spomedzi momentálne prístupných prvkov. Zlučovanie je podstatne jednoduchšie ako triedenie a používa sa ako pomocná operácia pre zložitejší proces triedenia sekvenčných súborov. Jedným z najjednoduchších algoritmov vonkajšieho triedenia zlučovaním je priame zlučovanie, ktoré pracuje takýmto spôsobom:

1. Postupnosť a sa rozdelí na dve polovice, ktoré označíme b a c.
2. Spájaním prvkov do usporiadaných dvojíc sa postupnosti b a c zlúčia do novej postupnosti a.
3. Na novovytvorenú postupnosť a sa aplikujú kroky 1 a 2, čím dostaneme z usporiadaných dvojíc usporiadané štvorice.
4. Prvé tri kroky sa postupne opakujú, čím z usporiadaných štvoríc získame usporiadané osmice atď., pričom v rámci každého kroku cyklu sa zdvojnásobí dĺžka zlučovaných podpostupností. Celý algoritmus končí v okamihu, keď je celá podpostupnosť usporiadaná.

Ako príklad zoberme nasledujúcu postupnosť:

44 55 12 42 94 18 06 67

V rámci kroku 1 sa postupnosť rozdelí na dve podpostupnosti:

44 55 12 42
94 18 06 67

Zlúčením jednotlivých prvkov (ktoré môžeme považovať za usporiadané postupnosti s jednotkovou dĺžkou) do usporiadaných dvojíc dostávame

44 94′ 18 55′ 06 12′ 42 67

Opätovným rozdelením (vzhľadom na stred postupnosti) a zlúčením usporiadaných dvojíc dostávame

06 12 44 94′ 18 42 55 67

Tretím rozdelením a zlúčením dostávame konečnú postupnosť v požadovanom usporiadanom tvare

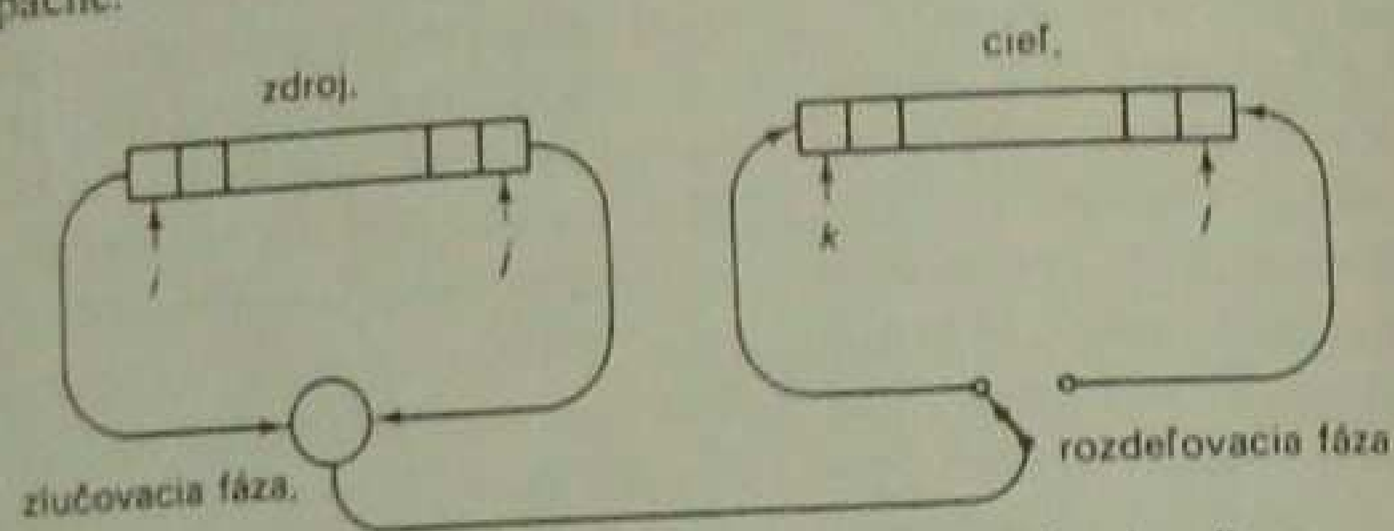
06 18 18 42 44 55 67 94

Každá operácia, pri ktorej sa manipuluje naraz s celou množinou údajov, sa nazýva *fáza*. Najmenší podproces, ktorého opakovaním sa realizuje triedenie, sa nazýva prechod alebo krok. V uvedenom príklade sme potrebovali tri prechody, z ktorých každý pozostával z fáz rozdelenia postupnosti a zlučovania. Na uskutočnenie triedenia sú potrebné tri pásky; triediaci proces sa preto nazýva trojpáskové zlučovanie. Vzhľadom na to, že rozdeľovacia fáza nespôsobuje preusporiadanie prvkov, nezaraďujeme ju do triedenia. V tomto zmysle je vlastne neproduktívna, aj keď predstavuje takmer polovicu všetkých kopírovacích operácií. Našou snahou preto bude odstrániť ju, čo sa dá ľahko urobiť tak, že spojíme rozdeľovanie a zlučovanie do jednej fázy. Namiesto zlučovania prvkov do jednej postupnosti sa tieto budú v rámci zlučovacieho procesu rovnomerne distribuovať na dve pásky, ktoré budú potom predstavovať vstup do ďalšej fázy. Na rozdiel od prvej dvojfázovej metódy triedenia zlučovaním sa táto metóda nazýva jednofázové zlučovanie alebo aj vyvážené zlučovanie. Je podstatne lepšie ako prvá metóda, pretože vyžaduje iba polovicu kopírovacích operácií, aj keď je potrebná štvrtá páska.

V ďalších článkoch tejto kapitoly ukážeme tvorbu niektorých typických zlučovacích programov. Na úvod, pre jednoduchosť, uvažujeme o reprezentácii údajov poľom, ktoré však budeme prehľadávať výlučne sekvenčným spôsobom. V ďalších verziách triedenia zlučovaním už budeme manipulovať so súbormi. Použitie dvoch rôznych reprezentácií údajov nám jednak umožní porovnať dva programy a súčasne zvýrazní veľkú závislosť štruktúry programu od zvolenej reprezentácie údajov.

Dva sekvenčné súbory môžeme jednoducho nahradiť poľom, ak sa naň pozeráme ako na postupnosť s dvoma koncami. Potom namiesto zlučovania dvoch súborov môžeme v rámci zlučovacej fázy postupne vyberať prvky z jednotlivých koncov poľa. Všeobecnú schému takejto kombinovanej fázy zlučovania a rozdeľovania možno vidieť na

obr. 2.12. Na konci každého prechodu sa zmení cieľová postupnosť (tvorená je výstupom zo zlučovacieho procesu). Táto situácia nastáva napr. pri vytvorení usporiadaných dvojíc v rámci prvého prechodu, pri vytvorení usporiadaných štvoríc v rámci druhého prechodu atď. Zmenou cieľa dochádza k rovnomernému generovaniu obidvoch cieľových postupností reprezentovaných dvoma koncami poľa. Záverom každého prechodu sa mení funkcia polí, t. j. zdrojové sa stáva cieľovým a opačne.



Obr. 2.12. Triedenie metódou priameho zlučovania pomocou dvoch polí

Ďalšie zjednodušenie programu dosiahneme zlúčením dvoch koncepčne rôznych polí do jedného poľa s dvojnásobnou dĺžkou. Údaje budú potom reprezentované pomocou poľa a, ktorého deklarácia je

$$a: \textbf{array}\ [1\,..\,2*n]\ \textbf{of}\ \text{prvok} \tag{2.20}$$

Ďalej budeme potrebovať štyri premenné, ktoré nám poslúžia na reprezentáciu zdrojovej a cieľovej postupnosti. Indexmi i a j budeme označovať dva zdrojové prvky, indexmi k a l dva cieľové prvky (*obr. 2.12*). Zdrojové (začiatočné) údaje, ktoré máme utriediť, sú tvorené prvkami $a_1, a_2, \ldots, a_n$. Na označenie smeru toku údajov v rámci jednotlivých fáz budeme potrebovať boolovskú premennú (nazveme ju *nahor*), ktorá bude mať hodnotu true, ak sa prvky $a_1, a_2, \ldots, a_n$ presúvajú smerom nahor, t. j. k prvkom $a_{n+1}, \ldots, a_{2n}$, resp. hodnotu false, ak sa prvky $a_{n+1}, \ldots, a_{2n}$ presúvajú smerom nadol, t. j. k prvkom $a_1, a_2, \ldots, a_n$. Hodnota premennej *nahor* sa mení od prechodu k prechodu. Poslednou premennou, ktorú potrebujeme na formulovanie algoritmu triedenia zlučovaním, je dĺžka podpostupností, ktoré sa majú zlučovať. Označí-

me ju symbolom p. Jej začiatočná hodnota sa bude rovnať jednotke ($p = 1$) a v každom ďalšom prechode sa jej hodnota zdvojnásobí. Kvôli jednoduchosti predpokladajme, že celkový počet prvkov sa vždy rovná mocnine dvojky. Prvú verziu algoritmu triedenia zlučovaním potom zapíšeme takto:

```
procedure triedeniezlučovanim;
  var i, j, k, l: index;
      nahor: boolean; p: integer;
begin nahor := true; p := 1;
  repeat {inicializácia indexov}
    if nahor then
      begin i := 1; j := n; k := n + 1; l := 2 * n
      end else                                              (2.21)
      begin k := 1; l := n; i := n + 1; j := 2 * n
      end;
    „zlúčenie p-prvkových podpostupností z postupnosti označených
    indexmi i a j do postupnosti označených indexmi k a l";
    nahor := ¬ nahor; p := 2 * p
  until p = n
end
```

V ďalšej etape vývoja tohto programu upresníme príkaz vyjadrený v úvodzovkách. Je jasné, že zlučovacia fáza, ktorá sa týka všetkých prvkov súboru, je sama o sebe postupnosťou zlučovaní p-prvkových podpostupností. Medzi jednotlivými čiastkovými zlučovaniami sa mení cieľová postupnosť z dolného konca na horný koniec poľa a naopak. To preto, aby sme zaručili rovnomernú distribúciu prvkov do obidvoch cieľových postupností. Ak je cieľom zlučovania dolný koniec poľa, tak je cieľová postupnosť určená indexom k, ktorého hodnota sa po každom presune prvku zväčší o jednotku. Ak je cieľom horný koniec poľa, celková postupnosť je určená indexom l, ktorého hodnota sa po každom presune prvku zníži o jednotku. Aby sme ešte zjednodušili proces zlučovania, budeme indexom k vždy označovať cieľovú postupnosť. Po každom zlúčení p-prvkových podpostupností sa hodnoty premenných k a l vymenia. Prírastok, resp. úbytok premenných k a l vyjadrime premennou h, ktorá môže nadobúdať hodnotu 1 alebo −1. Na základe

uvedených myšlienok môžeme zjemnenie zlučovacieho procesu vyjadriť nasledujúcim algoritmom:

```
h: = 1; m: = n; {m je počet prvkov, ktoré sa zlučujú}
repeat q: = p; r: = p; m: = m − 2 * p;
        „zlúčenie q prvkov z postupnosti i a r prvkov
        z postupnosti j; cieľová postupnosť je určená        (2.22)
        indexom k; prírastok je h";
    h: = −h;
    výmena hodnôt medzi indexmi k a l
until m = 0
```

Ďalším zjemnením algoritmu presnejšie sformulujeme príkaz zlučovania. Skôr, ako ho uvedieme, musíme ešte upozorniť na jednu dôležitú skutočnosť. Uvedomme si, že po realizácii zlučovania ostáva časť niektorej podpostupnosti neprázdna, a preto ju treba pripojiť k určenej cieľovej postupnosti pomocou jednoduchých kopírovacích operácií. Zjemnený príkaz zlučovania vrátane uvedených kopírovacích operácií bude vyzerať takto:

```
while (q ≠ 0) ∧ (r ≠ 0) do
begin {vyber prvok z postupnosti označených indexmi i a j}
  if a[i].kľúč < a[j].kľúč then
    begin „presuň prvok z postupnosti označnej indexom i do postup-
          nosti označenej indexom k";
          „nastav nové hodnoty indexov i a k";
          q: = q − 1
    end else
    begin „presuň prvok z postupnosti označenej indexom j do po-
          stupnosti označenej indexom k";
          „nastav nové hodnoty indexov j a k";
          r: = r − 1
    end
end;
„skopíruj koniec postupnosti označenej indexom i";
„skopíruj koniec postupnosti označenej indexom j"            (2.23)
```

Po ďalšom zjemnení uvedených kopírovacích operácií budeme mať

program triedenia zlučovaním úplný. Predtým, ako ho uvedieme v úplnom tvare, pokúsime sa ešte odstrániť nami zavedené obmedzenia, kladené na celkový počet triedených prvkov. Predpokladali sme, že tento počet sa musí rovnať mocnine dvojky. Súčasne preskúmame, ktoré časti programu budú postihnuté zrušením tohto obmedzenia. Ľahko sa presvedčíme, že najlepším spôsobom, ako zrušiť uvedené obmedzenie a prejsť na všeobecnejšiu situáciu, je postupovať podľa pôvodnej metódy. V ďalšom prípade to znamená, že proces zlučovania p-prvkových podpostupností budeme vykonávať dovtedy, pokým dĺžky zvyšujúcich zdrojových postupností budú menšie ako hodnota premennej p. Jediným miestom v programe, ktoré bude touto zmenou ovplyvnené, sú príkazy určujúce hodnoty premenných q a r, t. j. veľkosti postupností, ktoré sa budú zlučovať. Nasledujúce štyri príkazy nahradia uvedené tri príkazy:

```
q: = p;   r: = p;   m: = m − 2 * p
```

Čitateľ sa môže sám presvedčiť, že uvedenou zmenou dôjde k efektívnej implementácii špecifikovanej stratégie. Poznamenávame, že premenná m označuje celkový počet prvkov v oboch zdrojových postupnostiach, ktoré zostáva ešte zlúčiť.

```
if m ≧ p then q: = p else q: = m; m: = m − q;
if m ≧ p then r: = p else r: = m; m: = m − r;
```

Nakoniec je potrebné ešte upraviť podmienku ukončenia programu. Táto bude v našom prípade vyjadrená výrazom $p \geqq n$ namiesto pôvodného $p = n$. Je to podmienka, ktorou sa riadi priebeh vonkajšieho cyklu algoritmu.

Na základe uvedených modifikácií môžeme teraz uviesť kompletný algoritmus triedenia zlučovaním. Dokumentuje ho program 2.13.

PROGRAM 2.13. *Triedenie metódou priameho zlučovania*

```
procedure triedeniezlučovaním;
  var i, j, k, l, t: index;
      h, m, p, q, r: integer; nahor: boolean;
    {pole a má indexy z intervalu 1 .. 2 * n}
```

```
begin nahor: = true; p: = 1;
   repeat h: = 1; m: = n;
      if nahor then
      begin i: = 1; j: = n; k: = n + 1; l: = 2 * n
      end else
      begin k: = 1; l: = 1; i: = n + 1; j: = 2 * n
      end;
      repeat {zlúč postupnosti určené indexmi i a j do
               postupnosti určenej indexom k}
               {q je dĺžka i-tej postupnosti; r je dĺžka
               j-tej postupnosti}
         if m ≧ p then q: = p else q: = m; m: = m − q;
         if m ≧ p then r: = p else r: = m; m: = m − r;
         while (q ≠ 0) ∧ (r ≠ 0) do
         begin {zlučovanie}
            if a[i].kľúč < a[j].kľúč then
            begin a[k]: = a[i]; k: = k + h; i: = i + 1; q: = q − 1
            end else
            begin a[k]: = a[j]; k: = k + h; j: = j − 1; r: = r − 1
            end
         end;
            {skopíruj zvyšok j-tej postupnosti}
         while r ≠ 0 do
            begin a[k]: = a[j]; k: = k + h; j: = j − 1; r: = r − 1
            end;
            {skopíruj zvyšok i-tej postupnosti}
         while q ≠ 0 do
            begin a[k]: = a[i]; k: = k + h; i: = i + 1; q: = q − 1
            end;
         h: = −h; t: = k; k: = l; l: = t
      until m = 0;
      nahor: = ¬ nahor; p: = 2 * p
   until p ≧ n;
   if ¬ nahor then
   for i: = 1 to n do a[i]: = a[i + n]
end {triedeniezlučovaním}
```


Vzhľadom na to, že v každom prechode sa veľkosť premenej p zdvojnásobí a triedenie končí, len čo je splnená podmienka $p = n$, je počet potrebných prechodov algoritmu $\lceil \log_2 n \rceil$. Celkový počet presunov prvkov je presne

$$M = n \, . \, \lceil \log n \rceil \tag{2.24}$$

pretože v každom prechode algoritmu sa presúva presne n prvkov.

Počet C porovnaní kľúčov je dokonca menší ako M, pretože v operáciách kopírovania zvyškov postupnosti nie sú zahrnuté porovnania. Keďže sa však metóda triedenia zlučovaním obyčajne uplatňuje v súvislosti s používaním periférnych pamäťových zariadení, výpočty obsiahnuté v presunových operáciách prevažujú nad porovnaniami často o niekoľko rádov. Podrobná analýza počtu porovnaní má preto malý praktický význam.

Algoritmus triedenia zlučovaním sa celkom vyrovná i prepracovanejším metódam triedenia, o ktorých sme hovorili v predchádzajúcej kapitole. Administratívna réžia manipulácie s indexmi je však relatívne vysoká a rozhodujúcou nevýhodou je potreba pamäti s $2n$ prvkami. To je dôvod, prečo sa triedenie zlučovaním zriedka používa pre polia, t. j. pre údaje umiestnené v hlavnej pamäti. Parametre, charakterizujúce algoritmus triedenia zlučovaním v reálnom čase, sa nachádzajú v posledných riadkoch *tab. 2.9* a *tab. 2.10*. Ak porovnáme tieto čísla s parametrami triedenia haldou, je výsledok priaznivý v prospech triedenia zlučovaním, ale pre rýchle triedenie je výsledok porovnania nepriaznivý.

2.3.2 PRIRODZENÉ ZLUČOVANIE

Pri priamom zlučovaní nezískame žiadne výhody, keď sú údaje na začiatku už čiastočne utriedené. Dĺžka všetkých zlúčených podpostupností v k-tom prechode je menšia alebo sa rovná 2^k nezávisle od toho, či sú už dlhšie podpostupnosti usporiadané a či by sa mohli tiež zlučovať. V skutočnosti akékoľvek dve usporiadané podpostupnosti dĺžky m a n by sa mohli priamo zlúčiť do jednej postupnosti $m + n$ prvkov.

Triedenie zlučovaním, ktoré vždy zlučuje dve najdlhšie možné podpostupnosti, sa nazýva *triedenie prirodzeným zlučovaním*.

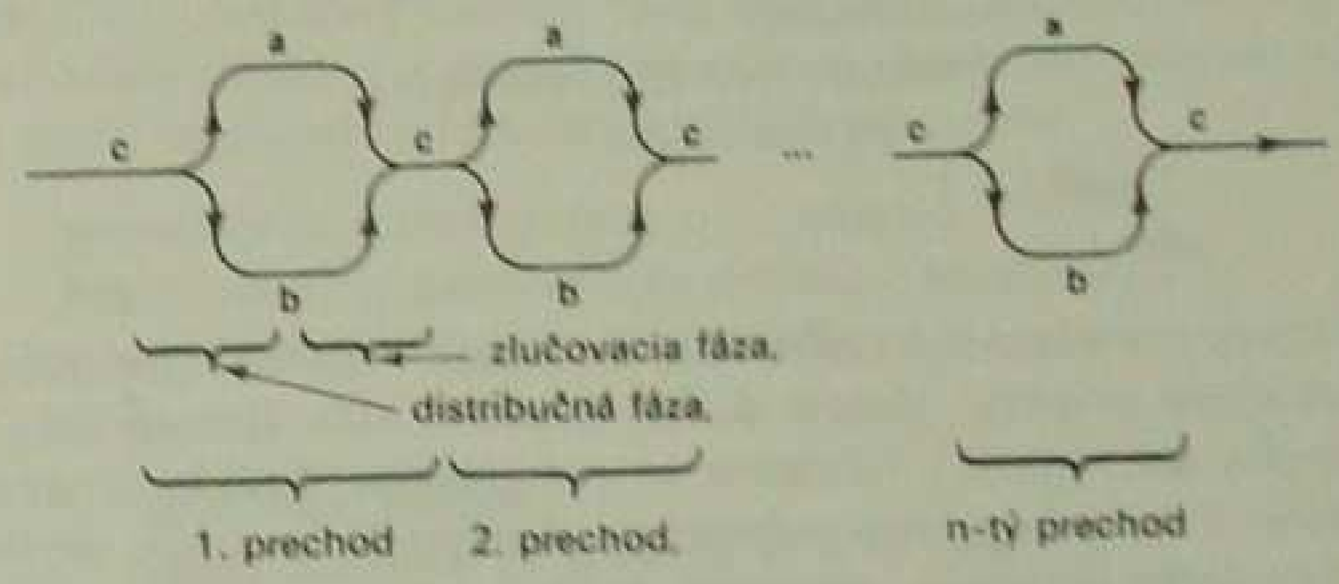
Usporiadaná postupnosť sa často nazýva *reťazec*. Keďže sa však slovo reťazec ešte častejšie používa na opis postupností znakov, budeme sa v našej terminológii pridŕžiavať Knutha a budeme používať namiesto reťazca slovo *beh*. Postupnosť $a_i \dots a_j$, kde

$$\begin{aligned} &a_k \le a_{k+1} \quad \text{pre} \quad k = i, \dots, j-1 \\ &a_{i-1} > a_i \\ &a_j > a_{j+1} \end{aligned} \tag{2.25}$$

nazývame *maximálny beh* alebo skrátene beh. Triedenie prirodzeným zlučovaním preto zlučuje (maximálne) behy namiesto postupností s pevnou, predurčenou dĺžkou. Behy majú takú vlastnosť, že ak sa zlúčia dve postupnosti s n behmi, vznikne jedna postupnosť presne s n behmi. Celkový počet behov sa potom v každom prechode zmenší o polovicu a počet potrebných presunov prvkov je v najhoršom prípade $n \cdot \lceil \log_2 n \rceil$, v priemernom ešte menej. Očakávaný počet porovnaní je však oveľa väčší, pretože okrem porovnaní potrebných pri výbere vhodného kandidáta potrebujeme na zistenie konca jednotlivých behov ešte ďalšie porovnania medzi jednotlivými, po sebe idúcimi prvkami každého súboru.

V ďalšej ukážke tvorby programu predvedieme tvorbu algoritmu prirodzeného zlučovania, pričom použijeme tú istú metódu postupného zjemňovania ako v prípade algoritmu priameho zlučovania. Nebudeme však už triediť prvky poľa, ale skutočné sekvenčné súbory. Použijeme pritom tri pásky a v rámci každého prechodu dve fázy. Prirodzené zlučovanie možno potom charakterizovať ako nevyvážené, dvojfázové, trojpáskové triedenie zlučovaním. Predpokladajme, že vstupné neusporiadané údaje sú reprezentované súborom c, v ktorom sa súčasne objavia aj výstupné utriedené údaje. (Pochopiteľne, že pri skutočných aplikáciách spracovania údajov sa kvôli bezpečnosti vstupné údaje najprv skopírujú z pôvodného súboru na súbor c.) Dve pomocné pásky označme symbolmi a a b. Ako sme už uviedli, každý prechod pozostáva z dvoch fáz, a to z distribúcie a zo zlučovania. V rámci distribučnej fázy sa behy rovnomerne rozdeľujú zo súboru c na pásky a, b; počas druhej

fázy sa zlučujú behy z pások a, b na súbor c. Tento proces je znázornený na *obr. 2.13*.



Obr. 2.13. Fázy a prechody triedenia

Príklad triedenia prirodzeným zlučovaním *Tabuľka 2.11*

17	31'	5	59'	13	41	43	67'	11	23	29	47'	3	7	71'	2	19	57'	37	61
5	17	31	59'	11	13	23	29	41	43	47	67'	2	3	7	19	57	71'	37	61
5	11	13	17	23	29	31	41	43	47	59	67'	2	3	7	19	37	57	61	71
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	57	59	61	67	71

V *tab. 2.11* je uvedený príklad triedenia súboru dvadsiatich čísel metódou prirodzeného zlučovania. Prvý riadok tabuľky zobrazuje súbor c v jeho začiatočnom stave, ďalšie riadky reprezentujú tvar po jednotlivých prechodoch algoritmu. Všimnime si, že pri triedení boli potrebné iba tri prechody. Proces triedenia skončí v okamihu, keď súbor c bude obsahovať jediný beh. (Predpokladáme, že vstupný súbor obsahuje aspoň jeden neprázdny beh.) Zavedieme teda premennú l, ktorej hodnotou bude počet behov súboru c. Ak ešte budeme definovať dva globálne objekty

type páska = **file of** prvok; (2.26)
var c: páska

tak prvú verziu algoritmu zapíšeme takto:

```
procedure prirodzené zlučovanie;
   var l: integer;
       a, b: páska;
```

```
begin
  repeat rewrite(a); rewrite(b); reset(c);                    (2.27)
         distribúcia;
         reset(a); reset(b); rewrite(c);
         l: = 0; zlučovanie
  until l = 1
end
```

Dve fázy prirodzeného zlučovania vystupujú v tomto algoritme ako dva oddelené príkazy. Našou ďalšou úlohou bude zjemniť ich, t.j. detailnejšie ich vyjadriť. Môžeme to principiálne dosiahnuť dvoma spôsobmi. Prvým je priama substitúcia detailnejšieho kódu, druhým formálny zápis pomocou procedúr, pričom spomenuté dva príkazy v uvedenom algoritme treba chápať ako volania týchto procedúr. Rozhodneme sa pre druhú metódu a definujeme procedúru distribúcia a procedúru zlučovanie:

```
procedure distribúcia; {z c na a, b}
begin
  repeat skopírujbeh (c, a);                                  (2.28)
        if ¬ eof(c) then skopírujbeh (c, b)
  until eof(c)
end
procedure zlučovanie {z a, b na c}
begin
  repeat zlúčbeh; l: = l + 1
  until eof(b);                                               (2.29)
  if ¬ eof(a) then
  begin skopírujbeh (a, b); l: = l + 1
  end
end
```

Uvedenou distribučnou metódou môžu vzniknúť dve výsledné situácie: Súbory *a*, *b* obsahujú rovnaký počet behov, alebo súbor *a* obsahuje o jeden beh viac ako súbor *b*. Zlučovaním zodpovedajúcich dvojíc behov môže potom nastať prípad, že na súbore *a* zostane jeden beh. Tento posledný beh sa potom jednoducho skopíruje na súbor *c*. Na

vyjadrenie procedúr zlučovanie a distribúcia sme použili zodpovedajúce jednoduchšie procedúry zlúčbeh a skopírujbeh s patričným významom. Tieto procedúry budeme ďalej zjemňovať. Budeme na to potrebovať ďalšiu globálnu boolovskú premennú, ktorú nazveme *eor* a bude nám slúžiť na signalizáciu konca jednotlivých behov.

```
procedure skopírujbeh (var x, y: páska);
begin {skopíruj jeden beh zo súboru x na súbor y}
  repeat skopírujprvok (x, y) until eor                       (2.30)
end

procedure zlúčbeh;
begin {zlúč beh zo súborov a, b na súbor c}
  repeat if a↑.kľúč < b↑.kľúč then
    begin skopírujprvok (a, c);
      if eor then skopírujbeh (b, c)
    end else                                                  (2.31)
    begin skopírujprvok (b, c);
      if eor then skopírujbeh (a, c)
    end
  until eor
end
```

Proces porovnávania a výberu kľúčov pri zlučovaní behov končí v okamihu, keď sa dosiahne koniec jedného zo zlučovaných behov. Zvyšok druhého behu sa potom jednoducho pripojí k novovytvorenému behu pomocou procedúry skopírujbeh. Procedúry zlúčbeh a skopírujbeh využívajú jednoduchú procedúru skopírujprvok, ktorá premiestňuje jeden prvok zo zdrojového súboru *x* na cieľový súbor *y*. Navyše táto procedúra, ktorá je vyjadrená prostredníctvom procedúr read a write, zisťuje, či sa dosiahol koniec príslušného behu alebo nie. Aby sme dokázali zistiť koniec behu, musíme uchovať kľúč posledného prečítaného (zapísaného) prvku na porovnanie s jeho nasledovníkom. Takýto „pohľad dopredu“ sa dosiahne jednoduchým preskúmaním hodnoty vyrovnávacej premennej $x\uparrow$ združenej so súborom *x*.

```
procedure skopírujprvok (var x, y: páska);
  var buf: prvok;
```

```
begin read (x, buf); write (y, buf);                    (2.32)
    if eof (x) then eor := true
               else eor := buf.kľúč > x↑.kľúč
end
```

Týmto sme zavŕšili proces tvorby algoritmu triedenia prirodzeným zlučovaním. Žiaľ, program nie je správny, čo si pozorný čitateľ už iste všimol. Nesprávny je v tom zmysle, že pre niektoré prípady je výsledkom triedenia nesprávne usporiadaný súbor. Uvažujme napríklad o nasledujúcej postupnosti vstupných údajov:

3 2 5 11 7 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67

Distribúciou jednotlivých behov striedavo na súbory *a*, *b* dostaneme:

$a = 3'\ 7\ 13\ \ 19'\ 29\ 37\ \ 43'\ 57\ 61\ \ 71'$

$b = 2\ \ 5\ 11'\ 17\ \ 23\ 31'\ 41\ \ 47\ 59'\ 67$

Tieto postupnosti sa ľahko zlúčia do jedného behu a potom sa triedenie úspešne ukončí. Aj keď tento príklad nevedie k chybnému správaniu sa programu, núti nás, aby sme si uvedomili, že sama distribúcia behov na niekoľko súborov môže mať za následok menší počet výstupných behov, ako je počet vstupných behov. Je to preto, že pravý prvok $(i + 2)$-ého behu môže byť väčší než posledný prvok i-tého behu, čím sa dosiahne automatické zlúčenie týchto dvoch behov do jedného behu.

Aj keď procedúra distribúcia, ako sa dalo predpokladať, delí výsledné behy rovnomerne na dva súbory, skutočný počet výsledných behov na súboroch *a* a *b* sa môže významne líšiť. Naša procedúra zlučovania bude však zlučovať iba dvojice behov. Skončí sa vtedy, keď sa prečíta súbor *b*, čím sa stratí koniec jedného zo súborov. Majme nasledujúce vstupné údaje, ktoré sú utriedené (a skrátené) vo dvoch po sebe idúcich prechodoch (*tab. 2.12*). Takáto chyba v programovaní je typická pre mnohé situácie v programovaní. Je spôsobená prehliadnutím jedného

Tabuľka 2.12

Nesprávny výsledok triedenia prirodzeným zlučovaním

17	19	13	57	23	29	11	59	31	37	7	61	41	43	5	67	47	71	2	3
13	17	19	23	29	31	37	41	43	47	57	71	11	59						
11	13	17	19	23	29	31	37	41	43	47	57	59	71						

z možných dôsledkov pravdepodobne jednoduchej operácie. Je typická aj v tom zmysle, že je možných niekoľko spôsobov opravy chyby a že sa musí vybrať jeden z nich. Často sa vyskytujú dve možnosti, ktoré sa podstatne líšia:

1. Pripustíme, že operácia distribúcie je nesprávne naprogramovaná a nespĺňa požiadavku, aby boli počty behov rovnaké (alebo sa líšili nanajvýš o 1). Pridržiavame sa pôvodnej schémy operácie a príslušným spôsobom opravujeme chybnú procedúru.

2. Pripusťme, že oprava chybnej časti má za následok ďalekosiahle modifikácie a pokúsime sa nájsť spôsoby, ktorými sa niektoré časti algoritmu môžu zmeniť, aby sa prispôsobili momentálne nesprávnej časti algoritmu.

Zdá sa, že prvý spôsob je vcelku bezpečnejší, jasnejší, priamejší a poskytuje dostatočnú mieru imunity pred neskoršími dôsledkami prehliadnutých komplikovaných vedľajších účinkov. Tento spôsob je preto cestou k riešeniu, ktoré sa všeobecne (a oprávnene) odporúča.

Treba však zdôrazniť, že druhú alternatívu by sme nemali celkom ignorovať. Preto v ďalšom kroku tvorby programu triedenia zlučovaním uvedieme modifikáciu procedúry zlučovania. Procedúru distribúcie behov, ktorá býva najčastejšie chybne naprogramovaná, ponecháme nezmenenú.

To znamená, že schému distribúcie nechávame nedotknutú a zriekame sa podmienky rovnomerného rozdelenia behov. Tým môže dôjsť k menej optimálnemu algoritmu. Pre najhorší prípad sa však účinnosť algoritmu nemení, navyše prípad veľmi nerovnomernej distribúcie je aj štatisticky značne nepravdepodobný. Preto ani parametre efektívnosti algoritmu nie sú závažnými argumentmi proti tomuto riešeniu.

Ak už neplatí podmienka rovnomernej distribúcie behov, tak treba meniť procedúru zlučovania v tom zmysle, že len čo sa dosiahne koniec jedného zo súborov, zvyšok druhého súboru sa celý skopíruje na vytváraný súbor (namiesto kopírovania iba jedného behu).

Táto zmena je v porovnaní s ľubovoľnou zmenou v distribučnej procedúre priamočiara a veľmi jednoduchá. (Čitateľ sa môže sám presvedčiť o pravdivosti tohto tvrdenia.) Revidovaná verzia zlučovacieho algoritmu je obsiahnutá v kompletnom programe 2.14.

PROGRAM 2.14. *Triedenie prirodzeným zlučovaním*

```
program Triedeniezlučovaním (input, output);
   {3-páskové, 2-fázové triedenie prirodzeným zlučovaním}
type prvok = record kľúč: integer;
                          {definícia ostatných zložiek}
             end;
     páska = file of prvok;
var c: páska; n: integer; buf: prvok;
procedure list (var f: páska);
   var x: prvok;
begin reset (f);
   while ¬ eof(f) do
      begin read(f, x); write(output, x.kľúč)
      end;
   writeln
end {list};
procedure prirodzenézlučovanie;
   var l: integer; {počet zlučovaných behov}
       eor: boolean; {indikátor konca behu}
       a, b: páska;
procedure skopírujprvok (var x, y: páska);
   var buf: prvok;
begin read(x, buf); write(y, buf);
      if eof(x) then eor: = true
                else eor: = buf.kľúč > x↑.kľúč
end;
procedure skopírujbeh (var x, y: páska);
begin {skopírovanie jedného behu zo súboru x na súbor y}
   repeat skopírujprvok (x, y) until eor
end;
procedure distribúcia;
begin {zo súboru c na súbory a, b}
   repeat skopírujben (c, a);
      if ¬ eof(c) then skopírujbeh (c, b)
   until eof(c)
end;
procedure zlúčbeh;
begin {zo súborov a, b na súbor c}
   repeat
      if a↑.kľúč < b↑.kľúč then
      begin skopírujprvok (a, c);
         if eor then skopírujbeh (b, c)
      end else
      begin skopírujprvok (b, c);
            if eor then skopírujbeh (a, c)
      end
   until eor
end;
procedure zlučovanie;
begin {zo súborov a, b na súbor c}
   while ¬ eof(a) ∧ ¬ eof(b) do
   begin zlúčbeh; l: = l + 1
   end;
   while ¬ eof(a) do
   begin skopírujbeh (a, c); l: = l + 1
   end;
   while ¬ eof(b) do
   begin skopírujbeh (b, c); l: = l + 1
   end;
      list(c)
   end;
begin
   repeat rewrite(a); rewrite(b); reset(c);
         distribúcia;
         reset(a); reset(b); rewrite(c);
         l: = 0; zlučovanie
   until l = 1
end;
begin {hlavný program; číta vstupné údaje končiace nulou}
   rewrite(c); read(buf.kľúč);
   repeat write(c.buf); read(buf.kľúč)
   until buf.kľúč = 0;
```

list (c);
prirodzenézlučovanie;
list (c)
end.

2.3.3 VYVÁŽENÉ VIACCESTNÉ ZLUČOVANIE

Zložitosť algoritmov vonkajšieho triedenia je úmerná počtu potrebných prechodov, pretože, podľa definície každý prechod zahŕňa kopírovanie celej množiny údajov. Jednou z možných ciest redukcie počtu prechodov je distribúcia behov na viac ako dva súbory. Zlučovaním r behov, ktoré sú rovnomerne distribuované na N pások, vznikne postupnosť r/N behov. Druhý prechod redukuje tento počet na r/N^2, tretí na r/N^3 a k-tý na r/N^k. Celkový počet prechodov potrebných na utriedenie n prvkov metódou N-cestného zlučovania bude potom $k = \lceil \log_N n \rceil$. Pretože každý prechod vyžaduje n operácií kopírovania, celkový počet týchto operácií bude v najhoršom prípade

$$M = n \, . \, \lceil \log_N n \rceil$$

V ďalšom budeme vyvíjať program triedenia založený na viaccestnom zlučovaní. Na rozdiel od predchádzajúceho algoritmu prirodzeného dvojfázového zlučovania budeme viacfázové zlučovanie formulovať ako jednofázové vyvážené triedenie zlučovaním. To znamená, že v každom prechode je rovnaký počet vstupných a výstupných súborov, na ktoré sa po sebe nasledujúce behy striedavo umiestňujú. Ak teda použijeme N súborov, predpokladáme pritom, že N je párne, bude algoritmus založený na $N/2$-cestnom zlučovaní. Vzhľadom na prijatú stratégiu nebudeme zisťovať automatické zlučovanie dvoch po sebe idúcich behov umiestnených na tej istej páske. Preto navrhneme zlučovací program bez predpokladu rovnakého počtu behov na vstupných páskach.

V tomto programe sa prvý raz objavuje prirodzená aplikácia štruktúry údajov, pozostávajúcej z poľa súborov. Je skutočne prekvapujúce, ako sa nasledujúci program vplyvom zmeny zlučovania z dvojfázového na viacfázové výrazne odlišuje od predchádzajúceho programu. Zmena je predovšetkým dôsledkom skutočnosti, že zlučovací proces nemožno

jednoducho ukončiť v okamiku, keď sa zistí koniec jedného zo vstupných behov. Namiesto toho budeme udržiavať zoznam vstupných súborov, ktoré sú ešte aktívne, t.j. nachádzajú sa v nich ešte nejaké nezlúčené behy. Ďalšia komplikácia vzniká z nevyhnutnosti prepinania vstupných a výstupných pások po každom prechode. Začneme s definíciou ďalšieho typu údajov, ktorý je dodatkom k dvom príbuzným typom prvok a páska:

$$\text{číslopásky} = 1 \,..\, n \qquad (2.33)$$

Obyčajne sa čísla pások využívajú na indexovanie poľa súborov položiek. Predpokladajme, že vstupná postupnosť prvokov je reprezentovaná premennou

$$f0\text{: páska} \qquad (2.34)$$

a že proces triedenia má k dispozícii n pások, pričom n je párne:

$$f\text{: } \textbf{array} \text{ [číslopásky] } \textbf{of} \text{ páska} \qquad (2.35)$$

Odporúčanou technikou, riešiacou problém prepínania pások, je zavedenie mapy indexov pások. Potom namiesto priameho adresovania konkrétnej pásky pomocou jej indexu ju adresujeme prostredníctvom mapy t, t.j. namiesto každého

$$f[i] \quad \text{píšeme} \quad f[t[i]]$$

pričom mapa je definovaná ako

$$t\text{: } \textbf{array} \text{ [číslopásky] } \textbf{of} \text{ číslopásky} \qquad (2.36)$$

Ak sa budú začiatočné hodnoty prvkov mapy rovnať indexom i, t.j. $t[i] = i$ pre všetky i, tak prepnutie pásky bude pozostávať iba z výmeny zodpovedajúcich dvojíc prvkov mapy

$$t[1] \leftrightarrow t[nh + 1]$$
$$t[2] \leftrightarrow t[nh + 2]$$
$$\dots$$
$$t[nh] \leftrightarrow t[n]$$

kde $nh = n/2$. Postupnosť

$$f[t[1]], \dots, f[t[nh]]$$

potom môžeme stále považovať za postupnosť vstupných pások a postupnosť

$$f[t[nh+1]], \ldots, f[t[n]]$$

za postupnosť výstupných pások. (V dôsledku toho budeme namiesto výrazu $f[t[j]]$ označujúceho j-tú pásku jednoducho písať j-tá páska.) Na základe doterajších úvah a definícií môžeme teraz pristúpiť k formulácii prvej verzie algoritmu vyváženého viaccestného zlučovania:

```
procedure páskovézlučovanie;
  var i, j: číslopásky;
      l: integer; {počet distribuovaných behov}
      t: array [číslopásky] of číslopásky;                    (2.37)
begin {distribúcia začiatočných behov na pásky t[1] ... t[nh]}
  j: = nh; l: = 0;
  repeat if j < nh then j: = j + 1 else j: = 1;
        „presun jedného behu z f0 na j-tú pásku";
        l: = l + 1
  until eof(f0);
  for i: = 1 to n do t[i]: = i;
  repeat {zlučovanie z t[1] ... t[nh] na t[nh + 1] ... t[n]}
        „nastav vstupné pásky";
        l: = 0;
        j: = nh + 1; {j je index výstupnej pásky}
        repeat l: = l + 1;
              „zlúč beh zo vstupných pások na výstupnú
              pásku t[j]"
           if j < n then j: = j + 1 else j: = nh + 1
        until „prečítali sa všetky vstupné pásky";
        „prepnutie pások"
  until l = 1;
  {utriedené údaje sú na páske t[1]}
end
```

Najskôr zjemníme operáciu kopírovania použitú v začiatočnej distribúcii behov. Použijeme na to pomocnú premennú *buf*, ktorá bude obsahovať posledný prečítaný prvok

```
buf: prvok
```

a nahradíme príkaz „presun jedného behu z $f0$ na i-tú pásku" príkazom

```
repeat read (f0, buf);
       write (f[j], buf)                                      (2.38)
until (buf.kľúč > f0↑.kľúč) ∨ eof(f0)
```

Proces kopírovania behov končí buď pri výskyte prvého prvku ďalšieho behu (*buf*.*kľúč* > $f0$↑.*kľúč*), alebo pri prečítaní posledného prvku vstupného súboru (eof($f0$)).

V uvedenom algoritme musíme ďalej detailnejšie špecifikovať príkazy

1. „Nastav vstupné pásky"
2. „Zlúč beh zo vstupných pások na výstupnú pásku $t[j]$"
3. „Prepnutie pások"

a predikát

4. „Prečítali sa všetky vstupné pásky"

Predovšetkým musíme presnejšie identifikovať momentálne vstupné súbory. Všimnime si, že počet aktívnych vstupných súborov môže byť menší ako $n/2$. V skutočnosti môže byť maximálne toľko zdrojov, koľko je behov. Triedenie končí, keď ostáva jediný súbor. Tým môže dôjsť k situácii, že na začiatku posledného prechodu triedenia bude menej ako nh behov. Preto zavádzame premennú $k1$, ktorou budeme označovať aktuálny počet vstupných súborov. Inicializáciu premennej $k1$ zaradíme do príkazu „nastav vstupné pásky" nasledujúcim spôsobom:

```
if l < nh then k1: = l else k1: = nh;
for i: = 1 to k1 do reset(f[t[i]]);
```

Pochopiteľne, v príkaze 2. z uvedených troch príkazov, sa znižuje hodnota premennej $k1$, a to vždy vtedy, keď sa prečíta koniec jedného zo súborov. Predikát 4 potom môžeme jednoducho vyjadriť reláciou:

$$k1 = 0$$

Príkaz 2 sa nedá zjemniť tak jednoducho. Pozostáva z opakovaného výberu najmenšieho kľúča spomedzi dostupných vstupných súborov a jeho premiestnenia do cieľového súboru, t. j. na momentálnu výstup-

nú pásku. Tento proces je opäť sťažený nevyhnutnosťou určenia konca každého behu. Koniec behu nastáva v okamihu, keď

a) nasledujúci kľúč má menšiu hodnotu, ako je hodnota momentálneho kľúča, alebo

b) len čo sa prečíta posledný prvok zdrojového súboru.

V prípade b) sa páska vylúči z ďalšieho spracúvania jednoduchým znížením hodnoty premennej $k1$; v prípade a) sa beh ukončí tým, že príslušný súbor ignorujeme pri výbere prvkov až do konca momentálneho prechodu, t.j. dovtedy, kým nie je momentálny výstupný beh skompletizovaný. Je pochopiteľné, že potom potrebujeme druhú pomocnú premennú, ktorú označíme symbolom $k2$ a ktorá bude označovať počet vstupných pások aktuálnych pre výber ďalšej položky. Hodnota tejto premennej je inicializovaná hodnotou premennej $k1$ a znižuje sa zakaždým, keď sa dosiahne koniec behu (podľa 1. podmienky). Žiaľ, zavedenie premennej $k2$ ešte nie je dostačujúce, pretože znalosť počtu pások nám nestačí. Potrebujeme presne vedieť, ktoré pásky sú ešte aktívne. Prirodzeným riešením tohto problému je použitie poľa boolovských prvkov označujúcich aktívnosť pások. Rozhodneme sa však pre iné riešenie, ktoré speje k efektívnejšej procedúre výberu. Procedúra výberu položiek predstavuje najčastejšie sa opakujúcu časť celého triediaceho algoritmu. Namiesto boolovského poľa potom použijeme ďalšiu mapu pások, ktorú označíme symbolom *ta*. Túto mapu budeme používať namiesto premennej *t*, t.j. výrazmi *ta*[1] ... *ta*[$k2$] vyjadrujeme indexy pások, ktoré sú ešte aktívne. Na základe uvedeného môžeme príkaz 2 formulovať takto:

```
k2: = k1;
repeat „vyber najmenší kľúč; nech ta[mx] je číslo jeho pásky";
       read (f[ta[mx]], buf);
       write (f[t[j]], buf);                                        (2.39)
       if eof (f[ta[mx]]) then „vylúč pásku" else
       if buf.kľúč > f[ta[mx]]↑.kľúč then „ukonči beh"
until k2 = 0
```

Vzhľadom na to, že počet magnetickopáskových zariadení býva v každom počítačovom prostredí obyčajne malý, algoritmus výberu, ktorý treba v ďalšom kroku zjemnenia podrobnejšie špecifikovať, môže

byť realizovaný prostredníctvom obyčajného lineárneho vyhľadávania. Príkaz „vylúč pásku" zahŕňa zníženie hodnôt premenných $k1$ a $k2$. Navyše tento príkaz spôsobí aktualizáciu indexov pások v mape *ta*. Príkaz „ukonči beh" spôsobí zmenšenie hodnoty premennej $k2$ a primeranú aktualizáciu prvkov mapy *ta*. Uvedené myšlienky sú podrobnejšie realizované v programe 2.15, ktorý je súčasne posledným zjemnením programov 2.37 až 2.39.

PROGRAM 2.15. *Triedenie vyváženým zlučovaním*

```
program Vyváženézlučovanie (output);
  {vyvážené n-cestné páskové triedenie zlučovaním}
const n = 6; nh = 3; {počet pások}
type prvok = record
                kľúč: integer
             end;
   páska = file of prvok;
   číslopásky = 1..n;
var leng, rand: integer {premenné potrebné pri vytváraní
                         vstupného súboru}
   eot: boolean;       {koniec pásky}
   buf: prvok;
   f0: páska; {f0 je vstupná páska obsahujúca pseudonáhodné čísla}
   f: array [1..n] of páska;
procedure list (var f: páska; n: číslopásky);
  var z: integer;
begin writeln ('PÁSKA', n: 2); z: = 0;
  while ¬ eof (f) do
  begin read (f, buf); write (output, buf.kľúč: 5); z: = z + 1;
    if z = 25 then
    begin writeln (output); z: = 0;
    end
  end;
  if z ≠ 0 then writeln (output); reset (f)
end {list};
procedure páskovézlučovanie;
```

```
  var i, j, mx, tx: číslopásky;
      k1, k2, l: integer;
      x, min: integer;
      t, ta: array [číslopásky] of číslopásky;
begin {distribúcia začiatočných behov na pásky t[1] ... t[nh]}
  for i: = 1 to nh do rewrite(f[i]);
  j: = nh; l: = 0;
  repeat if j < nh then j: = j + 1 else j: = 1;
    {presun jedného behu z f0 na j-tú pásku}
    l: = l + 1;
    repeat read(f0, buf); write(f[j], buf)
    until (buf. kľúč > f0↑. kľúč) ∨ eof(f0)
  until eof(f0);
  for i: = 1 to n do t[i]: = i;
  repeat {zlučovanie z t[1] ... t[nh] na t[nh + 1] ... t[n]}
    if l < nh then k1: = l else k1: = nh;
    {k1 je počet vstupných pások v rámci tejto fázy}
    for i: = 1 to k1 do
    begin reset(f[t[i]]); list(f[t[i]], t[i]); ta[i]: = t[i]
    end;
    l: = 0; {l je počet zlučovaných behov}
    j: = nh + 1; {j je index výstupnej pásky}
    repeat {zlúč beh z pások t[1] ... t[k1] na t[j]}
      k2: = k1; l: = l + 1; {k2 je počet aktívnych vstupných pások}
      repeat {vyber najmenší prvok}
        i: = 1; mx: = 1; min: = f[ta[1]]↑. kľúč;
        while i < k2 do
        begin i: = i + 1; x: = f[ta[i]]↑. kľúč;
              if x < min then
              begin min: = x; mx: = i
              end
        end;
        {ta[mx] obsahuje najmenší prvok; presuň ho na pásku t[j]}
        read(f[ta[mx]], buf); eot: = eof(f[ta[mx]]);
        write(f[t[j]], buf);
        if eot then
```



```
        begin rewrite(f[ta[mx]]); {vylúč pásku}
          ta[mx]: = ta[k2]; ta[k2]: = ta[k1];
          k1: = k1 − 1; k2: = k2 − 1
        end else
          if buf. kľúč > f[ta[mx]]↑. kľúč then
          begin tx: = ta[mx]; ta[mx]: = ta[k2];
                ta[k2]: = tx; k2: = k2 − 1
          end
      until k2 = 0;
      if j < n then j: = j + 1 else j: = nh + 1
    until k1 = 0;
    for i: = 1 to nh do
      begin tx: = t[i]; t[i]: = t[i + nh]; t[i + nh]: = tx
      end
  until l = 1;
  reset(f[t[1]]); list(f[t[1]], t[1]); {utriedené údaje sú na páske t[1]}
end {páskovézlučovanie};
begin {vygenerovanie súboru pseudonáhodných čísel f0}
  leng: = 200; rand: = 7789; rewrite(f0);
  repeat rand: = (131071 * rand) mod 2147483647;
         buf. kľúč: = rand div 2147484; write(f0, buf);
         leng: = leng − 1
  until leng = 0;
  reset(f0); list(f0, 1);
  páskovézlučovanie
end.
```

Všimnime si, že pásky sa previnú prostredníctvom procedúry rewrite, len čo sa prečítal ich posledný beh. Príkaz „prepnutie pások" sme implementovali na základe predchádzajúcich úvah.

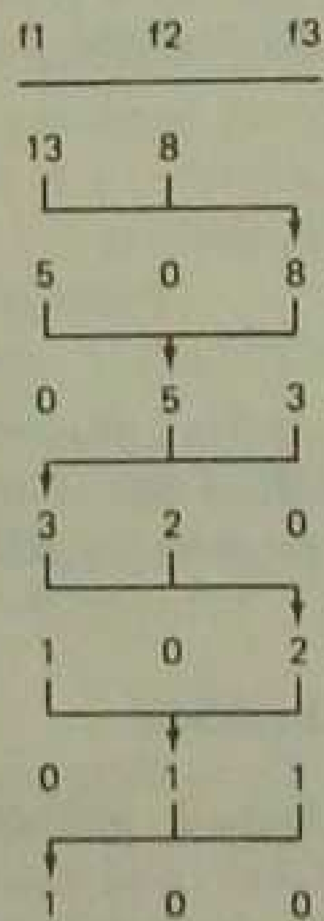
2.3.4 POLYFÁZOVÉ TRIEDENIE

Preskúmaním dôležitých techník, aplikovaných v rámci algoritmov vonkajšieho triedenia, sme získali solídne vedomosti pre návrh a implementáciu ďalšieho triediaceho algoritmu, ktorého parametre sú lepšie

vzhľadom na vyvážené triedenie. V predchádzajúcom algoritme sme videli, že vyvážené zlučovanie vylučuje čisto kopírovacie operácie potrebné v prípade spojenia distribúcie a zlučovania do jednej fázy.

Otázkou je, či by sa pásky nedali využiť lepšie. Preskúmanie tohto problému bude našou ďalšou úlohou. Kľúč na jej vyriešenie spočíva v zrieknutí sa prísneho dodržiavania presného počtu prechodov. Tým sa nám naskytá možnosť lepšieho využívania pások, pretože, ako sa ukáže neskôr, nebudeme potrebovať vždy $N/2$ zdrojových a $N/2$ cieľových pások, ktoré sme na konci každého prechodu prácne zamieňali. Pojem prechod sa týmto stáva „difúznym". Metóda triedenia, o ktorej sa podrobne zmienime v tomto odseku, pochádza od R. L. GILSTADA [2-3] a nazval ju polyfázové triedenie. Objasnime si ju na dvoch príkladoch. V prvom sú použité tri pásky. V každom prechode sa prvky zlučujú z dvoch pások na tretiu. Len čo sa dosiahne koniec jednej zo vstupných pások, zmení sa orientácia tejto pásky na výstupnú a triediaci proces pokračuje zlučovaním behov z pôvodnej cieľovej a zostávajúcej aktívnej pásky.

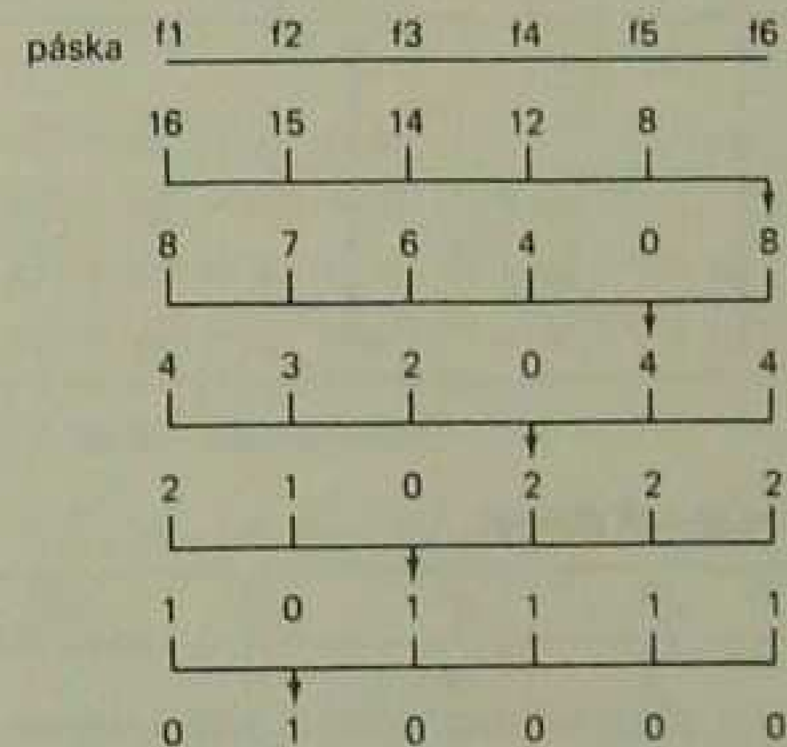
Pretože vieme, že n behov z každej vstupnej pásky sa premení na n behov výstupnej pásky, musíme udržiavať iba počet behov na každej



Obr. 2.14. Trojpáskové polyfázové triedenie 21 behov

páske (namiesto špecifikovania aktuálnych kľúčov). Na *obr. 2.14* je znázornený prvý príklad polyfázového triedenia. Predpokladáme, že dve začiatočné vstupné pásky $f1$ a $f2$ obsahujú 13 a 8 behov. Potom sa v prvom prechode zlúči 8 behov z pások $f1$ a $f2$ na pásku $f3$ a v druhom prechode sa zlúči zostávajúcich 5 behov z pások $f3$ a $f1$ na pásku $f2$ atď. Po poslednom prechode sa utriedený súbor bude nachádzať na páske $f1$.

V druhom príklade polyfázového triedenia použijeme 6 pások. Nech sú začiatočné behy rozmiestnené na jednotlivých páskach takto: 16 behov je na páske $f1$, 15 na $f2$, 14 na $f3$, 12 na $f4$ a 8 na $f5$. Prvým prechodom sa zlúči 8 behov z každej pásky na pásku $f6$ atď. Na konci triedenia bude páska $f2$ obsahovať utriedený súbor prvkov (*obr. 2.15*).



Obr. 2.15. Šesťpáskové polyfázové triedenie 55 behov

Polyfázové triedenie je oveľa efektívnejšie ako vyvážené zlučovanie, pretože pre N pások algoritmus pracuje vždy ako $(N-1)$-cestné zlučovanie, namiesto $N/2$-cestného zlučovania pri vyváženom zlučovaní. Keďže počet potrebných prechodov je približne $\log_N n$, kde n je počet triedených prvkov a N stupeň zlučovacích operácií, polyfázovým triedením dosiahneme výrazné zlepšenie oproti vyváženému triedeniu.

Pochopiteľne, že distribúcia začiatočných behov v uvedených príkla-

doch bola starostlivo vybratá. Aby sme dokázali zistiť najvýhodnejšiu distribúciu začiatočných behov, začneme postupovať odzadu, t.j. od posledného riadku *obr. 2.15*. Pre uvedené príklady vytvoríme dve tabuľky, v ktorých bude každý riadok obsahovať údaje o počte behov na jednotlivých páskach. Každý nasledujúci riadok tabuľky bude mať tieto údaje pootočené o jednu pozíciu vzhľadom na prechádzajúci riadok (vynecháme nuly, t.j. výstupné pásky). Takto dostaneme *tab. 2.13* a *2.14*, ktoré zobrazujú 6 prechodov trojpáskového a šesťpáskového polyfázového triedenia.

Ideálna distribúcia behov na dvoch páskach *Tabuľka 2.13*

l	$a_1^{(l)}$	$a_2^{(l)}$	$\Sigma a_i^{(l)}$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

Ideálna distribúcia behov na piatich páskach *Tabuľka 2.14*

l	$a_1^{(l)}$	$a_2^{(l)}$	$a_3^{(l)}$	$a_4^{(l)}$	$a_5^{(l)}$	$\Sigma a_i^{(l)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Z *tab. 2.13* môžeme odvodiť tieto vzťahy:

$$\left.\begin{aligned} a_2^{(l+1)} &= a_1^{(l)} \\ a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} \end{aligned}\right\} \quad \text{pre} \quad l > 0 \tag{2.40}$$

a $a_1^{(0)} = 1$, $a_2^{(0)} = 0$. Nech $a_1^{(l)} = f_{l+1}$. Potom dostávame

$$\begin{aligned} f_{i+1} &= f_i + f_{i-1} \quad \text{pre} \quad i \geqq 1 \\ f_1 &= 1 \\ f_0 &= 0 \end{aligned} \tag{2.41}$$

Uvedené vzťahy predstavujú rekurzívne pravidlá (alebo rekurentné vzťahy), ktoré definujú Fibonacciho čísla:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

Každé Fibonacciho číslo je súčtom svojich dvoch predchodcov. Na základe toho sa musia počty začiatočných behov na dvoch páskach rovnať dvom nasledujúcim Fibonacciho číslam. Potom bude trojpáskový polyfázový algoritmus triedenia pracovať efektívnym spôsobom.

Ako to bude v druhom príklade (*tab. 2.14*), kde sme mali 6 pások? Ľahko odvodíme nasledujúce vytvárajúce pravidlá:

$$\begin{aligned} a_5^{(l+1)} &= a_1^{(l)} \\ a_4^{(l+1)} &= a_1^{(l)} + a_5^{(l)} = a_1^{(l)} + a_1^{(l-1)} \\ a_3^{(l+1)} &= a_1^{(l)} + a_4^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} \\ a_2^{(l+1)} &= a_1^{(l)} + a_3^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)} \\ a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)} + a_1^{(l-4)} \end{aligned} \tag{2.42}$$

Substitúciou f_i za $a_1^{(i)}$ dostávame

$$\begin{aligned} f_{i+1} &= f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4} \quad \text{pre} \quad i \geqq 4 \\ f_4 &= 1 \\ f_i &= 0 \quad \text{pre} \quad i < 4 \end{aligned} \tag{2.43}$$

Uvedené čísla predstavujú Fibonacciho čísla štvrtého rádu. Vo všeobecnosti Fibonacciho čísla p-tého rádu sú definované takto:

$$\begin{aligned} f_{i+1}^{(p)} &= f_i^{(p)} + f_{i-1}^{(p)} + \ldots + f_{i-p}^{(p)} \quad \text{pre} \quad i \geqq p \\ f_p^{(p)} &= 1 \\ f_i^{(p)} &= 0 \quad \text{pre} \quad 0 \leqq i \leqq p \end{aligned} \tag{2.44}$$

Poznamenávame, že obyčajné Fibonacciho čísla sú prvého rádu. Ak teda chceme, aby n-páskový polyfázový algoritmus triedenia pracoval ideálne, musí sa začiatočný počet behov na páskach rovnať súčtu $n-1$, $n-2$, ..., 1 (pozri *tab. 2.15*) za sebou idúcich Fibonacciho čísel $(n-2)$-

-ého rádu. Z toho automaticky vyplýva, že táto metóda triedenia je použiteľná iba v tých prípadoch, keď sa počet behov na vstupe rovná súčtu $n - 1$ takýchto Fibonacciho súčtov. Tým, samozrejme, vzniká otázka: Čo robiť v prípade, ak sa počet začiatočných behov nerovná takémuto ideálnemu číslu? Odpoveď je jednoduchá (a pre takéto prípady typická): Počet behov musíme upraviť tak, aby sa rovnal ideálnemu počtu. To sa dá jednoducho uskutočniť nasimulovaním existencie hypotetických prázdnych behov, ktoré spolu so skutočnými vytvoria ideálny počet. Tieto prázdne behy nazveme *fiktívnymi*. Takáto odpoveď ešte stále nie je uspokojujúca, pretože okamžite vzniká ďalšia a ťažšia otázka: Ako dokážeme počas zlučovania rozoznať fiktívne behy od skutočných? Predtým, ako nájdeme odpoveď na túto otázku, musíme ešte vyriešiť predošlý problém distribúcie začiatočných behov a rozhodnúť o pravidle distribuovania skutočných a fiktívnych behov na $n - 1$ pások.

Aby sme navrhli primerané distribučné pravidlo, musíme vedieť, akým spôsobom sa zlučujú skutočné a fiktívne behy. Pochopiteľne, že výber fiktívneho behu z i-tej pásky znamená presne to, že táto páska sa v rámci bežného kroku zlučovania ignoruje, čím sa proces zlučovania vzťahuje na menej než na $n - 1$ vstupných pások. Zlučovanie fiktívnych behov zo všetkých $n - 1$ vstupných pások potom znamená, že sa nevykoná žiadna operácia zlučovania, ale na výstupnú pásku sa zapíše jeden výsledný fiktívny beh. Z toho vyplýva, že fiktívne behy treba distribuovať pokiaľ možno čo najrovnomernejšie na $n - 1$ pások, pretože nás zaujímajú aktívne zlučovania z najväčšieho možného počtu vstupných pások.

Nechajme na chvíľu fiktívne behy a venujme sa problému distribúcie neznámeho počtu behov na $n - 1$ pások. Je jasné, že Fibonacciho čísla $(n - 2)$-ho rádu, ktoré špecifikujú požadované počty behov na každej páske, sa dajú generovať počas distribučnej fázy. Predpokladajme napr. že máme k dispozícii 6 pások ($n = 6$). V súlade s *tab. 2.14* začnime s distribúciou behov, najprv podľa riadku označeného indexom $l = 1$ (1, 1, 1, 1, 1); ak zostali nejaké behy, pokračujeme podľa druhého riadku (2, 2, 2, 2, 1); ak je zdrojový súbor ešte stále neprázdny, prebehne ďalšia distribúcia podľa tretieho riadku (4, 4, 4, 3, 2) atď. Riadkový index budeme nazývať úrovňou. Čím väčší je počet behov, tým vyššia

bude úroveň Fibonacciho čísel, ktorá sa náhodou rovná počtu zlučovacích prechodov alebo aj prepnutí pások potrebných v nasledujúcej fáze zlučovania.

Prvú verziu algoritmu distribúcie môžeme formulovať takto:

1. Nech sú cieľom distribúcie Fibonacciho čísla $(n - 2)$-ého rádu prvej úrovne.

2. Vykonajme distribúcie v súlade s týmto cieľom.

3. Ak sme dosiahli stanovený cieľ, vypočítame ďalšiu úroveň Fibonacciho čísel. Rozdiel medzi nimi a číslami na prvej úrovni tvorí nový cieľ distribúcie. Vráťme sa ku kroku 2. Ak sme dospeli ku koncu vstupného súboru a súčasne nedosiahli stanovený cieľ, sme na konci distribučného procesu.

Pravidlá na výpočet ďalšej úrovne Fibonacciho čísel sú obsiahnuté v ich definícii (2.44). V dôsledku toho sa môžeme sústrediť na druhý krok distribučného algoritmu, kde v súlade so stanoveným cieľom treba jednotlivé behy umiestniť jeden po druhom na $n - 1$ pások.

Teraz sa znova začneme zaoberať fiktívnymi behmi. Predpokladajme, že so zvyšovaním úrovne určujeme nasledujúci cieľ pomocou rozdielov d_i pre $i = 1, \dots, n - 1$, kde d_i znamená počet behov, ktoré sa majú umiestniť v tomto kroku na i-tú pásku. Predpokladajme ďalej, že na i-tú pásku okamžite umiestnime d_i fiktívnych behov. Tým môžeme nasledujúcu distribúciu chápať ako nahradzovanie fiktívnych behov skutočnými. Každým nahradením behov znížime hodnotu d_i o jednotku. Tým hodnota d_i vlastne vyjadruje počet fiktívnych behov na i-tej páske, a to v momente, keď dôjde k vyprázdneniu zdrojového súboru.

Nie je známe, ktorý algoritmus prináša optimálnu distribúciu, ale nasledujúci sa považuje za jeden z najlepších. Nazývame ho horizontálna distribúcia (pozri [2-7], s. 270). Tento pojem pochopíme, ak si predstavíme, že behy sú umiestňované na pásky vo forme síl, ako ukazuje *obr. 2.16*, pre $n = 6$ a päť úrovní (porovnaj s *tab. 2.14*). Aby sme čo najrýchlejšie dosiahli rovnomernú distribúciu zostávajúcich fiktívnych behov, realizujeme proces ich nahradzovania skutočnými behmi, ktorý redukuje veľkosť síl tak, že odoberáme fiktívne behy v rámci horizontálnych úrovní smerom zľava doprava. Týmto spôsobom budú behy, označené svojimi poradovými číslami, distribuované na pásky podľa *obr. 2.16*. Teraz už vieme opísať algoritmus pomocou

8	1				
7	2	3	4		
6	5	6	7	8	
5	9	10	11	12	
4	13	14	15	16	17
3	18	19	20	21	22
2	23	24	25	26	27
1	28	29	30	31	32

Obr. 2.16. Horizontálna distribúcia behov

procedúry vyberpásku, ktorá sa volá vždy v okamihu, keď sa určitý beh umiestnil na nejakej páske a treba vybrať pásku na umiestnenie ďalšieho behu. Predpokladajme premennú j, ktorou budeme označovať index momentálnej cieľovej pásky. Symboly a_i a d_i budú označovať ideálny a fiktívny počet distribuovaných behov pre i-tú pásku.

```
j: číslopásky;
a, d: array [číslopásky] of index                    (2.45)
úroveň: integer
```

Tieto premenné budú mať nasledujúce začiatočné hodnoty:

$a_i = 1,\ d_i = 1$ pre $i = 1, ..., n - 1$
$a_n = 0,\ d_n = 0$ (fiktívne)
$j = 1$
$úroveň = 1$

Uvedomme si, že cieľom procedúry vyberpásku je vypočítať hodnoty ďalšieho riadku *tab. 2.14*, t. j. hodnoty $a_1^{(l)} \dots a_{n-1}^{(l)}$, a to vždy vtedy, keď sa zvýši číslo úrovne. Súčasne sa vypočítavajú rozdiely $d_i = a_i^{(l)} - a_i^{(l-1)}$, ktoré predstavujú cieľ ďalšej fázy. Uvažovaný algoritmus je založený na skutočnosti, že výsledné hodnoty d_i klesajú so stúpajúcim indexom (zostupné stupienky na *obr. 2.16*). Poznamenávame, že výnimku tvorí prechod z nultej na prvú úroveň; činnosť tohto algoritmu sa preto začína na prvej úrovni. Procedúra vyberpásku končí svoju činnosť znížením hodnoty premennej d_j o jednotku; táto operácia je ekvivalentná s nahradením fiktívneho behu na j-tej páske skutočným behom:


```
procedure vyberpásku;
  var i: číslopásky; z: integer;
begin
  if d[j] < d[j + 1] then j: = j + 1 else
  begin if d[j] = 0 then
    begin úroveň: = úroveň + 1; z: = a[1];
          for i: = 1 to n - 1 do                      (2.46)
      begin d[i]: = z + a[i + 1] - a[i];
            a[i]: = z + a[i + 1]
      end
    end;
    j: = 1
  end;
  d[j]: = d[j] - 1
end
```

Tým, že predpokladáme existenciu procedúry na kopírovanie behov zo zdrojového súboru $f0$ na pásku $f[j]$ môžeme formulovať začiatočnú distribučnú fázu takto (vždy predpokladáme, že zdrojový súbor obsahuje aspoň jeden beh):

```
repeat vyberpásku; skopírujbeh                        (2.47)
until eof(f0)
```

Na tomto mieste treba pripomenúť skutočnosť, s ktorou sme sa stretli pri distribúcii behov v rámci algoritmu prirodzeného zlučovania. Ako vieme, za určitých okolností mohla nastať taká situácia, že dva novovytvorené behy, ktoré sa umiestnili na tú istú pásku, mohli splynúť do jedného behu, čím sa stal počet behov, samozrejme, nesprávnym. Implementáciou algoritmu triedenia, ktorého korektnosť bude nezávislá od počtu behov, možno uvedený vedľajší účinok pokojne ignorovať. Pretože v rámci polyfázového triedenia udržujeme údaje o presnom počte behov na každej páske, nemôže sa stať, že by sme si nevšimli vplyv takéhoto náhodného prípadu zlučovania.

V dôsledku toho, že nemôžeme zanedbať výskyt takejto situácie v rámci distribučného procesu, musíme uchovávať kľúče posledných prvkov posledných behov na každej páske. Na tento účel zavádzame premennú

posledná: **array** [číslopásky] **of** integer

Ďalší tvar distribučného algoritmu môžeme vyjadriť takto:

```
repeat vyberpásku
   if posledná [j] ≦ f0↑.kľúč then
      „pokračuj starým behom";                    (2.48)
   skopírujbeh: posledná [j] := f0↑.kľúč
until eof(f0)
```

Pomerne často vzniká chyba tým, že sa zabúda na skutočnosť, že premenná *posledná* [j] nadobúda začiatočnú hodnotu až vtedy, keď sa prečíta prvý beh! Správne riešenie preto spočíva v umiestnení jedného behu na každú z $n - 1$ pások bez ohľadu na to, aby sa skúmala hodnota premennej posledná [j]. Zostávajúce behy sú distribuované podľa algoritmu 2.49:

```
while ¬ eof(f0) do
begin vyberpásku
  if posledná [j] ≦ f0↑.kľúč then
  begin „pokračuj starým behom"                   (2.49)
    skopírujbeh;
    if eof(f0) then d[j] := d[j] + 1 else
                       skopírujbeh
  end
  else skopírujbeh
end
```

Príkaz, ktorým sa priradí hodnota premennej *posledná* [j] je zahrnutý v rámci procedúry skopírujbeh.

Teraz sa už konečne môžeme pustiť do formulovania hlavného algoritmu polyfázového triedenia zlučovaním. Principiálne je štruktúra tohto algoritmu podobná štruktúre algoritmu prirodzeného zlučovania. Zhodné sú najmä tieto aktivity: vonkajší cyklus, v rámci ktorého sa zlučujú behy, pokiaľ sú nejaké pásky aktívne; vnútorný cyklus, kde sa formuluje nový beh zlúčením behov z každej pásky, a najvnútornejší cyklus, ktorým sa vyberie začiatočný kľúč a zodpovedajúci prvok sa prenesie do cieľového súboru. V čom sa líši polyfázové triedenie od prirodzeného zlučovania?

1. V rámci každého prechodu sa používa iba jedna výstupná páska namiesto $n/2$ pri prirodzenom zlučovaní.

2. Namiesto prepínania $n/2$ vstupných a $n/2$ výstupných pások po každom prechode sa pásky rotačným spôsobom vymenia. Toto sa dosiahne pomocou mapy indexov pások, označenej symbolom t.

3. Počet vstupných pások sa mení v každom behu; na začiatku každého behu sa tento počet určí na základe počtu fiktívnych behov d_i. Ak platí $d_i > 0$ pre všetky i, tak sa $n - 1$ fiktívnych behov fiktívne zlúči do jedného fiktívneho behu jednoduchým zvýšením hodnoty d_n pre výstupnú pásku. V opačnom prípade sa zlúči jeden beh z každej pásky s $d_i = 0$, pričom sa hodnota d_i zníži pre všetky ostatné pásky, čím sa naznačuje odobratie fiktívneho behu. Počet vstupných pások zahrnutých do zlučovania označujeme symbolom k.

4. Koniec príslušnej fázy nemožno určiť na základe zistenia konca súboru $(n - 1)$-vej pásky, pretože fiktívne behy z tejto pásky by mohli byť zahrnuté vo viacerých zlučovaniach. Teoreticky sa potrebný počet behov určuje na základe koeficientov a_i. Koeficienty $a_i^{(j)}$ sa vypočítavali počas distribučnej fázy; teraz ich môžeme opätovne vypočítať smerom „odzadu".

Na základe uvedených pravidiel môžeme formulovať hlavný algoritmus polyfázového triedenia. Predpokladáme pritom, že všetky pásky (je ich $n - 1$), na ktorých sa nachádzajú začiatočné behy, sú previnuté na začiatok a mapa pások má začiatočné hodnoty nastavené na $(t_i = i)$.

```
repeat {zlučovanie z pások t[1] ... t[n − 1] na pásku t[n]}
  z := a[n − 1]; d[n] := 0; rewrite (f[t[n]]);
  repeat k := 0; {zlúč jeden beh}
    {určenie počtu k aktívnych vstupných pások}
    for i := 1 to n − 1 do
    if d[i] > 0 then d[i] := d[i] − 1 else
      begin k := k + 1; ta[k] := t[i]
      end;
    if k = 0 then d[n] := d[n] + 1 else                   (2.50)
      „zlúč jeden skutočný beh z pások t[1] ... t[k]";
    z := z − 1
  until z = 0;
```

```
      reset (f[t[n]]);
      „rotácia pások v mape t; výpočet hodnoty a[i] pre
      ďalšiu úroveň":
      rewrite (f[t[n]]); úroveň : = úroveň − 1
until úroveň = 0;
{páska t[1] obsahuje utriedený súbor}
```

Skutočná operácia zlučovania je skoro taká istá ako v prípade n-cestného triedenia zlučovaním. Jediným rozdielom je, že algoritmus vylúčenia pásky je o niečo jednoduchší. Rotácia mapy indexov pások a zodpovedajúcich počtov fiktívnych behov d_i (a nízkoúrovňový prepočet koeficientov a_i) je priamočiara a jej detaily možno vidieť v programe 2.16, ktorý reprezentuje úplný algoritmus polyfázového triedenia.

PROGRAM 2.16. *Polyfázové triedenie*

```
program Polytriedenie (output);
{polyfázové triedenie s n páskami}
const n = 6; {počet pások}
type prvok = record
                kľúč: integer
             end;
    páska = file of prvok;
    číslopásky = 1 . . n;
var leng, rand: integer; {premenné leng a rand sa použijú
                          pri tvorbe vstupného súboru}
    eot: boolean;
    buf: prvok;
    f0: páska; {f0 je vstupná páska s náhodnými číslami}
    f: array [1 . . n] of páska;
procedure list (var f: páska; n: číslopásky);
  var z: integer;
begin z: = 0;
    writeln ('PÁSKA', n: 2);
    while ¬ eof (f) do
    begin read (f, buf); write (output, buf.kľúč: 5); z: = z + 1;
       if z = 25 then
          begin writeln (output); z: = 0
          end
    end;
    if z ≠ 0 then writeln (output); reset (f)
end {list};
procedure polyfázovétriedenie;
  var i, j, mx, tn: číslopásky;
      k, úroveň: integer;
      a, d: array [číslopásky] of integer;
            {a[j] = ideálny počet behov na j-tej páske}
            {d[j] = počet fiktívnych behov na j-tej páske}
      dn, x, min, z: integer;
      posledná: array [číslopásky] of integer;
            {posledná [j] = kľúč posledného prvku na j-tej páske}
      t, ta: array [číslopásky] of číslopásky;
            {mapy čísel pások}
  procedure vyberpásku;
    var i: číslopásky; z: integer;
  begin
    if d[j] < d[j + 1] then j: = j + 1 else
    begin if d[j] = 0 then
        begin úroveň: = úroveň + 1; z: = a[1];
          for i: = 1 to n − 1 do
          begin d[i]: = z + a[i + 1] − a[i]; a[i]: = z + a[i + 1]
          end
        end;
        j: = 1
    end;
    d[j]: = d[j] − 1
  end;
  procedure skopirujbeh;
  begin {skopírovanie jedného behu zo súboru f0 na j-tú pásku}
    repeat read (f0, buf); write (f[j], buf);
    until eof (f0) ∨ (buf.kľúč > f0↑.kľúč);
    posledná [j]: = buf.kľúč
  end;
```

```
begin {distribúcia začiatočných behov}
  for i: = 1 to n − 1 do
    begin a[i]: = 1; d[i] = 1; rewrite (f[i])
    end;
  úroveň: = 1; j: = 1; a[n]: = 0; d[n]: = 0;
  repeat vyber pásku; skopírujbeh
  until eof(f0) ∨ (j = n − 1);
  while ¬ eof(f0) do
  begin vyberpásku;
    if posledná [j] ≦ f0↑.kľúč then
    begin {pokračuj starým behom}
      skopírujbeh;
      if eof(f0) then d[j]: = d[j] + 1 else skopírujbeh
    end
    else skopírujbeh
  end;
  for i: = 1 to n − 1 do reset (f[i]);
  for i: = 1 to n do t[i]: = i;
  repeat {zlučovanie z pások t[1] ... t[n − 1] na pásku t[n]}
    z: = a[n − 1]; d[n]: = 0; rewrite (f[t[n]]);
    repeat k: = 0; {zlúčenie jedného behu}
      for i: = 1 to n − 1 do
        if d[i] > 0 then d[i]: = d[i] − 1 else
          begin k: = k + 1; ta[k]: = t[i]
          end;
        if k = 0 then d[n]: = d[n] + 1 else
        begin {zlúčenie jedného skutočného behu z pások t[1] ... t[k]}
          repeat i: = 1; mx: = 1;
            min: = f[ta[1]]↑.kľúč;
            while i < k do
            begin i: = i + 1; x: = f[ta[i]]↑.kľúč;
              if x < min then
              begin min: = x; mx: = i
              end
            end;
            {ta[mx] obsahuje najmenší prvok;
```



```
            preto ju presunieme na pásku t[n]}
            read (f[ta[mx]], buf); eot: = eof(f[ta[mx]]);
            write (f[t[n]], buf);
            if (buf.kľúč > f[ta[mx]]↑.kľúč) ∨ eot then
            begin {túto pásku ďalej neuvažujeme}
              ta[mx]: = ta[k]; k: = k − 1
            end
          until k = 0
        end;
        z: = z − 1
      until z = 0;
      reset (f[t[n]]); list (f[t[n]], t[n]); {rotácia pások}
      tn: = t[n]; dn: = d[n]; z: = a[n − 1];
      for i: = n downto 2 do
        begin t[i]: = t[i − 1]; d[i]: = d[i − 1]; a[i]: = a[i − 1] − z
        end;
        t[1]: = tn; d[1]: = dn; a[1]: = z;
        {utriedený súbor je na páske t[1]}
        list (f[t[1]], t[1]); úroveň: = úroveň − 1
    until úroveň = 0;
end {polyfázovétriedenie};
begin {vytvorenie súboru s pseudonáhodnými prvkami}
  leng: = 200; rand: = 7789;
  repeat rand: = (131071 * rand) mod 2147483647;
    buf.kľúč: = rand div 2147484; write (f0, buf);
    leng: = leng − 1
  until leng = 0;
  reset (f0); list (f0, 1);
  polyfázovétriedenie
end.
```

2.3.5 DISTRIBÚCIA ZAČIATOČNÝCH BEHOV

Vzhľadom na to, že pri jednoduchých metódach vnútorného triedenia polí sme implicitne predpokladali existenciu dostatočne veľkej operačnej pamäti (na uchovanie celej množiny triedených údajov), boli sme

v prípade vonkajšieho triedenia (rozsiahlych súborov údajov) nútení dospieť k dômyselnejším metódam. Pre veľké sekvenčné súbory obyčajne nemáme k dispozícii dostatočne veľkú operačnú pamäť počítača. Preto musíme siahnuť po iných veľkokapacitných sekvenčných pamäťových médiách, akými sú napr. magnetické pásky. Uvedomme si pritom, že doterajšie metódy vonkajšieho triedenia nevyžadovali prakticky žiadnu primárnu pamäť okrem pomocných premenných (potrebných na realizáciu algoritmu) a samotného programu. Na druhej strane si treba uvedomiť, že aj minipočítače obsahujú určitú primárnu pamäť, ktorá je takmer vždy väčšia ako požadovaná veľkosť triediacich programov. Zanedbanie optimálneho využitia tejto pamäti by bolo preto ozaj neospravedlniteľné.

Riešenim je kombinácia metód vnútorného triedenia s vonkajším. Prakticky to znamená, že prispôsobená technika vnútorného triedenia polí sa dá využiť pri distribúcii začiatočných behov s tým, že tieto behy budú mať vždy dĺžku *l*, ktorá sa približne rovná veľkosti dostupnej primárnej pamäti. Je pochopiteľné, že žiadne dodatočné vnútorné triedenie v nasledujúcich zlučovacích fázach už nemôže zlepšiť účinnosť algoritmu, pretože behy sa stále zväčšujú, a tým, samozrejme, ich dĺžka presiahne veľkosť operačnej pamäti. Vzhľadom na to sa kľudne môžeme venovať vylepšeniu algoritmu generovania začiatočných behov.

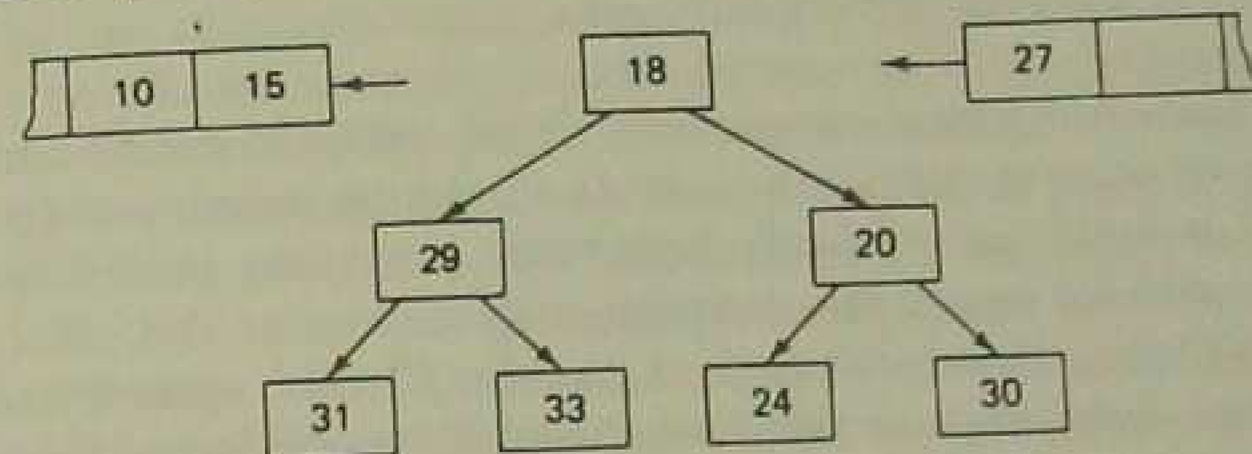
Prirodzene, sústredíme sa na niektorú z logaritmických metód vnútorného triedenia. Najvhodnejšou sa ukazuje triedenie haldou (pozri odsek 2.2.5). Haldu si môžeme predstaviť ako tunel, cez ktorý musia prejsť triedené prvky; niektoré rýchlejšie, niektoré pomalšie. Najmenší kľúč sa ľahko sníme z koreňa haldy a jeho nahradenie je veľmi efektívny proces. Celý proces „putovania prvku" zo vstupnej pásky $f0$ cez „haldový tunel" h na výstupnú pásku $f[j]$ môžeme jednoducho vyjadriť týmto spôsobom

```
write (f[j], h[1]);
read (f0, h[1]);                (2.51)
vytvorhaldu (1, n)
```

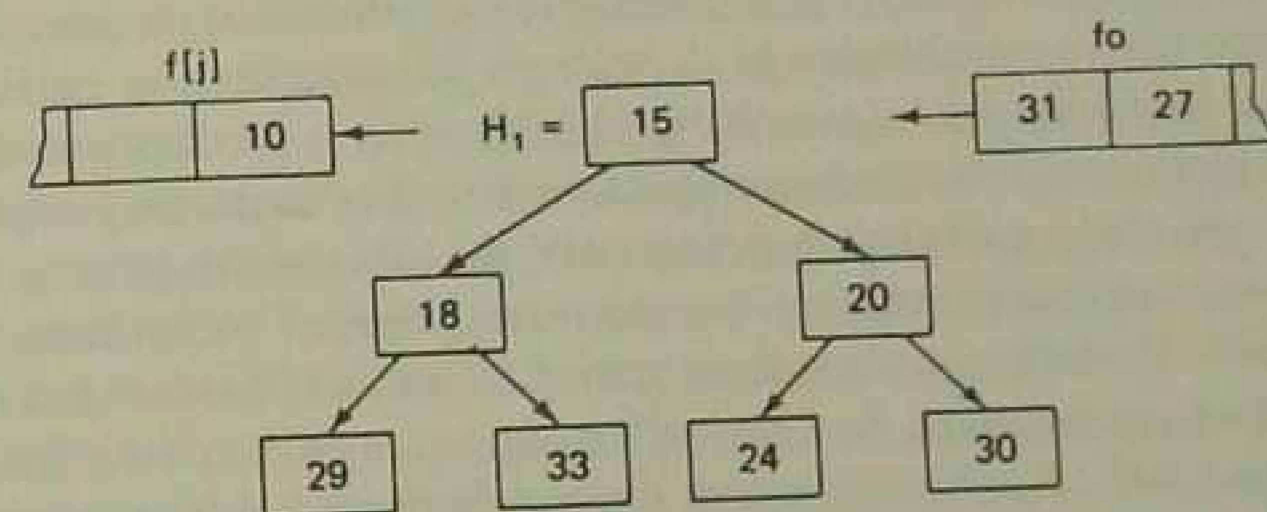
Procedúra vytvorhaldu je opísaná v odseku 2.2.5. Pomocou nej sa novovložený prvok $h[1]$ dostane na svoju správnu pozíciu v halde.

Uvedomme si, že prvok $h[1]$ je najmenší prvok haldy. Príklad je znázornený na *obr. 2.17.*

Situácia pred presunom:



Situácia po presune:



Obr. 2.17. Odobratia prvku z koreňa haldy a pridanie nového prvku do haldy

Program bude pravdepodobne po týchto modifikáciách zložitejší, pretože:

1. Halda je spočiatku prázdna a musí sa najskôr zaplniť začiatočnými prvkami.
2. Tesne pred koncom triedenia je halda už iba čiastočne zaplnená a na konci sa vyprázdni.
3. Musíme udržiavať údaje o začiatku nových behov, aby sme mohli v pravý čas zmeniť index j výstupnej pásky.

Skôr ako budeme pokračovať, formálne zadeklarujeme všetky premenné, ktoré budeme v ďalšom potrebovať:

```
var f0: páska;
    f: array [číslopásky] of páska;
    h: array [1 .. m] of prvok;                    (2.52)
    l, r: integer
```

Konštantou *m* budeme označovať veľkosť haldy *h*, konštantou *mh* budeme vyjadrovať polovičnú hodnotu konštanty *m* ($mh = m/2$). Premenné *l*, *r* budú indexy haldy *h*. Proces „putovania prvku" potom môžeme rozdeliť na päť samostatných etáp:

1. Prečítanie prvých *mh* prvkov z pásky *f*0 a ich umiestnenie do hornej časti haldy, kde sa nevyžaduje žiadne usporiadanie kľúčov.

2. Prečítanie ďalších *mh* prvkov a ich umiestnenie na správne pozície v dolnej časti haldy. Použijeme pritom procedúru vytvorhaldu.

3. Priradenie konštanty *m* premennej *l* a opakovanie nasledujúceho kroku pre všetky zostávajúce prvky súboru *f*0: umiestnenie prvku *h*[1] z koreňa haldy na príslušnú výstupnú pásku; ak je hodnota jej kľúča menšia, alebo sa rovná hodnote kľúča nasledujúceho prvku vstupného súboru, tak tento nasledujúci prvok patrí do toho istého behu a môže sa prostredníctvom procedúry vytvorhaldu umiestniť na správnu pozíciu v halde. V opačnom prípade sa zredukuje veľkosť haldy a príslušný nový prvok sa umiestni do „hornej" polovice haldy, kde sa formuje nasledujúci beh. Rozhranie medzi dvoma časťami haldy označujeme pomocou indexu *l*. Potom „dolná" alebo momentálna časť haldy obsahuje prvky *h*[1] ... *h*[*l*] a „horná" alebo „formujúca nasledujúci beh" obsahuje prvky *h*[*l* + 1] ... *h*[*m*]. Ak $l = 0$, tak treba vymeniť vstupné pásky a premennej *l* priradiť hodnotu *m*.

4. Prečítal sa posledný prvok zdrojového súboru. Najskôr sa premennej *r* priradí hodnota *m*, potom sa prvky z ľavej časti haldy, ktoré tvoria jeden beh, postupne presunú na výstupnú pásku a súčasne sa vybuduje horná časť haldy. V nej sa prvky postupne premiestnia na pozície *h*[*l* + 1] ... *h*[*r*].

5. Posledný beh sa vytvorí zo zostávajúcich prvkov haldy.

Na základe opisu týchto piatich častí môžeme ich detailne formulovať v tvare kompletného programu. Tento využíva procedúru vyberpásku, len čo sa zistí koniec behu. V dôsledku toho sú potrebné niektoré činnosti súvisiace so zmenou indexu výstupnej pásky. V programe

2.17 je však namiesto nich použitá iba fiktívna procedúra, ktorej jedinou činnosťou je sčítavanie vygenerovaných behov. Všetky prvky sa zapisujú na pásku *f*1.

Ak by sme sa teraz pokúsili začleniť tento program napr. do programu pre polyfázové triedenie, narazili by sme na vážne ťažkosti, ktoré vzniknú z týchto dôvodov: program triedenia obsahuje vo svojej inicializačnej fáze pomerne zložitú procedúru na prepínanie pások a spolieha sa na existenciu procedúry skopírujbeh, ktorá je schopná premiestniť práve jeden beh na zvolenú výstupnú pásku. Program triedenia haldou zase potrebuje procedúru vyberpásku, ktorou sa vyberie ďalšia páska. Týmto by, samozrejme, ešte nemuselo dôjsť ku konfliktom, ak by sa v rámci jedného alebo obidvoch uvedených programov volala požadovaná procedúra na jedinom mieste. Skutočná situácia je, žiaľ, iná: procedúra je volaná v oboch programoch na viacerých miestach. Táto situácia sa dá najlepšie vyriešiť použitím tzv. korutiny, ktorá je výhodná najmä v tých prípadoch, keď súčasne existujú viaceré procesy. Typickým prípadom je kombinácia procesu, ktorý vytvára tok informácií v rôznych entitách, a procesu, ktorý tento tok spracúva. Tento vzťah producent — spotrebiteľ možno vyjadriť prostredníctvom dvoch korutín. Jednou z nich môže byť aj hlavný program.

Z hľadiska teórie programovacích jazykov a programovania považujeme korutinu za procedúru alebo podprogram, ktorý obsahuje jeden alebo viacero bodov prerušenia. Len čo sa v priebehu toku riadenia programu narazí na takýto bod, vráti sa riadenie do programu, ktorý túto korutinu vyvolal. Ďalšou aktiváciou korutiny bude jej výpočet pokračovať inštrukciou, nasledujúcou za bodom prerušenia. V našom príklade môžeme algoritmus polyfázového triedenia považovať za hlavný program, z ktorého sa vyvoláva procedúra skopírujbeh, formulovaná ako korutina. Pozostáva z hlavného tela programu 2.17, v ktorom každé volanie procedúry vyberpásku predstavuje prerušenie. Test, ktorým sa zisťuje koniec súboru, sa potom systematicky nahrádza testom na koniec tela korutiny. Logicky znejúca formulácia bude mať teda tvar eoc(skopírujbeh) namiesto eof(*f*0).

PROGRAM 2.17. *Distribúcia začiatočných behov prostredníctvom haldy*

```
program Distribúcia (f0, f1, output);
{distribúcia začiatočných behov prostredníctvom triedenia haldou}
const m = 30; mh = 15; {veľkosť haldy}
type prvok = record
                 kľúč: integer
             end;
     páska = file of prvok;
     index = 0..m;
var l, r: index;
    f0, f1: páska;
    počet: integer; {počítač behov}
    h: array [1..m] of prvok; {halda}
procedure vyberpásku;
begin počet: = počet + 1;
     {prázdne telo procedúry; počíta iba celkový počet
     distribuovaných behov}
end {vyberpásku};
procedure vytvorhaldu (l, r: index);
   label 13;
   var i, j: integer; x: prvok;
begin i: = l; j: = 2 * i; x: = h[i];
   while j ≦ r do
   begin if j < r then
            if h[j].kľúč > h[j + 1].kľúč then j: = j + 1;
         if x.kľúč ≦ h[j].kľúč then goto 13;
         h[i]: = h[j]; i: = j; j: = 2 * i
   end;
13: h[i]: = x
end;
begin {vytvorenie začiatočných behov pomocou triedenia haldou}
   počet: = 0; reset (f0); rewrite (f1);
   vyperpásku;
{1. krok: naplnenie hornej časti haldy h}
   l: = m;
```



```
   repeat read (f0, h[l]); l: = l - 1
   until l = mh;
{2. krok: naplnenie dolnej polovice haldy h}
   repeat read (f0, h[l]); vytvorhaldu (l, m); l: = l - 1
   until l = 0;
{3. krok: generovanie behov prostredníctvom haldy}
   l: = m;
   while ¬ eof (f0) do
   begin write (f1, h[1]);
      if h[1].kľúč ≦ f0↑.kľúč then
      begin {nový prvok patrí do bežného behu}
         read (f0, h[1]); vytvorhaldu (1, l);
      end else
      begin {nový prvok patrí do nového behu}
         h[1]: = h[l]; vytvorhaldu (1, l - 1);
         read (f0, h[l]); if l ≦ mh then vytvorhaldu (l, m);
         l: = l - 1;
         if l = 0 then
         begin {halda je plná; začiatok nového behu}
            l: = m; vyberpásku
         end
      end
   end;
{4. krok: vyprázdnenie dolnej časti haldy}
   r: = m;
   repeat write (f1, h[1]);
      h[1]: = h[l]; vytvorhaldu (1, l - 1);
      h[l]: = h[r]; r: = r - 1;
      if l ≦ mh then vytvorhaldu (l, r); l: = l - 1
   until l = 0;
{5. krok: vyprázdnenie hornej časti haldy; vytvorenie posledného behu}
   vyberpásku;
   while r > 0 do
   begin write (f1, h[1]); h[1]: = h[r]; vytvorhaldu (1, r);
         r: = r - 1
   end;
```

```
    writeln (počet)
end.
```

Analýza a závery

Akú efektívnosť môžeme očakávať od algoritmu polyfázového triedenia s distribúciou začiatočných behov prostredníctvom haldy? Najskôr preskúmame zlepšenie, ktoré očakávame zavedením haldy.

Priemerná očakávaná dĺžka behov v prípade postupnosti s náhodne distribuovanými kľúčmi sa rovná dvom. Aká bude v prípade distribúcie prostredníctvom haldy s veľkosťou m? Prirodzená odpoveď, ktorá nás okamžite napadá, znie m, ale výsledok analýzy pravdepodobnosti je oveľa priaznivejší, a to $2m$ (pozri [2-7], s. 254). Preto očakávaný koeficient zlepšenia sa rovná konštante m. Účinnosť polyfázového triedenia možno odhadnúť na základe údajov uvedených v *tab. 2.15*, označujúcich maximálne počty začiatočných behov, ktoré sa dajú utriediť v rámci daného počtu prechodov (úrovní) s daným počtom pások (n). Ako príklad si zoberme 6 pások a haldu s veľkosťou $m = 100$. Potom súbor s počtom začiatočných behov až do 165 680 100 sa dá utriediť v rámci dvadsiatich prechodov algoritmu. Tento výsledok je skutočne pozoruhodný.

Počty behov umožňujúcich ideálnu distribúciu *Tabuľka 2.15*

l \ n	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1 261	1 531
10	144	653	1 297	1 921	2 501	3 049
11	233	1 201	2 500	3 777	4 961	6 073
12	377	2 209	4 819	7 425	9 841	12 097
13	610	4 063	9 289	14 597	19 521	24 097
14	987	7 473	17 905	28 697	38 721	48 001
15	1 597	13 745	34 513	56 417	76 806	95 617
16	2 584	25 281	66 526	110 913	152 351	190 465
17	4 181	46 499	128 233	218 049	302 201	379 399
18	6 765	85 525	247 177	428 673	599 441	755 749
19	10 946	157 305	476 449	842 749	1 189 041	1 505 425
20	17 711	289 329	918 385	1 656 801	2 358 561	2 998 753

Vráťme sa späť ku kombinácii polyfázového triedenia s distribúciou prostredníctvom haldy. Musíme žasnúť nad zložitosťou tohto programu. Koniec koncov, tento program v prvom rade plní svoju základnú funkciu, ktorou je, podobne ako v prípade ktoréhokoľvek priameho algoritmu vnútorného triedenia, preusporiadanie množiny nejakých prvkov.

Záverom zhrňme zámery tejto kapitoly. Išlo nám predovšetkým o poukázanie na

a) úzke spojenie medzi algoritmom a štruktúrami údajov (s ktorými tento algoritmus manipuluje) a najmä na vplyv štruktúr na algoritmus,

b) dômyselné stratégie, pomocou ktorých sa dá zlepšiť výkonnosť programu, dokonca aj v tých prípadoch, keď štruktúra reprezentujúca jeho údaje je nevhodná na konkrétnu úlohu (napr. použitie postupnosti namiesto poľa).

Cvičenia

2.1. Ktoré z algoritmov uvedených v programoch 2.1—2.6, 2.8, 2.10 a 2.11 sú stabilnými metódami triedenia?

2.2. Bude program 2.2 pracovať korektne, ak podmienku $l \leqq r$ nahradíme podmienkou $l < r$ v príkaze cyklu **while**? Bude korektný, ak príkazy $r := m - 1$ a $l := m + 1$ zjednodušíme na $r := m$ a $l := m$? Ak nie, nájdite množiny takých hodnôt $a_1 \dots a_n$, pre ktoré bude tento program nekorektný.

2.3. Naprogramujte a zmerajte čas výpočtu všetkých troch priamych metód triedenia na vašom počítači a zistite hodnoty koeficientov, ktorými by bolo potrebné vynásobiť faktory C a M, aby určovali odhad reálneho času.

2.4. Otestujte program triedenia haldou 2.8 pre viaceré náhodne zvolené vstupné postupnosti a zistite priemerný počet vykonaní

príkazu **goto** 13. Pretože tento počet je pomerne malý, zaujímavejšou bude zrejme otázka: existuje spôsob vylúčenia testu

$$x.kľúč \geqq a[j].kľúč$$

z príkazu cyklu **while**?

2.5. Majme nasledujúcu „rudimentárnu" verziu programu rozdeľovania poľa na úseky 2.9:

```
i: = 1; j: = n;
x: = a[(n + 1)] div 2.kľúč;
repeat
   while a[i].kľúč < x do i: = i + 1;
   while x < a[j].kľúč do j: = j - 1;
   w: = a[i]; a[i]: = a[j]; a[j]: = w
until i > j
```

Nájdite množiny hodnôt $a_1, \ldots, a_n$, pre ktoré bude tento program nekorektný.

2.6. Napíšte program, ktorý by používal aj quicksort, aj bublinové triedenie, a to nasledujúcim spôsobom: Algoritmom quicksort by sa vytvorili (neusporiadané) úseky poľa dĺžky m $(1 \leqq m \leqq n)$ a bublinovým triedením by sa tieto úseky dotriedili. Poznamenávame, že druhý krok (bublinové triedenie) môže postihnúť celé n-prvkové pole, a tým minimalizovať uchovávanie informácií, ktoré je potrebné preniesť z jedného kroku algoritmu do druhého. Nájdite takú hodnotu konštanty m, pre ktorú bude celkový čas triedenia najmenší.

Poznámka: Je isté, že optimálna hodnota konštanty m bude celkom malá. Preto sa zrejme oplatí aplikovať na poli bublinové triedenie $(m - 1)$-krát, čím sa ušetrí posledný prechod, ktorý zisťuje potrebu ďalšej výmeny.

2.7. Uskutočnite podobný experiment ako v cvičení 2.6, ale namiesto bublinového triedenia použite metódu priameho výberu. Vzhľadom na to, že triedením pomocou priameho výberu sa v jednom kroku nemôže preusporiadať celé pole, bude manipulácia s indexmi poľa o niečo náročnejšia.

2.8. Napíšte rekurzívnu verziu algoritmu quicksort, v ktorej sa bude

menší úsek poľa triediť predtým, než sa začne triedenie väčšieho úseku. Nech sa prvé triedenie uskutoční iteratívne, druhé prostredníctvom rekurzívneho volania procedúry triedenia. (Tým bude vaša procedúra triedenia obsahovať jedno rekurzívne volanie namiesto dvoch, ako to bolo v prípade programu 2.10, alebo žiadne, ako to bolo v programe 2.11.)

2.9. Nájdite permutáciu kľúčov 1, 2, ..., n, pre ktorú dosiahne quicksort najhoršie (najlepšie) výsledky (n = 5, 6, 8).

2.10. Vytvorte program triedenia prirodzeným zlučovaním podobný programu 2.13 (t. j. algoritmu priameho zlučovania). Nech tento program manipuluje s dvojnásobne dlhým poľom z obidvoch koncov smerom dovnútra. Porovnajte jeho výkonnosť s programom 2.13.

2.11. Pri (dvojcestnom) prirodzenom zlučovaní nie je potrebné vyberať stále naslepo najmenšiu hodnotu spomedzi všetkých kľúčov. Namiesto toho pri výskyte konca behu sa zvyšok druhého behu jednoducho pripojí k výstupnej postupnosti. Napríklad zlučovaním prvkov

2, 4, 5, 1, 2, ...
3, 6, 8, 9, 7, ...

vznikne postupnosť

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

namiesto postupnosti

2, 3, 4, 5, 1, 2, 6, 8, 9, ...

ktorá sa zdá byť lepšie usporiadanou. Aký je dôvod pre takúto stratégiu?

2.12. Zistite, prečo sme v programe 2.15 zaviedli premennú *ta* a za akých okolností sa vykoná príkaz

begin rewrite (*f*[*ta*[*mx*]]); ...

a za akých príkaz

begin *tx*: = *ta*[*mx*]; ...

2.13. Načo potrebujeme v programe polyfázového triedenia 2.16 premennú *posledná*? Prečo ju nepotrebujeme v programe 2.15?

2.14. Metóda triedenia podobná polyfázovému triedeniu sa nazýva triedenie kaskádovým zlučovaním [2-1] a [2-9]. Je však založená na inom princípe zlučovania. Majme napr. 6 pások *T*1, ..., *T*6. Kaskádové zlučovanie takisto začína „ideálnou distribúciou" behov na pásky *T*1, ..., *T*5. Potom sa vykoná päťcestné zlučovanie z pások *T*1, ..., *T*5 na pásku *T*6. Končí v okamihu, keď je páska *T*5 prázdna. Za ním nasleduje štvorcestné zlučovanie (bez uvažovania pásky *T*6) na pásku *T*5, potom trojcestné na *T*4, dvojcestné na *T*3 a nakoniec operácia kopírovania z pásky *T*1 na *T*2. Nasledujúci prechod sa vykonáva podobným spôsobom: začína päťcestným zlučovaním na pásku *T*1 atď. Hoci sa táto triediaca schéma zdá byť oproti polyfázovej ťažkopádnejšou (najmä preto, že občas vyberá a necháva niektoré pásky neaktívnymi a navyše zahŕňa jednoduché kopírovacie operácie), je pre (veľmi) veľké súbory a pre šesť a viacero pások prekvapujúco účinnejšia ako polyfázové triedenie. Napíšte dobre štruktúrovaný program triedenia založený na princípoch kaskádového zlučovania.

Zoznam použitej literatúry

2-1. Betz, B. K.: A Carter ACM National Conf., 14 (1959), Referát č. 14.

2-2. Floyd, R. W.: Treesort (Algoritmy 113 a 243), Comm. ACM, 5, No. 8 (1962), s. 434, a Comm. ACM, 7, No. 12 (1964), s. 701.

2-3. Gilstad, R. L.: Polyphase Merge Sorting-An Advanced Technique, Proc. AFIPS Eastern Jt. Comp. Conf., 18 (1960), s. 143—148.

2-4. Hoare, C. A. R.: Proof of a Program: FIND, Comm. ACM, 13, No. 1 (1970), s. 39—45.

2-5. Hoare, C. A. R.: Proof of a Recursive Program: Quicksort, Comp. J., 14, No. 4 (1971), s. 391—395.

2-6. Hoare, C. A. R.: Quicksort, Comp. J., 5, No. 1 (1962), s. 10—15.

2-7. Knuth, D. E.: The Art of Computer Programming, Vol. 3 Reading, Mass.: Addison-Wesley (1973).

2-8. Knuth D. E.: The Art of Computer Programming, 3, s. 86—95.

2-9. Knuth, D. E.: The Art of Computer Programming, 3, s. 289.

2-10. Lorin, H.: A Guided Bibliography to Sorting, IBM Syst. J., 10, No. 3 (1971), s. 244—254.

2-11. Shell, D. L.: A Highspeed Sorting Procedure, Comm. ACM, 2, No. 7 (1959), s. 30—32.

2-12. Singleton, R. C.: An Efficient Algorithm for Sorting with Minimal Storage (Algorithm 347), Comm. ACM, 12, No. 3 (1969), s. 185.

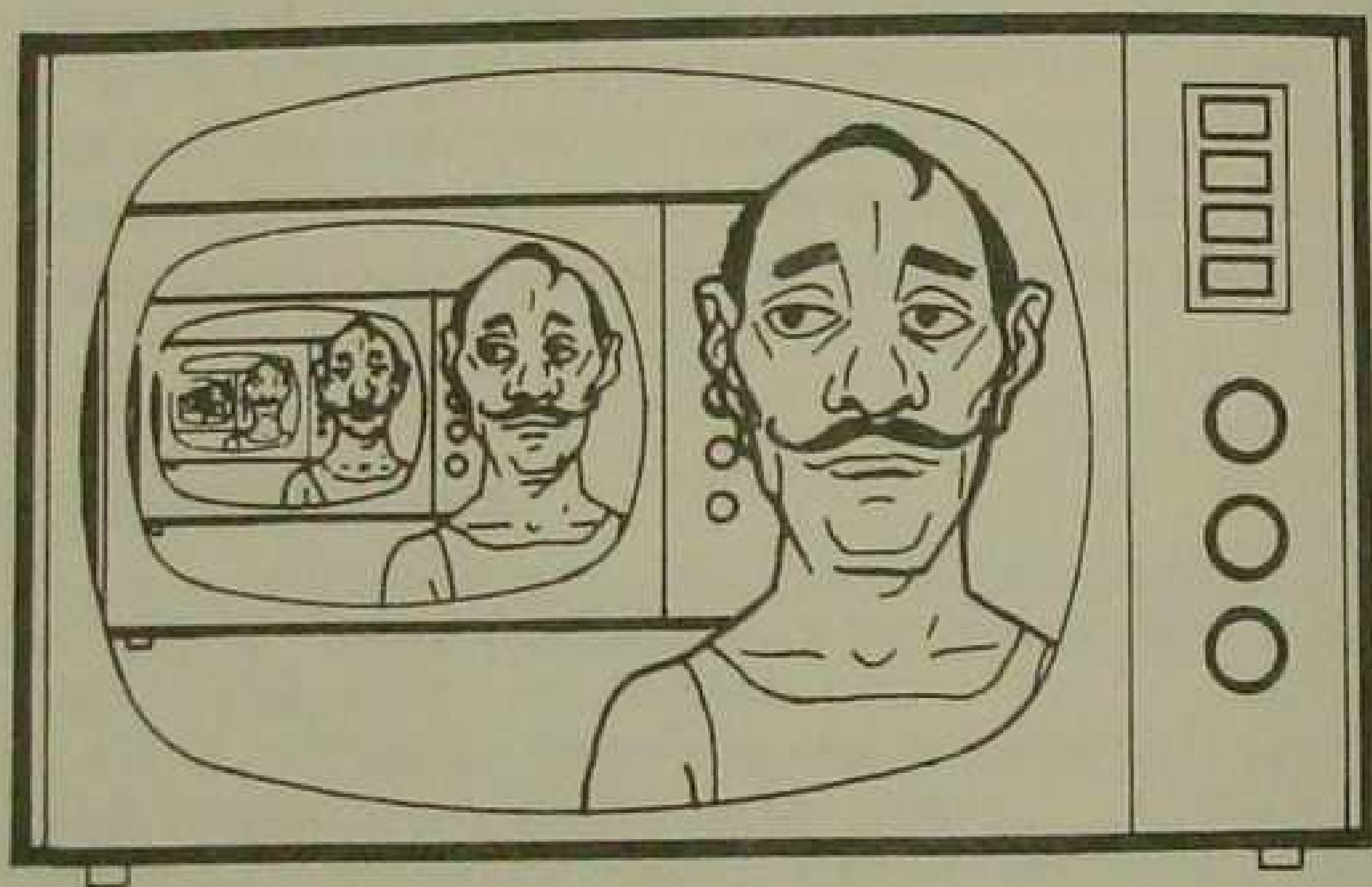
2-13. Van Emden, M. H.: Increasing the Efficiency of Quicksort (Algorithm 402), Comm. ACM, 13, No. 9 (1970), s. 563—566, 693.

2-14. Williams, J. W. J.: Heapsort (Algorithm 232), Comm. ACM, 7, No. 6 (1964), s. 347—348.

3 REKURZÍVNE ALGORITMY

3.1 ÚVOD

Hovoríme, že objekt je *rekurzívny*, ak sa čiastočne skladá, alebo je definovaný pomocou seba samého. S rekurziou sa nestretávame iba v matematike, ale aj v bežných situáciách denného života. Kto by sa napr. ešte nestretol s reklamným obrázkom, ktorý obsahuje sám seba?



Obr. 3.1. Rekurzívny obrázok

Rekurzia je silným nástrojom najmä pri matematických definíciách. Známe sú príklady prirodzených čísel, stromových štruktúr a niektorých funkcií:

1. Prirodzené čísla:

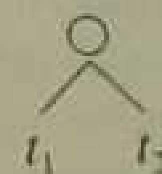
a) 1 je prirodzené číslo,

b) nasledovníkom prirodzeného čísla je prirodzené číslo.

2. Stromové štruktúry:

a) ○ je strom (nazývaný prázdny strom),

b) ak t_1 a t_2 sú stromy, tak



je strom (nakreslený obrátene — doluznačky).

3. Funkcia n-faktoriál $n!$ (pre nezáporné celé čísla):

a) $0! = 1$,

b) ak $n > 0$, tak $n! = n.(n-1)!$.

Sila rekurzie očividne spočíva v možnosti definovať nekonečnú množinu objektov konečným príkazom. Podobným spôsobom možno nekonečný počet výpočtov opísať pomocou konečného rekurzívneho programu, dokonca i vtedy, keď tento program neobsahuje explicitné opakovania. Prirodzene, rekurzívne algoritmy sú najvhodnejšie pri riešení problémov a pri výpočtoch funkcií alebo spracúvaní takých štruktúr údajov, ktoré už samy osebe sú definované rekurzívnym spôsobom. Vo všeobecnosti možno rekurzívny program vyjadriť kompozíciou $\mathscr{P}$ základných príkazov S_i (neobsahujúcich P) a samotné P:

$$P = \mathscr{P}[S_i, P] \tag{3.1}$$

Dôležitým a účinným prostriedkom na vyjadrenie rekurzie v programoch je procedúra (alebo podprogram), ktorej identifikátor slúži na rekurzívnu aktiváciu jej tela. Ak procedúra P obsahuje priamy odkaz na seba, tak o nej hovoríme, že je *priamo rekurzívna*; ak P obsahuje odkaz na inú procedúru Q, ktorá obsahuje (priamy alebo nepriamy) odkaz na procedúru P, tak P je *nepriamo rekurzívna*. Použitie rekurzie nemusí byť teda okamžite zrejmé z textu programu.

Býva zvykom združovať množinu lokálnych objektov s procedúrou, t. j. množinu premenných, konštánt, typov a procedúr, ktoré sú definované lokálne v rámci tejto procedúry a ktoré neexistujú, resp. nemajú zmysel mimo tejto procedúry. Každým rekurzívnym volaním takejto procedúry vznikne nová množina jej lokálnych premenných. Tieto premenné majú síce tie isté identifikátory, aké mali pri predošlom volaní procedúry, ale ich hodnoty sú odlišné. Konfliktom pri pomenú-

vaní sa predchádza na základe pravidla viditeľnosti (rozsahu) identifikátorov: identifikátory sa vzťahujú vždy na najposlednejšie vytvorenú množinu premenných. Rovnaké pravidlo platí pre parametre procedúr, ktoré sú na základe definície zviazané s procedúrou.

Podobne, ako príkazy cyklov, aj rekurzívne procedúry dávajú možnosť nekonečných výpočtov. Preto je potrebné zaoberať sa otázkou ukončenia výpočtu. Základnou požiadavkou je, aby sa rekurzívne volanie procedúry P riadilo podmienkou B, ktorá môže byť za určitých okolností nesplnená. Preto sa dá schéma rekurzívneho algoritmu presnejšie vyjadriť týmto vzťahom:

$$P \equiv \textbf{if } B \textbf{ then } \mathscr{P}[S_i, P] \tag{3.2}$$

alebo

$$P \equiv \mathscr{P}[S_i, \textbf{if } B \textbf{ then } P] \tag{3.3}$$

Základnou technikou dôkazu konečnosti cyklu je určenie funkcie $f(x)$ (kde x je množina premenných programu) takej, že $f(x) \leqq 0$ implikuje podmienku ukončenia (príkazov cyklu **while** alebo **repeat**), a dôkaz toho, že hodnota funkcie $f(x)$ klesá v každom kroku cyklu. Tým istým spôsobom možno dokázať aj ukončenie rekurzívneho programu: stačí ukázať, že každé vykonanie P znižuje hodnotu $f(x)$. Jedným z vhodných spôsobov zaručenia ukončenia rekurzie je použitie (konštantného) parametra, povedzme n, procedúry P, ktorú potom voláme s $n - 1$, ako hodnotou parametra. Ukončenie rekurzívnych volaní je potom zaručené prostredníctvom podmienky $n > 0$ namiesto podmienky B. Toto môžeme vyjadriť nasledujúcou programovou schémou:

$$P(n) \equiv \textbf{if } n > 0 \textbf{ then } \mathscr{P}[S_i, P(n-1)] \tag{3.4}$$

$$P(n) \equiv \mathscr{P}[S_i, \textbf{if } n > 0 \textbf{ then } P(n-1)] \tag{3.5}$$

V praktických aplikáciách sa odporúča poukázať nielen na konečnosť hĺbky rekurzie, ale aj na malú hodnotu tejto hĺbky. Dôvod je prostý: každým vyvolaním rekurzívnej procedúry P sa vyžaduje pridelenie určitého množstva pamäti potrebnej na lokálne premenné procedúry P. Okrem týchto združených lokálnych premenných potrebujeme nejakú pamäť na uchovanie momentálneho stavu výpočtu, v ktorom sa nachádza rekurzívna procedúra. Tento moment je dôležitý vtedy, keď

skončí výpočet posledného vyvolania procedúry a je potrebné obnoviť jej predchádzajúci stav, v ktorom sa nachádzala pred posledným vyvolaním. S touto situáciou sme sa už stretli v druhej kapitole pri tvorbe procedúry quicksort. Tam sme zistili, že jednoduchým zložením príkazu, ktorým sa rozdeľuje množina n prvkov na dva úseky s dvoma rekurzívnymi volaniami procedúry triedenia (ktorou sa triedia tieto dva úseky), sa môže hĺbka rekurzie v najhoršom prípade rovnať hodnote n. Pri šikovnejšom odhade situácie je možné obmedziť túto hĺbku na hodnotu $\log n$. Rozdiel medzi hodnotou n a $\log n$ je postačujúci na zmenu prípadu, veľmi nevhodného pre rekurziu, na prípad, keď je použitie rekurzie zvlášť užitočné.

3.2 KEDY NEPOUŽÍVAŤ REKURZIU

Rekurzívne algoritmy sú výhodné najmä vtedy, keď problém, ktorý sa rieši, alebo údaje, s ktorými sa má manipulovať, sú definované rekurzívne. To však neznamená, že rekurzívne definície automaticky zaručujú, že použitie rekurzívneho algoritmu bude najlepším spôsobom riešenia daného problému. Skutočne, objasnenie pojmu rekurzívny algoritmus prostredníctvom nevhodných príkladov bolo najväčším dôvodom vzniku všeobecných obáv a antipatií proti použitiu rekurzie pri programovaní. Súčasne vznikla domnienka, že rekurzia je neefektívna. Navyše si pripomeňme, že i taký rozšírený programovací jazyk, akým je fortran, zakazuje používanie rekurzívneho volania procedúr. Tak sa zabránilo úvahám nad rekurzívnym riešením, dokonca aj v tých prípadoch, keď by to bolo vhodné. Programy, v ktorých sa nemá použiť rekurzia, možno charakterizovať pomocou schémy, ktorá zobrazuje vzor ich kompozície. Je to schéma (3.6) a jej ekvivalent (3.7).

$$P \equiv \textbf{if } B \textbf{ then } (S; P) \tag{3.6}$$

$$P \equiv (S; \textbf{if } B \textbf{ then } P) \tag{3.7}$$

Tieto schémy sú prirodzené v tých prípadoch, keď sa majú vypočítať hodnoty, definované pomocou jednoduchých rekurentných vzťahov. Zoberme napr. dobre známe čísla $f_i = i!$:

$$i = 0, 1, 2, 3, 4, 5, \dots$$
$$f_i = 1, 1, 2, 6, 24, 120, \dots \qquad (3.8)$$

Nulté číslo je explicitne definované ako $f_0 = 1$, pričom všetky nasledujúce čísla sú definované rekurzívne prostredníctvom ich predchodcu:

$$f_{i+1} = (i + 1) . f_i \qquad (3.9)$$

Tento vzorec naznačuje rekurzívny algoritmus, postupujúci po n-té číslo. Ak zavedieme dve premenné I a F na označenie hodnôt i a f_i na i-tej úrovni rekurzie, tak výpočet potrebný na prechod na ďalšiu úroveň (ďalšieho čísla postupnosti (3.8)) môžeme vyjadriť nasledujúcimi vzťahmi:

$$I: = I + 1; \quad F: = I * F \qquad (3.10)$$

Substitúciou vzťahov (3.10) za S vo vzťahu (3.6) dostávame rekurzívny program:

```
P ≡ if I < n then (I: = I + 1; F: = I * F; P)
I: = 0; F: = 1; P                                   (3.11)
```

Prvý riadok vzťahu (3.11) možno vyjadriť zaužívaným programovým zápisom:

```
procedure P;
begin if I < n then
      begin I: = I + 1; F: = I * F; P               (3.12)
      end
end
```

Ekvivalentný, ale častejšie používanejší zápis, vyjadruje vzťah (3.13). Procedúra P je v ňom nahradená funkciou, t. j. procedúrou, s ktorou je explicitne spojená návratová hodnota a ktorá sa môže preto priamo použiť ako operand v rámci výrazu. Tým sa premenná F stáva zbytočnou a úlohu premennej I preberá parameter procedúry.

```
function F(I: integer): integer;
begin if I > 0 then F: = I * F(I - 1)
                else F: = 1                         (3.13)
end
```


Očividné je, že v tomto prípade sa dá rekurzia nahradiť jednoduchou iteráciou, a to nasledujúcim programom:

```
I: = 0; F: = 1;
while I < n do
      begin I: = I + 1; F: = I * F                  (3.14)
      end
```

Vo všeobecnosti programy, ktorých štruktúra zodpovedá schémam (3.6) alebo (3.7), je možné transformovať na tvar zodpovedajúci schéme (3.15):

$$P \equiv (x: = x_0; \textbf{while } B \textbf{ do } S) \qquad (3.15)$$

Existujú, samozrejme, oveľa zložitejšie rekurzívne schémy, ktoré možno transformovať na iteratívny tvar. Príkladom môže byť výpočet Fibonacciho čísel definovaných rekurentným vzťahom

$$\mathrm{fib}_{n+1} = \mathrm{fib}_n + \mathrm{fib}_{n-1} \quad \text{pre} \quad n > 0 \qquad (3.16)$$

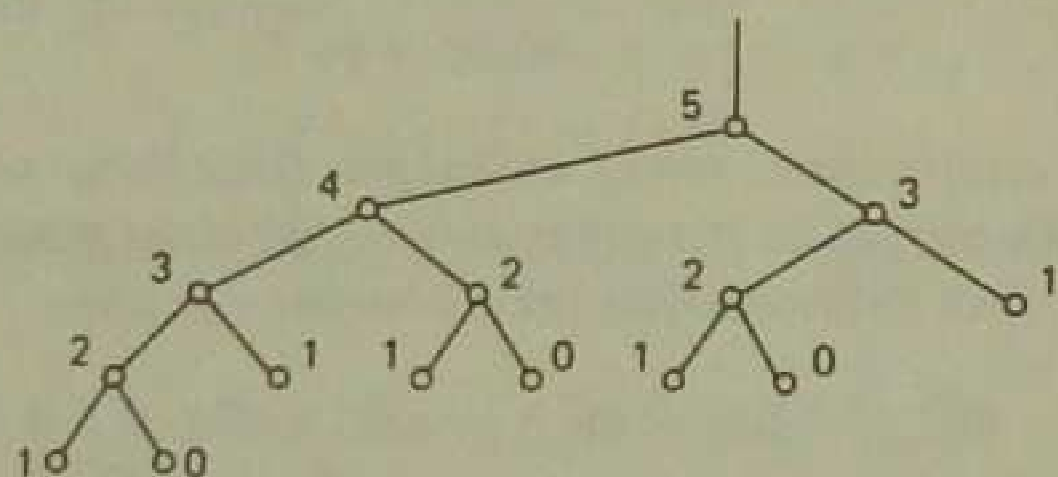
Navyše platí: $\mathrm{fib}_1 = 1$ a $\mathrm{fib}_0 = 0$. Priame a naivné riešenie môže viesť k tomuto programu:

```
function Fib(n: integer): integer;
begin if n = 0 then Fib: = 0 else
      if n = 1 then Fib: = 1 else                   (3.17)
      Fib: = Fib(n - 1) + Fib(n - 2)
end
```

Výpočet hodnoty fib_n prostredníctvom vyvolania funkcie Fib(n) spôsobí rekurzívne volanie tejto funkcie. Otázkou je, koľkokrát sa táto funkcia volá. Poznamenávame, že každé volanie s hodnotou parametra $n > 1$ má za následok dve ďalšie volania, t. j. celkový počet volaní rastie exponenciálne (*obr. 3.2*). Takýto program je zjavne nepraktický. Je jasné, že Fibonacciho čísla môžeme vypočítať i prostredníctvom iteratívnej schémy algoritmu, v ktorej sa navyše vyhneme vypočítavaniu rovnakých hodnôt pomocou zavedenia dvoch pomocných premenných x, y takých, že $x = \mathrm{fib}_i$ a $y = \mathrm{fib}_{i-1}$.

{výpočet $x = \mathrm{fib}_n$ pre $n > 0$}
$i := 1$; $x := 1$; $y := 0$;
while $i < n$ **do** (3.18)
  begin $z := x$; $i := i + 1$;
    $x := x + y$; $y := z$
  end

(Poznamenávame, že tri priraďovacie príkazy, ktoré priraďujú hodnoty premenným x, y, z, možno nahradiť dvoma príkazmi, pri ktorých nepotrebujeme pomocnú premenú z: $x := x + y$; $y := x - y$.)



Obr. 3.2. Znázornenie pätnástich vykonaní funkcie Fib(n) pre $n = 5$

Z uvedených úvah pre nás vyplýva ponaučenie: pokiaľ je možné daný problém vyriešiť pomocou iterácie, vyhýbajme sa rekurzii.

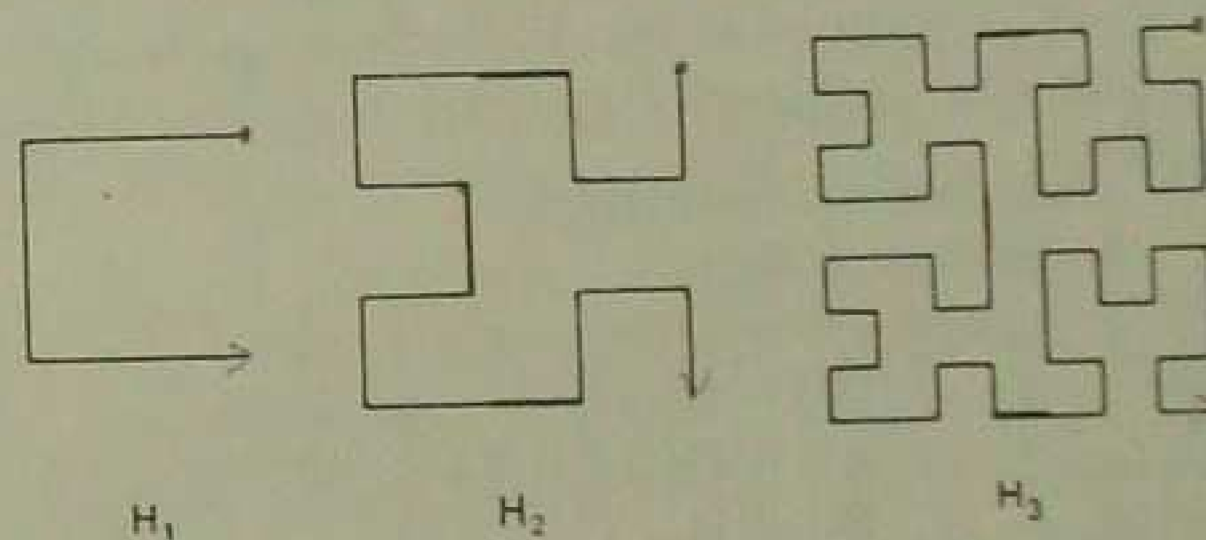
To, samozrejme, neznamená, že by sme sa mali za každú cenu snažiť zbaviť rekurzie. Existuje predsa mnoho vhodných aplikácií rekurzie, ako dokumentujú nasledujúce články.

Skutočnosť, že existujú implementácie rekurzívnych procedúr aj na nerekurzívnych počítačoch, potvrdzuje fakt, že každý rekurzívny program možno pre praktické účely transformovať na iteratívny tvar. To, pochopiteľne, vyžaduje explicitnú manipuláciu so zásobníkom rekurzívnej procedúry, čo môže viesť k zložitým operáciám, sťažujúcim prehľadnosť a zrozumiteľnosť programu. Nech je preto pre nás pravidlom, že algoritmy, ktoré sú svojou podstatou rekurzívne (skôr než iteratívne) majú byť implementované pomocou rekurzívnych procedúr. Na uvedomenie si tohto faktu odporúčame, aby si čitateľ všimol a hlavne porovnal programy 2.10 a 2.11.

Zvyšná časť tejto kapitoly je venovaná tvorbe niektorých rekurzívnych programov, najmä v prípadoch, keď je použitie rekurzie namieste. Podobne štvrtá a piata kapitola využíva rekurziu, a to hlavne v situáciách, keď použité štruktúry údajov priamo nabádajú k použitiu rekurzívnych algoritmov.

3.3 DVA PRÍKLADY REKURZÍVNYCH PROGRAMOV

Grafický obrazec, znázornený na *obr. 3.5*, je vytvorený superpozíciou piatich kriviek. Tieto krivky tvoria pravidelný vzor a možno ich nakresliť pomocou kresliaceho zariadenia (plottra) riadeného počítačom. Našim cieľom bude nájsť rekurzívnu schému, na základe ktorej možno skonštruovať program pre kresliace zariadenie. Podrobnejším prieskumom zistíme, že tri zo superponovaných kriviek majú tvary, ktoré sú zobrazené na *obr. 3.3*. Tieto krivky označíme H_1, H_2 a H_3.



Obr. 3.3. Hilbertove krivky prvého, druhého a tretieho rádu

Obrázok nám súčasne dokumentuje vznik krivky H_{i+1} z krivky H_i: krivku H_{i+1} dostaneme kompozíciou štyroch kriviek H_i polovičnej veľkosti, ich vhodnou rotáciou a spojením pomocou troch priamok. Krivku H_1 môžeme pritom považovať za kompozíciu štyroch prázdnych kriviek H_0 spojených tromi priamkami. Krivka H_i sa nazýva *Hilbertova krivka* i-tého rádu podľa svojho objaviteľa D. HILBERTA (1891).

Predpokladajme, že našimi základnými kresliacimi prostriedkami sú dve súradnicové premenné x a y, procedúra nastavplotter (nastaví

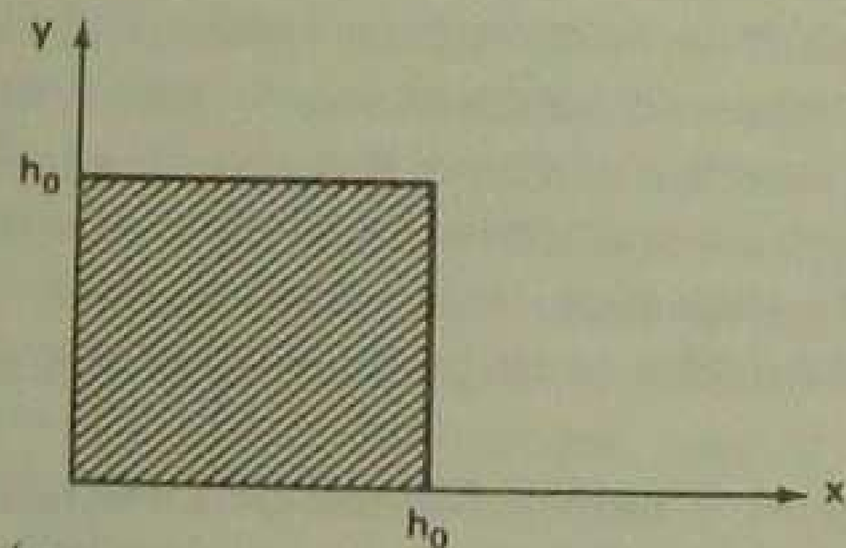
kresliace pero do polohy určenej súradnicami x, y) a procedúra plot (posunie pero z jeho momentálnej polohy do polohy určenej súradnicami x, y).

Pretože každá krivka H_i pozostáva zo štyroch polovičných kópií krivky H_{i-1}, je prirodzené vyjadriť procedúru pre nakreslenie krivky H_i ako kompozíciu štyroch častí, z ktorých každá nakreslí krivku H_{i-1} v primeranej mierke a pootočení. Ak označíme tieto štyri časti symbolmi A, B, C, D a procedúry, kresliace spájacie priamky pomocou šípok v zodpovedajúcom smere, tak získame nasledujúcu rekurzívnu schému (*obr. 3.3*):

$$\begin{aligned} &A: D \leftarrow A \downarrow A \leftarrow B \\ &B: C \uparrow B \leftarrow B \downarrow A \\ &C: B \rightarrow C \uparrow C \leftarrow D \\ &D: A \downarrow D \leftarrow D \uparrow C \end{aligned} \qquad (3.19)$$

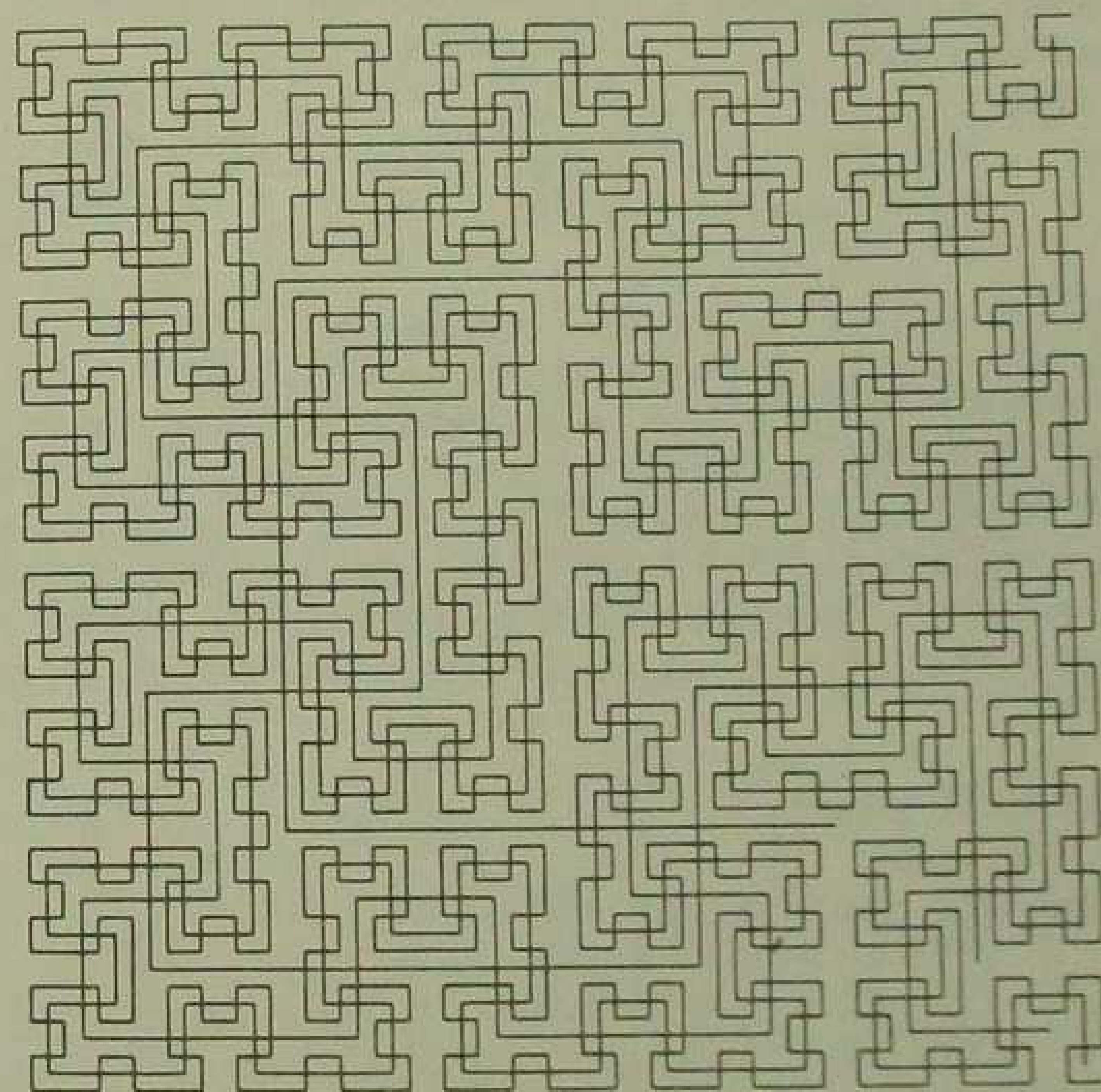
Ak označíme jednotkovú dĺžku priamky symbolom h, tak procedúra zodpovedajúca schéme A sa dá vystižne vyjadriť pomocou rekurzívneho volania podobne navrhnutých procedúr B a D a seba samej.

```
procedure A(i: integer);
begin if i > 0 then
    begin D(i − 1); x := x − h; plot;
          A(i − 1); y := y − h; plot;          (3.20)
          A(i − 1); x := x + h; plot;
          B(i − 1)
    end
end
```



Obr. 3.4. Jednotkový rámec

Táto procedúra sa volá v hlavnom programe, pričom každé jej volanie zodpovedá jednej superponovanej Hilbertovej krivke. V hlavnom programe sa súčasne stanoví začiatočný bod krivky, t. j. začiatočné hodnoty súradníc x, y a jednotkový prírastok h. Symbolom $h0$ vyjadrujeme maximálnu šírku strany. Vyžadujeme, aby $h0 = 2^k$ pre ľubovoľné $k \geqq n$ (*obr. 3.4*). Úplný program nakreslí n Hilbertových kriviek $H_1 \ldots H_n$ (pozri *obr. 3.5* a program 3.1).

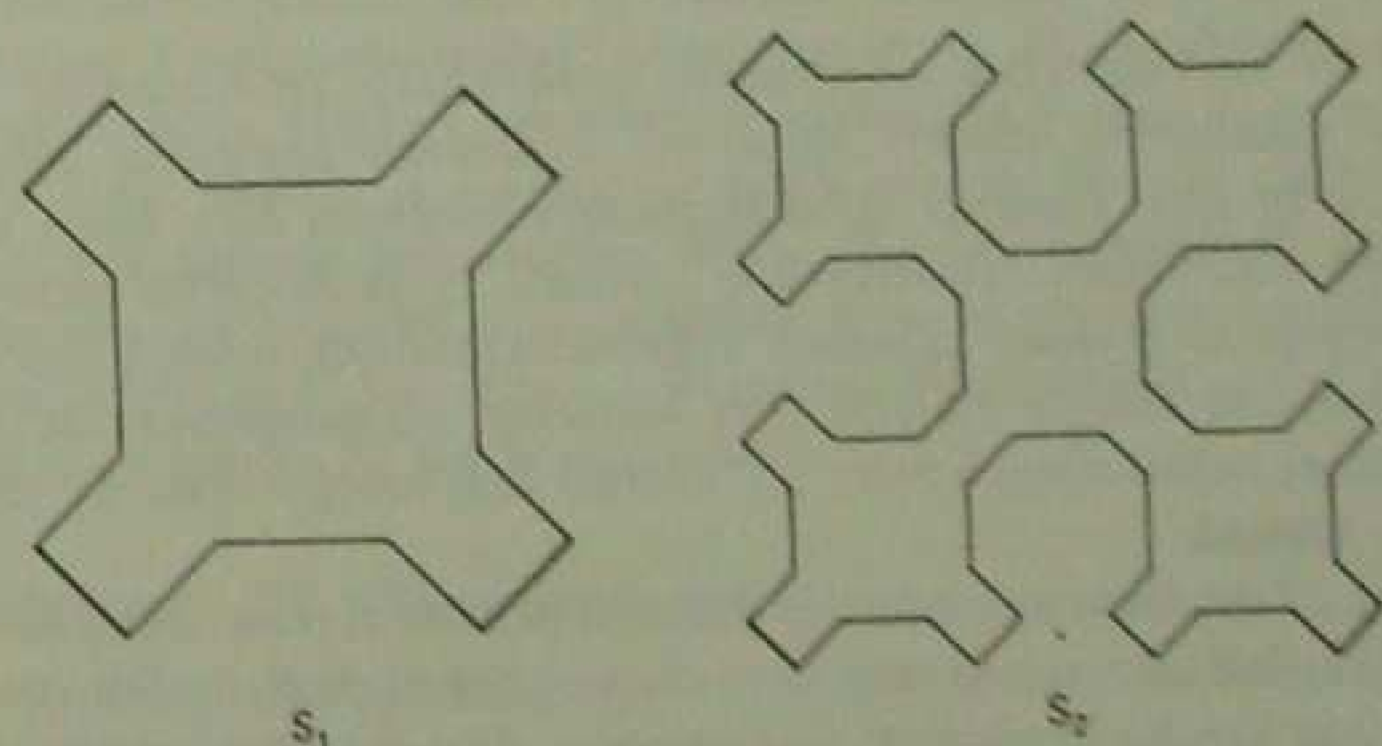


Obr. 3.5. Hilbertove krivky H_1, ..., H_5

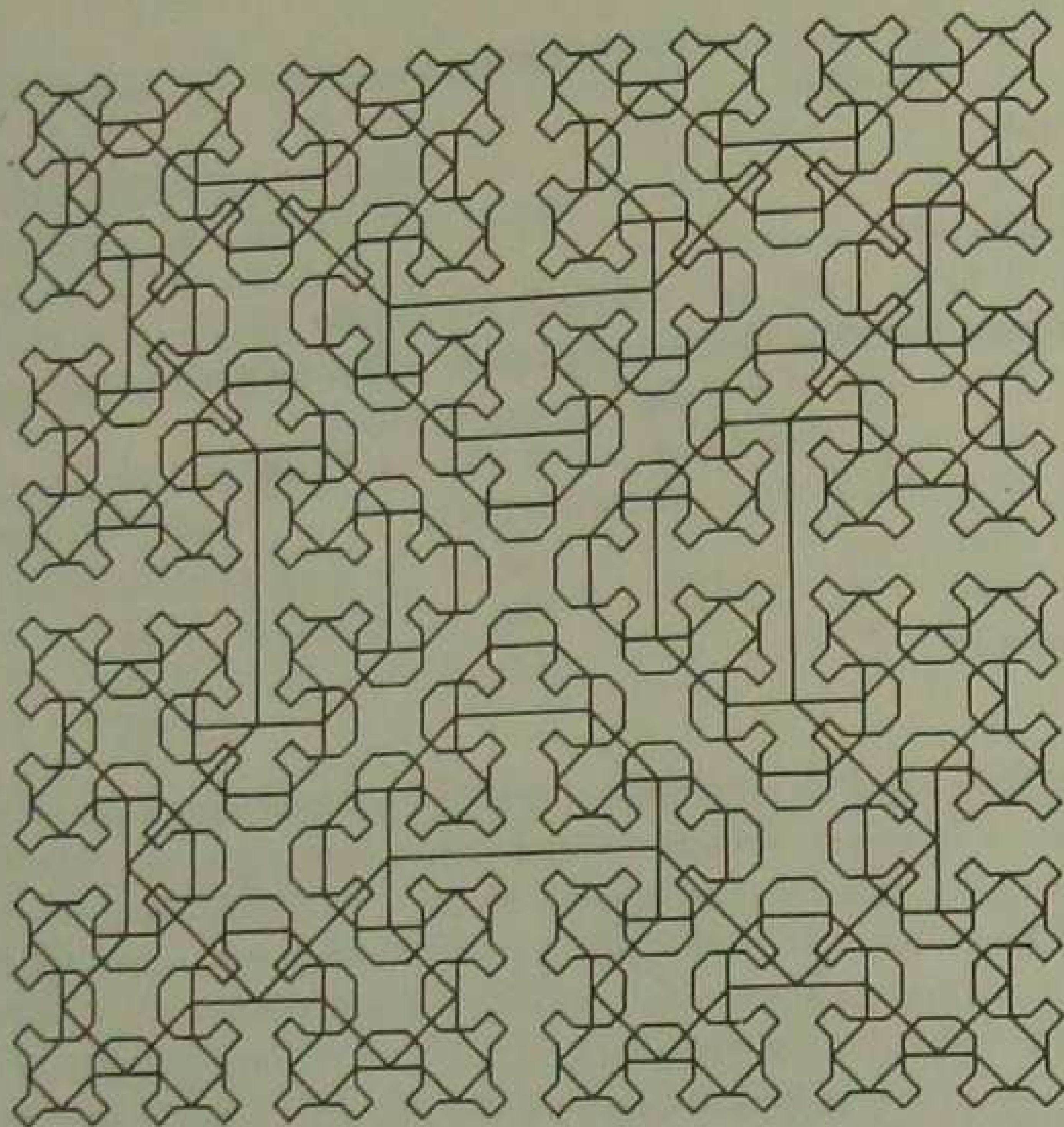
PROGRAM 3.1. *Hilbertove krivky*

```
program Hilbert(pf, output);
{kresli Hilbertove krivky rádu 1 až n}
const n = 4; h0 = 512;
var i, h, x, y, x0, y0: integer;
    pf: file of integer; {súbor obsahujúci príkazy pre
                          kresliace zariadenie}
procedure A(i: integer);
begin if i > 0 then
    begin D(i - 1); x: = x - h; plot;
          A(i - 1); y: = y - h; plot;
          A(i - 1); x: = x + h; plot;
          B(i - 1)
    end
end;
procedure B(i: integer);
begin if i > 0 then
    begin C(i - 1); y: = y + h; plot;
          B(i - 1); x: = x + h; plot;
          B(i - 1); y: = y - h; plot;
          A(i - 1)
    end
end;
procedure C(i: integer);
begin if i > 0 then
    begin B(i - 1); x: = x + h; plot;
          C(i - 1); y: = y + h; plot;
          C(i - 1); x: = x - h; plot;
          D(i - 1)
    end
end;
procedure D(i: integer);
begin if i > 0 then
    begin A(i - 1); y: = y - h; plot;
          D(i - 1); x: = x - h; plot;
          D(i - 1); y: = y + h; plot;
          C(i - 1)
    end
end;
begin startplot;
    i: = 0; h: = h0; x0: = h div 2; y0: = x0;
    repeat {kreslenie Hilbertovej krivky i-tého rádu}
      i: = i + 1; h: = h div 2;
      x0: = x0 + (h div 2); y0: = y0 + (h div 2);
      x: = x0; y: = y0; nastavplotter;
      A(i)
      until i = n;
    endplot
end.
```


Podobný príklad, ale o niečo zložitejší a dômyselnejší, je znázornený na *obr. 3.7.* Tento obrazec sa opäť získa superpozíciou niekoľkých kriviek, z ktorých dve sú zobrazené na *obr. 3.6.* S_i znamená *Sierpinského krivku i*-tého rádu. Ako bude vyzerať príslušná rekurzívna schéma? Keď sa pozrieme na *obr. 3.6*, tak nás to nabáda k tomu, aby sme list S_1 zvolili za základný stavebný blok, prípadne s jednou vynechanou hranou. Ale toto nevedie k úspešnému riešeniu. Podstatný rozdiel medzi Sierpinského a Hilbertovými krivkami spočíva v tom, že Sierpinského



Obr. 3.6. Sierpinského krivky prvého a druhého rádu



Obr. 3.7. Sierpinskeho krivky $S_1, \ldots, S_4$

krivky sú uzavreté (bez vzájomných prekrížení). To znamená, že základná rekurzívna schéma musí byť vyjadrená pomocou otvorenej krivky a že štyri časti sú spojené prostredníctvom čiar, ktoré nie sú súčasťou samotného rekurzívneho obrazca. V skutočnosti tieto čiary pozostávajú zo štyroch priamok v najvzdialenejších rohoch, nakreslených hrubšou čiarou na *obr. 3.6*. Môžeme ich považovať za súčasť neprázdnej začiatočnej krivky S_0, ktorou je štvorec stojaci na jednom svojom vrchole.

Teraz už môžeme jednoducho vytvoriť príslušnú rekurzívnu schému. Štyri základné obrazce sa opäť označia symbolmi A, B, C, D a spojovacie čiary sa kreslia explicitne. Všimnime si, že uvedené štyri rekurzívne obrazce sú skutočne identické okrem 90° pootočenia.

Základný vzor Sierpinskeho kriviek je

$$S:\ A \searrow B \swarrow C \nwarrow D \nearrow \tag{3.21}$$

a rekurzívne vzorce sú:

$$\begin{aligned} A&:\ A \searrow B \Rightarrow D \nearrow A \\ B&:\ B \swarrow C \Downarrow A \searrow B \\ C&:\ C \nwarrow D \Leftarrow B \swarrow C \\ D&:\ D \nearrow A \Uparrow C \nwarrow D \end{aligned} \tag{3.22}$$

(Dvojité šípky znamenajú dvojnásobnú jednotkovú dĺžku.)

Ak použijeme tie isté primitívne operácie kreslenia ako v prípade Hilbertových kriviek, môžeme uvedenú rekurzívnu schému bez ťažkosti transformovať (priamo alebo nepriamo) na rekurzívny algoritmus:

```
procedure A(i: integer);
begin if i > 0 then
      begin A(i − 1); x: = x + h; y: = y − h; plot;
            B(i − 1); x: = x + 2 * h; plot;
            D(i − 1); x: = x + h; y: = y + h; plot;          (3.23)
            A(i − 1)
      end
end
```

Táto procedúra je odvodená z prvého riadku rekurzívnej schémy (3.22). Procedúry, ktoré zodpovedajú obrazcom B, C, D, sa odvodia podobným spôsobom. Hlavný program je vytvorený v súlade so schémou (3.21). Jeho úlohou je priradenie začiatočných hodnôt súradniciam kreslenia a určenie jednotkovej dĺžky čiary h na základe rozmerov kresliaceho papiera. Hlavný program je uvedený ako program 3.2. Výsledok jeho realizácie (pre $n = 4$) možno vidieť na *obr. 3.7*. Poznamenávame, že krivka S_0 nie je nakreslená.

PPROGRAM 3.2. *Sierpinskeho krivky*

```
program Sierpinski (pf, output);
{kresli Sierpinskeho krivky rádu 1 až n}
const n = 4; h0 = 512;
```

```
var i, h, x, x0, y, y0: integer;
    pf: file of integer {súbor obsahujúci príkazy pre
                         kresliace zariadenie}
procedure A(i: integer);
begin if i > 0 then
    begin A(i − 1); x: = x + h; y: = y − h; plot;
          B(i − 1); x: = x + 2 * h; plot;
          D(i − 1); x: = x + h; y: = y + h; plot;
          A(i − 1)
    end
end;
procedure B(i: integer);
begin if i > 0 then
    begin B(i − 1); x: = x − h; y: = y − h; plot;
          C(i − 1); y: = y − 2 * h; plot;
          A(i − 1); x: = x + h; y: = y − h; plot;
          B(i − 1)
    end
end;
procedure C(i: integer);
begin if i > 0 then
    begin C(i − 1); x: = x − h; y: = y + h; plot;
          D(i − 1); x: = x − 2 * h; plot;
          B(i − 1); x: = x − h; y: = y − h; plot;
          C(i − 1)
    end
end;
procedure D(i: integer);
begin if i > 0 then
    begin D(i − 1); x: = x + h; y: = y + h; plot;
          A(i − 1); y: = y + 2 * h; plot;
          C(i − 1); x: = x − h; y: = y + h; plot;
          D(i − 1)
    end
end;
begin startplot;
```



```
    i: = 0; h: = h0 div 4; x0: = 2 * h; y0: = 3 * h;
    repeat i: = i + 1; x0: = x0 − h;
           h: = h div 2; y0: = y0 + h;
           x: = x0; y: = y0; nastavplotter;
           A(i); x: = x + h; y: = y − h; plot;
           B(i); x: = x − h; y: = y − h; plot;
           C(i); x: = x − h; y: = y + h; plot;
           D(i); x: = x + h; y: = y + h; plot;
    until  i = n;
    endplot
end.
```

Uvedené dva príklady presvedčivo dokumentujú vhodnosť použitia rekurzie. Správnosť programov možno jednoducho odvodiť na základe ich štruktúry a rekurzívnych vzorcov. Navyše použitie explicitného parametra úrovne i v súlade so schémou (3.5) zaručuje ukončenie algoritmu, pretože hĺbka rekurzie nemôže byť nikdy väčšia, ako je hodnota konštanty n. Programy, ekvivalentné s uvedenými dvoma rekurzívnymi programami, ktoré by explicitne nevyužívali rekurziu, by boli dosť ťažkopádne a nezrozumiteľné. Čitateľovi odporúčame, aby sa sám presvedčil o správnosti tohto tvrdenia, a to tým, že sa pokúsi porozumieť programu, ktorý je uvedený v [3-3].

3.4 ALGORITMY PREHĽADÁVANIA S NÁVRATOM

Zvlášť zaujímavý, ale predovšetkým zložitý okruh problémov, predstavuje v programovaní riešenie všeobecných problémov. Úlohou je nájdenie a zostrojenie takých algoritmov, ktoré by riešili dané špecifické problémy, nie však podľa pevne stanovených pravidiel výpočtu, ale na základe metódy pokusov a chýb. Vo všeobecnosti sa celý proces pokusov a chýb rozkladá na niekoľko čiastkových úloh, ktoré sa veľmi často a najprirodzenejšie vyjadrujú v rekurzívnych terminoch a spočívajú v preskúmaní konečného počtu podúloh. Všeobecne sa môžeme na celý proces pozerať ako na neustále opakovanie pokusov alebo vyhľadávaní, ktorými sa postupne vybuduje a skúma (zmenšuje) strom pod-

úloh. Tento vyhľadávací strom rastie pri mnohých riešeniach problémov veľmi rýchlo, obyčajne exponenciálne, v závislosti od daného parametra. Podľa toho sa zvyšujú i vyhľadávacie náklady. Často sa dá vyhľadávací strom redukovať pomocou použitia heuristík, čím je možné znížiť zložitosť výpočtu na únosnú hranicu.

Naším cieľom však nie je preberanie heuristických pravidiel. Hlavným zameraním tejto kapitoly je všeobecný princíp delenia úloh, riešiacich daný problém, na podúlohy a aplikácia rekurzie v rámci ich riešenia. Začneme tým, že základnú techniku predvedieme na dobre známom príklade, založenom na pohybe koňa po šachovnici.

Majme šachovnicu s rozmermi $n \times n$, t.j. celkový počet políčok je n^2. Kôň, ktorý má pohyby predpísané pravidlami šachu, sa na šachovnici nachádza v pozícii určenej začiatočnými súradnicami x_0, y_0. Nájdime pokrytie celej šachovnice, ak existuje čo len jediné, t.j. vypočítajme dráhu $n^2 - 1$ takých pohybov, pomocou ktorých sa kôň dostane do každého políčka šachovnice iba jediný raz.

Zvyčajnou metódou redukcie uvedeného problému pokrytia n^2 políčok je jeho zámena za nasledujúci problém: vykonať ďalší ťah, alebo zistiť, že takýto ťah už nie je možný. Definujme preto algoritmus, ktorým sa zisťuje možnosť vykonania ďalšieho ťahu. Jeho prvé priblíženie uvádza algoritmus (3.24).

```
procedure skús ďalší ťah;
begin inicializuj výber ťahov;
  repeat vyber ďalšieho kandidáta zo zoznamu ďalších ťahov;
    if prijateľný then
    begin zaznamenaj ťah;                                    (3.24)
      if šachovnica nie je plná then
      begin skús ďalší ťah;
        if neúspešný then vymaž predchádzajúci záznam
      end
    end
  until (ťah bol úspešný) ∨ (nie sú ďalší kandidáti)
end
```

Ak chceme byť pri opise tohto algoritmu presnejší, musíme urobiť určité rozhodnutia, týkajúce sa reprezentácii údajov. Šachovnicu zo-

brazíme najprirodzenejším spôsobom prostredníctvom matice, ktorú označíme symbolom h. Hodnoty indexov budú definované v rozsahu $1..n$. Zavedieme tiež typ indexu:

```
type index = 1..n;                                           (3.25)
var h: array [index, index] of integer
```

Rozhodnutie reprezentovať každé pole šachovnice prostredníctvom celého čísla, namiesto boolovskej hodnoty označujúcej obsadenosť príslušného poľa, sme uskutočnili na základe toho, že chceme mať zaznamenaný prístup pri obsadzovaní šachovnice. Zvyčajne volíme takúto stratégiu:

$h[x, y] = 0$; v poli $\langle x, y \rangle$ kôň ešte nebol

$h[x, y] = i$; v poli $\langle x, y \rangle$ bol kôň v i-tom ťahu $(1 \leq i \leq n^2)$ (3.26)

Naše ďalšie úvahy sa budú týkať výberu vhodných parametrov, na základe ktorých stanovíme začiatočné podmienky pre ďalší ťah a podáme informáciu o jeho úspešnosti (či neúspešnosti). Začiatočné podmienky pre ďalší ťah jednoducho určíme zo súradníc x, y, špecifikujúcich pozíciu, z ktorej sa má vykonať ďalší ťah, a špecifikáciou čísla tohto ťahu (pre záznamové účely). Informáciu o úspešnosti vykonaného ťahu môžeme podávať pomocou boolovského výstupného parametra q, ktorý v prípade, že má hodnotu true, znamená úspešný ťah, v opačnom prípade (q = false) znamená neúspešnosť vykonaného ťahu.

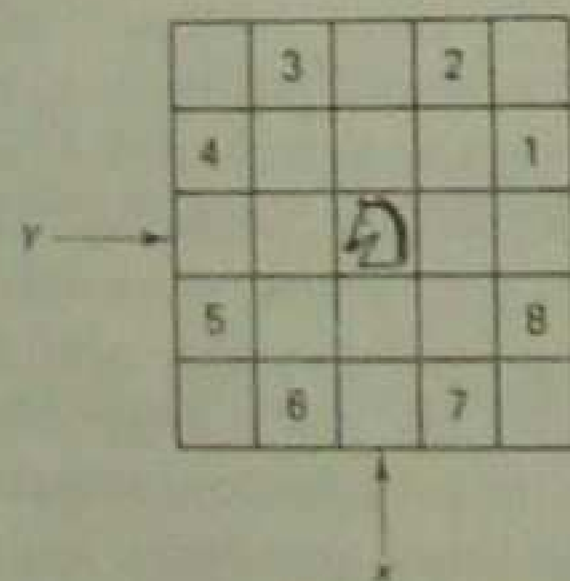
Ktorý príkaz alebo podmienku v algoritme (3.24) už môžeme na základe uvedených rozhodnutí zjemniť? Istotne to môže byť podmienka „šachovnica nie je plná“, ktorú môžeme nahradiť výrazom $i < n^2$. Navyše ak zavedieme dve lokálne premenné u, v, označujúce súradnice možnej cieľovej pozície ťahu koňa (určených na základe pravidla pohybu šachového koňa), tak predikát „prijateľný“ môžeme vyjadriť pomocou logickej kombinácie podmienok: nová cieľová pozícia leží na šachovnici, t.j. $1 \leq u \leq n$ a $1 \leq v \leq n$, resp. príslušné pole na šachovnici ešte nebolo navštívené, t.j. $h[u, v] = 0$. Operáciu zaznamenania správneho ťahu môžeme vyjadriť priraďovacím príkazom $h[u, v] := i$. Zrušenie takéhoto záznamu urobíme priradením $h[u, v] := 0$. Ak ešte zavedieme lokálnu premennú $q1$, ktorú použijeme ako výstupný parameter

rekurzívneho volania uvedeného algoritmu, tak touto premennou môžeme nahradiť podmienku „ťah bol úspešný".

Na základe týchto úvah môžeme formulovať prvú zjemnenú verziu nášho algoritmu:

```
procedure vyskúšaj (i: integer; x, y: index; var q: boolean);
var u, v: integer; q1: boolean;
begin inicializuj výber ťahov;
  repeat nech u, v sú súradnice ďalšieho ťahu, definovaného na základe
        pravidiel šachovej hry;
     if (1 ≦ u ≦ n) ∧ (1 ≦ v ≦ n) ∧ (h[u, v] = 0) then
     begin h[u, v] := i;                                            (3.27)
        if i < sqr(n) then
        begin vyskúšaj (i + 1, u, v, q1);
          if ¬ q1 then h[u, v] := 0
        end else q1 := true
     end
  until q1 ∨ (nie sú ďalší kandidáti);
  q := q1
end
```

Ďalším krokom zjemňovacieho procesu sa dostaneme k programu, vyjadrenému úplne v termínoch našej základnej programovacej notácie. Uvedomme si, že až doteraz sa program vyvíjal úplne nezávisle od zákonov pohybu šachového koňa. Odloženie úvah, týkajúcich sa detailov problému, bol úmyselný. Teraz však nadišiel čas, aby sme vzali do úvahy i tieto detaily.



Obr. 3.8. Osem možných ťahov šachového koňa

Ak máme danú dvojicu začiatočných súradníc ⟨x, y⟩, tak existuje osem možných súradníc ⟨u, v⟩ cieľovej pozície ťahu šachového koňa. Tieto sú očíslované od 1 po 8, ako možno vidieť na *obr. 3.8*.

Jednoduchou metódou, pomocou ktorej sa dajú určiť súradnice u, v, je pripočítanie (rozdielov súradníc) k súradniciam x, y. Tieto rozdiely môžu byť uložené buď v dvojrozmernom poli vo forme príslušných dvojíc rozdielov, alebo v jednorozmernom poli vo forme jednoduchých rozdielov. Zvoľme druhú alternatívu. Nech sú spomenuté polia, označené symbolmi a, b, vhodne inicializované. Pomocou indexu k budeme číslovať „ďalších kandidátov". (Pozri program 3.3.)

PROGRAM 3.3. *Pohyb šachového koňa*

```
program Pohybkoňa (output);
const n = 5; nsq = 25;
type index = 1..n;
var i, j: index;
    q: bollean;
    s: set of index;
    a, b: array [1..8] of integer;
    h: array [index, index] of integer;
procedure vyskúšaj (i: integer; x, y: index; var q: boolean);
  var k, u, v: integer; q1: boolean;
begin k := 0;
  repeat k := k + 1; q1 := false;
        u := x + a[k]; v := y + b[k];
     if (u in s) ∧ (v in s) then
     if h[u, v] = 0 then
     begin h[u, v] := i;
        if i < nsq then
        begin vyskúšaj (i + 1, u, v, q1);
          if ¬ q1 then h[u, v] := 0
        end else q1 := true
     end
  until q1 ∨ (k = 8);
  q := q1
end {vyskúšaj}
```

```
begin s: = [1, 2, 3, 4, 5];
      a[1]: = 2;    b[1]: = 1;
      a[2]: = 1;    b[2]: = 2;
      a[3]: = −1; b[3]: = 2;
      a[4]: = −2; b[4]: = 1;
      a[5]: = −2; b[5]: = −1;
      a[6]: = −1; b[6]: = −2;
      a[7]: = 1;    b[7]: = −2;
      a[8]: = 2;    b[8]: = −1;
   for i: = 1 to n do
      for j: = 1 to n do h[i, j]: = 0;
   h[1, 1]: = 1; vyskúšaj (2, 1, 1, q);
   if q then
      for i: = 1 to n do
      begin for j: = 1 to n do write (h[i, j]: 5);
         writeln
      end
   else writeln ('niet riešenia')
end.
```

Začiatočné volanie rekurzívnej procedúry sa uskutoční s parametrami, ktorých skutočné hodnoty sa rovnajú súradniciam x_0, y_0, určujúcim štartovaciu pozíciu šachového koňa. Tejto pozícii priradíme hodnotu 1; všetky ostatné pozície označíme ako neobsadené (priradíme im hodnotu 0).

$$H[x_0, y_0] := 1;\ \text{vyskúšaj}\,(2, x_0, y_0, q)$$

Ďalší detail, ktorý nemôžeme prehliadnuť, sa týka premennej $h[u, v]$. Táto existuje iba v prípade, ak sa hodnoty súradníc u, v nachádzajú v intervale $1 \ldots n$. Podobne výraz v (3.27), ktorým sme nahradili podmienku „prijateľný" v algoritme (3.24), je korektný iba v tom prípade, ak jeho prvé dva termy sú pravdivé. Správna formulácia je uvedená v programe 3.3. V ňom je navyše dvojitá relácia $1 \leq u \leq n$ nahradená výrazom u **in** $[1, 2, \ldots, n]$, ktorý je pre dostatočne malé hodnoty n oveľa efektívnejší (pozri odsek 1.10.3). V *tab. 3.1* sú uvedené výsledné riešenia, ku ktorým sme dospeli zo začiatočných pozícií ⟨1, 1⟩, ⟨3, 3⟩ pre $n = 5$ a ⟨1, 1⟩ pre $n = 6$.

Tri možné trasy šachového koňa *Tabuľka 3.1*

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

Ďalšie zjednodušenie programu dosiahneme tým, že výstupný parameter q a lokálnu premennú $q1$ nahradíme globálnou premennou.

Čo môžeme vyabstrahovať z uvedeného príkladu? Ktorú zo schém algoritmov, riešiacich triedu podobných problémov, tento príklad najviac pripomína? Čo je z neho pre nás poučné? Podstatnou charakteristikou uvedeného riešenia je skutočnosť, že jednotlivé kroky, na základe ktorých sme postupovali k finálnemu riešeniu, sa metódou pokusov a chýb najprv preskúmali a vzápäti zaznamenali. Umožnilo nám to (v prípade zistenia „slepej uličky", ktorá neviedla k riešeniu) vymazať príslušný záznam o tomto kroku a vrátiť sa na predchádzajúcu pozíciu. Takáto stratégia sa nazýva prehľadávanie s návratom (backtracking). Všeobecnú schému (3.28) získame z algoritmu (3.24) za predpokladu, že počet kandidátov prichádzajúcich do úvahy je v každom kroku konečný.

```
procedure vyskúšaj;
begin inicializuj výber kandidátov;
   repeat vyber ďalšieho kandidáta;
      if prijateľný then
      begin zaznamenaj ho;                                  (3.28)
            if riešenie nie je úplné then
            begin vyskúšaj ďalší krok;
```

```
            if neúspešný then zruš záznam
        end
    end
  until úspešný ∨ nie sú ďalší kandidáti
end
```

V skutočných programoch sa môžu, samozrejme, použiť rôzne modifikácie základnej schémy algoritmu (3.28). Pomerne často sa používa explicitný parameter úrovne, označujúci hĺbku rekurzie, ktorý umožňuje stanoviť jednoduchú podmienku ukončenia rekurzie.

Navyše ak je počet kandidátov, ktorých treba preskúmať v rámci každého kroku algoritmu, pevný, povedzme m, možno použiť odvodenú schému (3.29). Jej začiatočné volanie sa realizuje príkazom vyskúšaj (1).

```
procedure vyskúšaj (i: integer);
  var k: integer;
begin k := 0;
  repeat k := k + 1; vyber k-tého kandidáta;
    if prijateľný then
    begin zaznamenaj ho;                                    (3.29)
        if i < n then
        begin vyskúšaj (i + 1);
            if neúspešný then zruš záznam
        end
    end
  until úspešný ∨ (k = m)
end
```

Zvyšok tejto kapitoly je venovaný tvorbe algoritmov k ďalším trom príkladom. Tieto predstavujú ďalšie tri konkrétne verzie abstraktnej schémy (3.29). Ich zaradenie do tejto kapitoly má opätovne potvrdiť vhodnosť a efektívnosť používania rekurzie.

3.5 PROBLÉM ÔSMICH DÁM

Problém ôsmich dám je dobre známy príklad použitia metódy pokusov a chýb a algoritmov prehľadávania s návratom. Preskúmal ho C. F. Gauss v roku 1850, ale jeho riešenie nedoviedol do úspešného konca. Nie je to nič prekvapujúce, veď charakteristickou črtou týchto problémov je, že vzdorujú analytickému spôsobu riešenia. Namiesto neho vyžadujú veľké množstvo exaktnej práce, trpezlivosti a presnosti. Vieme, že tieto schopnosti sú typické pre počítače. Preto môžeme konštatovať, že uvedené algoritmy získali na dôležitosti práve vďaka vzniku počítačov. Navyše počítače sú schopné riešiť spomenuté problémy oveľa efektívnejšie ako ľudia, ba aj géniovia.

Problém ôsmich dám je definovaný nasledujúcim spôsobom (pozri aj [3-4]): Osem dám treba rozmiestniť na šachovnici takým spôsobom, aby žiadna z nich neohrozovala niektorú z ostatných figúrok (v zmysle pravidla pohybov šachovej dámy).

Ak použijeme schému (3.29) ako vzor štruktúry algoritmu, dostaneme veľmi ľahko nasledujúcu hrubú verziu algoritmu uvedeného problému:

```
procedure vyskúšaj (i: integer);
begin
  inicializuj výber pozícií pre i-tú dámu;
  repeat uskutočni ďalší výber;
      if bezpečný then
      begin umiestni dámu;                                  (3.30)
          if i < 8 then
          begin vyskúšaj (i + 1);
              if neúspešný then odstráň dámu
          end
      end
  until úspešný ∨ niet viac pozícií
end
```

Aby sme boli schopní pokračovať, musíme urobiť určité závažné rozhodnutia týkajúce sa reprezentácií údajov. Vzhľadom na to, že poznáme pravidlá šachovej hry, vieme, že dáma buď ohrozuje figúrky,

ktoré sa nachádzajú buď v tom istom stĺpci, riadku alebo na tej istej diagonále šachovnice. Z toho vyplýva, že každý stĺpec šachovnice môže obsahovať iba jednu dámu, čím sa problém voľby pozície i-tej dámy môže obmedziť na i-tý stĺpec. Parametrom i budeme preto označovať index stĺpca šachovnice. Druhý index j bude určovať príslušný riadok šachovnice a bude slúžiť na výber jednej z ôsmich možných pozícií.

Otvorenou ostala ešte otázka reprezentácie ôsmich dám na šachovnici. Prirodzenou voľbou by mohla byť opäť štvorcová matica, ale ľahko zistíme, že takáto reprezentácia prináša pomerne ťažkopádne operácie kontroly prípustných pozícií. To je pre nás, samozrejme, neželateľný stav, pretože uvedené operácie sú najčastejšie aktivované. Preto zvolíme takú reprezentáciu, ktorá umožní čo najjednoduchšie vykonávanie uvedených kontrol. Najvýhodnejším sa ukazuje čo najpriamejšie zobrazenie tej informácie, ktorá je skutočne podstatná a najčastejšie používaná. V našom prípade to nie je pozícia dám, ale informácia o tom, či v danom riadku alebo na diagonále už bola umiestnená dáma alebo ešte nie. (Vieme už, že v každom stĺpci môže byť umiestnená jediná dáma, ak označíme stĺpec symbolom k, tak platí $1 \leqq k \leqq i$.) Z uvedených úvah vyplývajú nasledujúce voľby reprezentácií premenných:

```
var x: array [1..8] of integer;
    a: array [1..8] of boolean;            (3.31)
    b: array [b1..b2] of boolean;
    c: array [c1..c2] of boolean;
```

pričom

- $x[i]$ označuje pozíciu dámy v i-tom stĺpci,
- $a[j]$ znamená, že žiadna dáma sa nenachádza v j-tom riadku,
- $b[k]$ znamená, že žiadna dáma sa nenachádza na k-tej diagonále s orientáciou ↙,
- $c[k]$ znamená, že žiadna dáma sa nenachádza na k-tej diagonále s orientáciou ↘.

Voľba hraníc indexov $b1$, $b2$, $c1$, $c2$ polí b, c je ovplyvnená skutočnosťou, že indexy polí b, c sa vypočítavajú; poznamenávame, že v diagonále s orientáciou ↙ majú všetky pozície rovnaké súčty svojich súradníc i, j, zatiaľ čo pri opačne orientovanej diagonále sú konštantné rozdiely súradníc $i - j$. Vhodné riešenie je uvedené v programe 3.4.

Teda ak už máme definované štruktúry údajov, tak príkaz „umiestni dámu" možno vyjadriť nasledujúcou postupnosťou príkazov:

$$x[i] := j;\ a[j] := \text{false}; \qquad b[i + j] := \text{false};\ c[i - j] := \text{false} \tag{3.32}$$

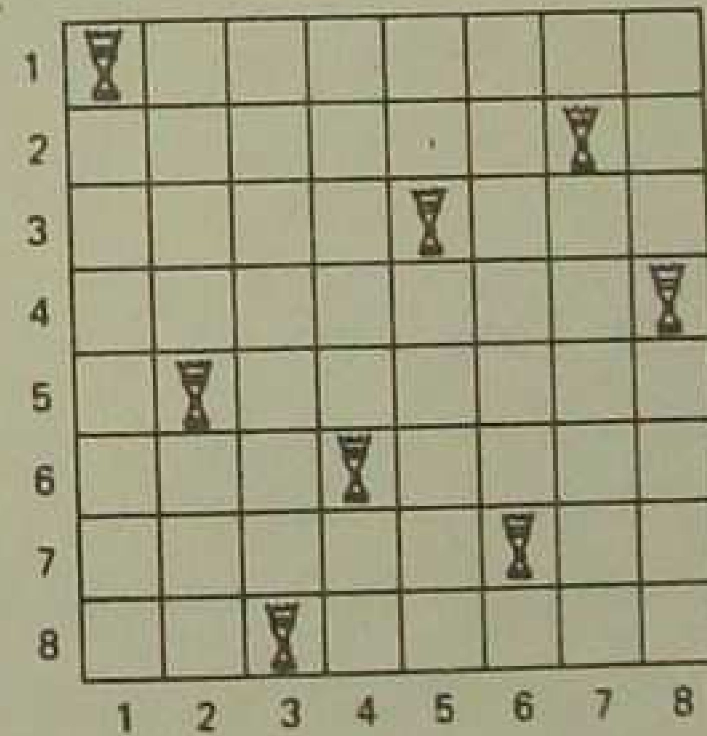
Príkaz „odstráň dámu" môžeme vyjadriť postupnosťou týchto príkazov:

$$a[j] := \text{true};\ b[i + j] := \text{true};\ c[i - j] := \text{true} \tag{3.33}$$

Podmienka bezpečný bude splnená, ak sa bude pozícia $\langle i, j \rangle$ nachádzať v riadku a diagonálach, ktoré sú zatiaľ voľné (reprezentované hodnotou true). Túto skutočnosť môžeme vyjadriť nasledujúcim logickým výrazom:

$$a[j] \wedge b[i + j] \wedge c[i - j] \tag{3.34}$$

Tento výraz súčasne ukončuje vývoj algoritmu riešiaceho problém ôsmich dám. Úplná verzia algoritmu je uvedená v programe 3.4. Vypočítané riešenie je $x = (1, 5, 8, 6, 3, 7, 2, 4)$ a je znázornené na *obr. 3.9*.



Obr. 3.9. Riešenie problému ôsmich dám

PROGRAM 3.4. *Osem dám*

```
program Osemdám1 (output);
{nájde jedno riešenie problému ôsmich dám}
```

```
var i: integer; q: boolean;
    a: array [1 .. 8] of boolean;
    b: array [2 .. 16] of boolean;
    c: array [-7 .. 7] of boolean;
    x: array [1 .. 8] of integer;
procedure vyskúšaj (i: integer; var q: boolean);
  var j: integer;
begin j: = 0;
  repeat j: = j + 1; q: = false;
    if a[j] ∧ b[i + j] ∧ c[i − j] then
    begin x[i]: = j;
      a[j]: = false; b[i + j]: = false; c[i − j]: = false;
      if i < 8 then
      begin vyskúšaj (i + 1, q);
        if ¬ q then
        begin a[j]: = true; b[i + j]: = true; c[i − j]: = true
        end
      end else q: = true
    end
  until q ∨ (j = 8)
end {vyskúšaj};
begin
  for i: = 1 to 8 do a[i]: = true;
  for i: = 2 to 16 do b[i]: = true;
  for i: = −7 to 7 do c[i]: = true;
  vyskúšaj (1, q);
  if q then
    for i: = 1 to 8 do write (x[i]: 4);
  writeln
end.
```

Prv než sa prestaneme zaoberať problémami súviasiacimi so šachovnicou, ukážeme si, ako možno použiť príklad ôsmich dám na ilustráciu dôležitého zovšeobecnenia algoritmu založeného na princípe metódy pokusov a chýb. Zovšeobecnenie, resp. rozšírenie algoritmu predstavuje nájdenie nielen jedného, ale všetkých možných riešení daného problému.

Rozšírenie algoritmu je ľahko uskutočniteľné. Treba si len uvedomiť, že generovanie kandidátov musí postupovať systematicky, aby sa niektorý z nich nevybral dvakrát. Táto vlastnosť algoritmu zodpovedá vyhľadávaniu kandidátov v strome, a to systematickým spôsobom, ktorým sa zaručí, že každý vrchol stromu bude preskúmaný presne jedenkrát. Len čo nájdeme jedno riešenie a patrične ho zaznamenáme, môžeme pristúpiť k ďalšiemu kandidátovi, vybranému systematickým procesom výberu. Všeobecná schéma algoritmu je odvodená opäť zo schémy (3.29) a je zobrazená v procedúre vyskúšaj (3.35).

```
procedure vyskúšaj (i: integer);
  var k: integer;
begin
  for k: = 1 to m do
  begin vyber k-tého kandidáta;
    if prijateľný then                                          (3.35)
    begin zaznamenaj ho;
      if i < n then vyskúšaj (i + 1) else vytlač riešenie;
      zruš záznam
    end
  end
end
```

Poznamenávame, že vzhľadom na zjednodušenie podmienky ukončenia procesu výberu, ktorú sme nahradili jednoduchým termom $k = m$, môžeme príkaz cyklu **repeat** vhodne nahradiť príkazom cyklu **for**. Zaiste prekvapí skutočnosť, že program na vyhľadanie všetkých možných riešení problému ôsmich dám je jednoduchši ako program na zistenie jedného riešenia.

Rozšírená verzia algoritmu, ktorou sa dá zistiť všetkých 92 možných riešení problému ôsmich dám, je uvedená v programe 3.5. V skutočnosti existuje iba 12 zásadne rozdielnych riešení; náš program však nerozoznáva symetrické riešenia.

PROGRAM 3.5. *Osem dám*

```
program Osemdám (output);
var i: integer;
    a: array [1..8] of boolean;
    b: array [2..16] of boolean;
    c: array [−7..7] of boolean;
    x: array [1..8] of integer;
procedure print;
  var k: integer;
begin for k := 1 to 8 do write (x[k]: 4);
    writeln
end {print};
procedure vyskúšaj (i: integer);
  var j: integer;
begin
  for j := 1 to 8 do
   if a[j] ∧ b[i + j] ∧ c[i − j] then
   begin x[i] := j;
     a[j] := false; b[i + j] := false; c[i − j] := false;
     if i < 8 then vyskúšaj (i + 1) else print;
     a[j] := true; b[i + j] := true; c[i − j] := true
   end
end {vyskúšaj};
begin
  for i := 1 to 8 do a[i] := true;
  for i := 2 to 16 do b[i] := true;
  for i := −7 to 7 do c[i] := true;
  vyskúšaj (1)
end.
```

Prvých dvanásť vygenerovaných riešení je uvedených v *tab. 3.2*. Hodnoty symbolu N, ktorý sa nachádza v pravej časti tabuľky, určujú frekvenciu vykonania testu príslušnej pozície na šachovnici. Jej priemerná hodnota pre všetkých 92 riešení je 161.

Dvanásť riešení problému ôsmich dám *Tabuľka 3.2*

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	4	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

3.6 PROBLÉM STABILNÉHO MANŽELSTVA

Predpokladajme, že sú dané dve disjunktné množiny A, B, ktoré majú zhodnú kardinalitu n. Treba nájsť množinu n dvojíc $\langle a, b\rangle$ takých, že a **in** A, resp. b **in** B spĺňajú určité obmedzujúce podmienky. Pre takéto dvojice existuje prirodzene viacero rôznych kritérií; jedno z nich sa nazýva „pravidlo stabilného manželstva".

Predpokladajme ďalej, že A je množina mužov a B množina žien. Každý muž a každá žena si stanovili rôzne požiadavky na svojich partnerov. Keď sa vyberie n takých dvojíc, v ktorých muž a žena nie sú spolu zosobášení, ale obaja by dali svojmu vybratému partnerovi prednosť pred svojím skutočným manželským partnerom, tak takéto priradenie partnerov nazývame nestabilným. Ak takéto dvojice neexistujú, hovoríme o stabilnom zväzku.

Uvedená situácia je charakteristická pre mnohé príbuzné problémy, pri ktorých sa priradenia uskutočňujú na základe určitých požiadaviek, akými sú napr. voľba školy študentmi, voľba odvedených brancov niektorými zložkami ozbrojených síl atď.

Príklad manželstiev je však zvlášť intuitívny; uvedomme si predsa, že vytvorený zoznam požiadaviek partnerov je invariantný a nemení sa

ani po realizácii príslušných priradení. Toto pravidlo zjednodušuje uvedený problém, ale súčasne predstavuje aj skreslenie skutočnej situácie (takémuto javu hovoríme abstrakcia).

Jedným z možných spôsobov, ako nájsť riešenie, je pokúsiť sa popáriť členov oboch množín, jedného za druhým, až kým obidve množiny nebudú prázdne. Keď sa teda pokúsime nájsť všetky stabilné priradenia, môžeme ľahko načrtnúť riešenie, a to tým spôsobom, že použijeme schému (3.25) ako vzor. Nech procedúra vyskúšaj (m) predstavuje algoritmus nájdenia partnerky pre muža m a nech toto vyhľadávanie postupuje podľa zoznamu mužom stanovených predností. Prvú verziu algoritmu založenú na uvedených predpokladoch zobrazuje schéma (3.36).

```
procedure vyskúšaj (m : muž);
  var r : poradie;
begin
  for r := 1 to n do
  begin vyber r-tú prednosť muža m;                          (3.36)
    if prijateľná then
    begin zaznamenaj manželstvo;
      if m nie je posledným mužom then vyskúšaj (succ (m))
      else zaznamenaj stabilnú množinu;
      zruš manželstvo
    end
  end
end.
```

Opäť sme dospeli k situácii, po ktorej nemôžeme ďalej postupovať bez toho, aby sme neurobili isté rozhodnutia ohľadom reprezentácie údajov. Začneme tým, že zavedieme tri skalárne typy, ktorých hodnotami nech sú kvôli jednoduchosti celé čísla z intervalu 1 . . n. I keď tieto tri typy sú formálne zhodné, ich rozlíšenie prostredníctvom rôznych identifikátorov významne prispieva k zlepšeniu zrozumiteľnosti programu. Predovšetkým bude na prvý pohľad jasné, čo príslušná premenná znamená.

```
type muž = 1..n;
     žena = 1..n;                                            (3.37)
     poradie = 1..n;
```


Začiatočné množiny údajov sú reprezentované dvoma maticami, ktoré vyjadrujú požiadavky mužov a žien.

```
var žmp : array [muž, poradie] of žena
    mžp : array [žena, poradie] of muž                       (3.38)
```

Potom premennou $žmp[m]$ označujeme zoznam požiadaviek m-tého muža, $žmp[m][r] = žmp[m, r]$ je žena, ktorá sa nachádza na r-tom mieste poradovníka m-tého muža. Podobne $mžp[ž]$ je zoznam požiadaviek $ž$-tej ženy a $mžp[ž, r]$ je jej r-tý výber.

Výsledok je reprezentovaný takým poľom žien x, že $x[m]$, znamená partnerku pre muža m. Aby sme zachovali symetriu, nazývanú aj rovnaké práva medzi mužmi a ženami, zavedieme ďalšie pole y také, že $y[ž]$ označuje partnera $ž$-tej ženy.

```
var x : array [muž] of žena;
    y : array [žena] of muž                                  (3.39)
```

Je jasné, že premenná y nie je až taká potrebná, pretože reprezentuje informáciu, ktorá už je prítomná, a to v premennej x. Skutočne, vzťahy

$$x[y[ž]] = ž, \quad y[x[m]] = m \tag{3.40}$$

platia pre všetkých mužov m a všetky ženy $ž$, ktorí sú zosobášení. Potom môžeme hodnotu $y[ž]$ zistiť pomocou jednoduchého prehľadania x; pole y však zjavne zvyšuje efektívnosť algoritmu. Informácia reprezentovaná pomocou premenných x a y je potrebná na zistenie stability uvažovanej množiny manželstiev. Pretože táto množina sa vytvára postupným spájaním jednotlivcov a testovaním stability ich manželstva, je potrebné, aby premenné x a y boli definované už predtým, než sú definované ich zložky. Aby sme mohli zaznamenávať informácie o definovaných zložkách, zavedieme boolovské polia

```
voľnýmuž : array [muž] of boolean
voľnážena : array [žena] of boolean                          (3.41)
```

s týmto významom:

¬ *voľnýmuž* [m] znamená, že $x[m]$ je definované
¬ *voľnážena* [$ž$] znamená, že $y[ž]$ je definované

Jednoduchou analýzou navrhovaného algoritmu však rýchlo zistíme, že manželský stav muža sa dá jednoducho určiť pomocou hodnoty m, a to nasledujúcim spôsobom:

$$\neg \mathit{voľnýmuž}[k] \equiv k < m \tag{3.42}$$

Táto skutočnosť nám umožňuje vynechať pole *voľnýmuž*, a tým zjednodušiť, resp. nahradiť pole *voľnážena* poľom *voľná*.

Na základe týchto konvencií môžeme uskutočniť zjemnenie algoritmu, ktoré je uvedené v schéme (3.43). Predikát *prijateľná* môžeme nahradiť konjunkciou premennej *voľná* a funkciou *stabilné*, ktorú vypočítame neskôr.

```
procedure vyskúšaj (m: muž);
  var r: poradie; ž: žena;
begin for r: = 1 to n do
  begin ž: = žmp[m, r];
    if voľná[ž] ∧ stabilné then                                  (3.43)
    begin x[m]: = ž; y[w]: = m; voľná[ž]: = false;
      if m < n then vyskúšaj (succ(m))
               else zaznamenaj stabilnú množinu;
      voľná[ž]: = true
    end
  end
end.
```

Všimnime si veľkú podobnosť tohto riešenia s programom 3.5.

Našou ďalšou podstatnou úlohou bude zjemnenie algoritmu určujúceho stabilitu. Žiaľ, stabilitu nemôžeme reprezentovať takým jednoduchým výrazom, ako to bolo v prípade bezpečnosti pozícií šachovej dámy v programe 3.5. Prvým problémom, ktorý si musíme uvedomiť, je, že stabilita sa, podľa definície, riadi porovnaniami požiadaviek alebo poradí. Poradia mužov a žien však nie sú doteraz v našich množinách údajov nikde explicitne uvedené. Preto ich treba vypočítať. Pochopiteľne, výpočet poradia ženy $ž$ v mysli muža m bude vyžadovať nákladné vyhľadávanie hodnoty $ž$ v poli $žmp[m]$.

Vzhľadom na to, že výpočet stability je veľmi častá operácia, je

rozumné, aby informácia o stabilite bola priamo prístupná. Zavádzame preto ešte dve matice:

```
pmž: array [muž, žena] of poradie;
pžm: array [žena, muž] of poradie                                (3.44)
```

$pmž[m, ž]$ označuje poradie ženy $ž$ v zozname požiadaviek muža m a $pžm[ž, m]$ znamená poradie muža m v zozname ženy $ž$. Samozrejme, že hodnoty týchto pomocných polí sú konštantné a na začiatku sa dajú určiť z hodnôt polí *žmp* a *mžp*.

Proces určenia predikátu *stabilné* pokračuje ďalej presne v súlade s jeho pôvodnou definíciou. Uvedomme si opäť, že našim cieľom je zistiť uskutočniteľnosť manželského zväzku muža m a ženy $ž$, pričom $ž = žmp[m, r]$, t. j. žena $ž$ sa nachádza na r-tej pozícii v zozname požiadaviek muža m. Ak budeme optimitisti, predpokladajme najprv prevládanie stability a až potom začnime skúmať možné zdroje problémov. V čom môžu tieto problémy spočívať? Existujú dve symetrické možnosti:

1. Môže existovať žena $pž$, ktorú by uprednostňoval muž m pred svojou partnerkou $ž$ a ona sama by uprednostňovala muža m pred svojím manželom.

2. Môže existovať muž pm, ktorého by uprednostňovala žena $ž$ pred svojím partnerom m a on sám by uprednostňoval ženu $ž$ pred svojou manželkou.

Zoberme prvý variant možného zdroja problémov. Porovnáme poradia $pžm[pž, m]$ a $pžm[pž, y[pž]]$ pre všetky ženy, ktoré sú u muža m uprednostňované pred ženou $ž$, t. j. pre všetky $pž = žmp[m, i]$, pričom $i < r$. Náhodou vieme, že všetky takéto ženské kanditátky sú už vydaté, pretože ak by bola niektorá z nich ešte slobodná, bol by si ju muž m už vybral. Uvedený proces môžeme formulovať prostredníctvom jednoduchého lineárneho prehľadávania (s znamená stabilitu).

```
s: = true; i: = 1;
while (i < r) ∧ s do                                             (3.45)
  begin pž: = žmp[m, i]; i: = i + 1;
    if ¬ voľná[pž] then s: = pžm[pž, m] > pžm[pž, y[pž]]
  end
```

Pri druhom variante zdroja problémov musíme zistiť všetkých mužských kandidátov *pm*, ktorých uprednostňuje žena *ž* pred svojím partnerom, t.j. skúmanými mužmi sú všetci muži *pm* = *mžp*[*ž*, *i*] pre *i* < *pžm*[*ž*, *m*]. Podobne ako pri prvom variante zdroja problémov treba uskutočniť porovnania medzi poradiami *pmž*[*pm*, *ž*] a *pmž*[*pm*, *x*[*pm*]]. Musíme byť opatrní a nesmieme zabudnúť vynechať porovnanie, ktoré by sa týkalo *x*[*pm*], kde *pm* je stále voľný. Dosiahneme to tak, že zavedieme podmienku *pm* < *m*, pretože vieme, že všetci uprednostňovaní muži pred mužom *m* sú už ženatí. Úplný algoritmus vyjadruje program 3.6. V *tab. 3.3* máme uvedenú množinu vstupných údajov, ktoré reprezentujú polia *žmp* a *mžp*. Deväť vypočítaných riešení problému stabilného manželstva je v *tab. 3.4*.

Ukážka vstupných údajov pre program riešiaci problém stabilného manželstva *Tabuľka 3.3*

Poradie	1	2	3	4	5	6	7	8
Muž č. 1 vyberá ženy	7	2	6	5	1	3	8	4
2	4	3	2	6	8	1	7	5
3	3	2	4	1	8	5	7	6
4	3	8	4	2	5	6	7	1
5	8	3	4	5	6	1	7	2
6	8	7	5	2	4	3	1	6
7	2	4	6	3	1	7	5	8
8	6	1	4	2	7	5	3	8
Žena č. 1 vyberá mužov	4	6	2	5	8	1	3	7
2	8	5	3	1	6	7	4	2
3	6	8	1	2	3	4	7	5
4	3	2	4	7	6	8	5	1
5	6	3	1	4	5	7	2	8
6	2	1	3	8	7	4	6	5
7	3	5	7	2	4	1	8	6
8	7	2	8	4	5	6	3	1

PROGRAM 3.6. *Stabilné manželstvá*

```
program Manželstvo (input, output);
{problém stabilného manželstva}
```



```
const n = 8;
type muž = 1..n; žena = 1..n; poradie = 1..n;
var m: muž; ž: žena; r: poradie;
    žmp: array [muž, poradie] of žena;
    mžp: array [žena, poradie] of muž;
    pmž: array [muž, žena] of poradie;
    pžm: array [žena, muž] of poradie;
    y: array [žena] of muž;
    x: array [muž] of žena;
    voľná: array [žena] of boolean;
  procedure print;
    var m: muž; rm, rž: integer;
  begin rm: = 0; rž: = 0;
    for m: = 1 to n do
    begin write (x[m]: 4);
      rm: = rm + pmž[m, x[m]]; rž: = rž + pžm[x[m], m]
    end;
    writeln (rm: 8, rž: 4);
  end {print};
  procedure vyskúšaj (m: muž);
    var r: poradie; ž: žena;
    function stabilné: boolean;
    var pm: muž; pž: žena;
        i, lim: poradie; s: boolean;
    begin s: = true; i: = 1;
      while (i < r) ∧ s do
      begin pž: = žmp[m, i]; i: = i + 1;
        if voľná[pž] then s: = pžm[pž, m] > pžm[pž, y[pž]]
      end;
      i: = 1; lim: = pžm[ž, m];
      while (i < lim) ∧ s do
      begin pm: = mžp[ž, i]; i: = i + 1;
        if pm < m then s: = pmž[pm, ž] > pmž[pm, x[pm]]
      end;
      stabilné: = s
    end {stabilné};
```

```
begin {vyskúšaj}
  for r: = 1 to n do
  begin ž: = žmp[m, r];
    if voľná[ž] then
      if stabilné then
      begin x[m]: = ž; y[ž]: = m; voľná[ž]: = false;
        if m < n then vyskúšaj (succ (m)) else print;
        voľná[ž]: = true
      end
  end
end }vyskúšaj};
begin {hlavný program}
for m: = 1 to n do
  for r: = 1 to n do
  begin read (žmp[m, r]); pmž[m, r]]: = r
  end;
for ž: = 1 to n do
  for r: = 1 to n do
  begin read (mžp[ž, r]); pžm[ž, mžp[ž, r]]: = r
  end;
for ž: = 1 to n do voľná[ž]: = true;
vyskúšaj (1)
end.
```

Tento algoritmus je zjavne založený na princípoch prehľadávania s návratom. Jeho efektívnosť v prvom rade závisí od úspešnosti vyriešenia schémy redukcie vyhľadávacích stromov. O niečo rýchlejší, no zložitejší a menej prehľadný algoritmus, vytvorili McVitie a Wilson [3-1] a [3-2], ktorí ho navyše rozšírili i pre prípad množín (mužov a žien) nerovnakej veľkosti.

Algoritmy, ktoré sú svojou podstatou podobné uvedeným dvom príkladom a ktoré generujú všetky možné riešenia daného problému (s určitými obmedzeniami), sa často používajú na výber jedného alebo niekoľkých riešení, ktoré sú v istom zmysle optimálne. V našom poslednom príklade by niekoho mohlo napr. zaujímať riešenie, ktoré by

v priemere najviac vyhovovalo mužom alebo ženám, alebo všetkým osobám.

Všimnime si, že *tab. 3.4* uvádza súčty poradí všetkých žien uvedených v zoznamoch predností svojich manželov a obrátene, súčty poradí všetkých mužov uvedených v zoznamoch predností svojich manželiek. Sú to hodnoty

$$rm = \sum_{m=1}^{n} pmž[m, x[m]], \quad rž = \sum_{m=1}^{n} pžm[x[m], m] \tag{3.46}$$

Výsledok riešenia problému stabilného manželstva *Tabuľka 3.4*

		x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	rm	$rž$	c^*
Riešenie	1	7	4	3	8	1	5	2	6	16	32	21
	2	2	4	3	8	1	5	7	6	22	27	449
	3	2	4	3	1	7	5	8	6	31	20	59
	4	6	4	3	8	1	5	7	2	26	22	62
	5	6	4	3	1	7	5	8	2	35	15	47
	6	6	3	4	8	1	5	7	2	29	20	143
	7	6	3	4	1	7	5	8	2	38	13	47
	8	3	6	4	8	1	5	7	2	34	18	758
	9	3	6	4	1	7	5	8	2	43	11	34

*c = počet vyhodnocovaní stability
Riešenie 1 = optimálne riešenie pre mužov.
Riešenie 9 = optimálne riešenie pre ženy.

Riešenie, ktoré obsahuje najmenšiu hodnotu *rm*, sa nazýva mužské optimálne riešenie stability, riešenie s najmenšou hodnotou *rž* sa nazýva ženské optimálne riešenie stability. Vyplýva to z povahy zvolenej stratégie vyhľadávania: optimálne riešenia z hľadiska mužov sa generujú najprv, kým vhodné riešenia pre ženy sa produkujú až na konci. V tomto zmysle možno uvedený algoritmus pokladať za predpojatý voči mužom. To sa dá, samozrejme, rýchlo zmeniť systematickou zámenou úloh mužov a žien, t. j. zámenou premennej *mžp* premennou *žmp* a vzájomnou výmenou premenných *pmž* a *pžm*.

Tento program už nebudeme ďalej rozširovať. Problém zistenia optimálneho riešenia prenecháme ďalšiemu a súčasne poslednému príkladu algoritmov prehľadávania s návratom.

3.7 PROBLÉM OPTIMÁLNEHO VÝBERU

Posledný príklad algoritmu prehľadávania s návratom je logickým rozšírením predchádzajúcich dvoch príkladov, reprezentovaných všeobecnou schémou (3.35). V prvom príklade sme použili princíp prehľadávania s návratom na zistenie jedného riešenia daného problému. Tak to bolo v prípade pohybov šachového koňa alebo v prípade ôsmich dám. Potom sme hľadali všetky možné riešenia daného problému; dokumentuje to prípad ôsmich dám a stabilného manželstva. Teraz sa pokúsime nájsť optimálne riešenie daného problému.

Princíp nájdenia optimálneho riešenia spočíva v tom, že sa snažíme nájsť všetky možné riešenia problému, pričom v priebehu ich generovania si pamätáme konkrétne riešenie, ktoré je z určitého hľadiska optimálne.

Predpokladajme, že optimálnosť je definovaná pomocou nejakej funkcie $f(s)$, ktorej hodnotami sú nejaké kladné čísla. Potom sa algoritmus riešenia dá odvodiť zo schémy (3.35), pričom príkaz vytlač riešenie nahradíme príkazom

if f(riešenie) > f(optimum) **then** *optimum* : = riešenie (3.47)

Premenná *optimum* obsahuje najlepšie riešenie v každom okamihu výpočtu. Je pochopiteľné, že musí byť vhodne inicializovaná; navyše je vhodné zaznamenávať hodnotu funkcie f(*optimum*) prostredníctvom ďalšej premennej, a to z toho dôvodu, aby sme sa vyhli jej častému vypočítavaniu.

Uvažujme o nasledujúcom príklade všeobecného problému nájdenia optimálneho riešenia danej úlohy: vyberieme si významný a často sa vyskytujúci problém nájdenia optimálneho výberu spomedzi danej množiny objektov na základe určitých výberových kritérií. Jednotlivé výbery, ktoré predstavujú prijateľné riešenia, sa vytvárajú postupne, preskúmaním jednotlivých objektov základnej množiny. Procedúra vyskúšaj, ktorá realizuje proces zistenia vhodnosti individuálnych objektov, sa volá rekurzívne (vzhľadom na preskúmanie ďalšieho objektu) dovtedy, kým sa nepreskúmajú všetky objekty základnej množiny.

Poznamenávame, že prieskum každého objektu (v predchádzajúcich príkladoch sme ich nazývali kandidátmi) má dva možné výsledky, a to buď zahrnutie skúmaného objektu do daného výberu alebo jeho vylúčenie. Z toho vyplýva, že použitie príkazov cyklu **repeat** a **for** bude v tomto prípade nevhodné. Namiesto nich uvedieme explicitne obidva možné prípady, ako to znázorňuje algoritmus (3.48). Predpokladáme, že skúmané objekty sú identifikovateľné číslami 1, 2, ..., n.

```
procedure vyskúšaj (i: integer);
begin
1: if zahrnutie je prijateľné then
   begin zahrň i-tý objekt;
      if i < n then vyskúšaj (i + 1) else over optimálnosť
      eliminuj i-tý objekt                                    (3.48)
   end;
2: if vylúčenie je prijateľné then
      if i < n then vyskúšaj (i + 1) else over optimálnosť
end
```

Z tohto algoritmu je jasné, že existuje 2^n množín, ktoré prichádzajú do úvahy. Ďalej z toho vyplýva skutočnosť, že musíme zvoliť vhodné kritériá prijateľnosti, aby sa podstatne znížil počet skúmaných kandidátov. Aby sme lepšie objasnili tento proces, zvolíme si konkrétnejší príklad problému výberu: Predpokladajme, že každý objekt z n-prvkovej množiny objektov $a_1, a_2, \ldots, a_n$ je charakterizovaný svojou váhou w a hodnotou v. Nech je optimálna tá množina, ktorá obsahuje také prvky, že súčet ich hodnôt je najväčší a súčasne súčet ich váh spĺňa určité stanovené kritériá. Toto je súčasne dobre známa skutočnosť pre všetkých cestovateľov, ktorí sú postavení pred problém: Zabaliť si do svojich kufrov veci potrebné na cestu, ktoré treba vybrať spomedzi n možných vecí, a to tak, aby ich celkový počet bol optimálny, ale ich celková váha nebola väčšia ako maximálna prístupná hodnota.

Dospeli sme opäť k situácii, keď musíme rozhodnúť o voľbe reprezentácii pre uvedené skutočnosti pomocou doteraz známych údajových

typov. Naša voľba je v (3.49). Zvolené typy údajov vyplývajú priamo z predošlých algoritmov.

$$
\begin{aligned}
&\textbf{type } \text{index} = 1 \mathinner{..} n; \\
&\qquad \text{objekt} = \textbf{record } w, v\colon \text{integer } \textbf{end} \\
&\textbf{var } a\colon \textbf{array } [\text{index}] \textbf{ of } \text{objekt}; \\
&\qquad limw, totv, maxv\colon \text{integer}; \\
&\qquad s, opts\colon \textbf{set of } \text{index}
\end{aligned} \tag{3.49}
$$

Premenné *limw* a *totv* označujú maximálnu prístupnú váhu a celkovú hodnotu všetkých n objektov. Tieto dve hodnoty sú konštantné počas celého procesu výberu. Premenná s reprezentuje bežný výber objektov, v rámci ktorého je každý objekt reprezentovaný svojím menom (indexom). Premenná *opts* predstavuje optimálny výber, ktorý sa až do tohto okamihu zistil, premenná *maxv* vyjadruje jeho hodnotu.

Aké budú kritériá prijatia objektu do bežného výberu? Ak sa naše úvahy týkajú zahrnutia objektov, tak daný objekt môžeme vybrať vtedy, ak je jeho váha prípustná vzhľadom na stanovenú hodnotu. Ak je neprípustná, môžeme ukončiť skúšanie pridávania ďalších objektov do momentálneho výberu. Ak sa naše úvahy týkajú vylúčenia objektov, tak kritérium výberu, t. j. kritérium pre pokračovanie v procese tvorby bežného výberu, môžeme vyjadriť tak, že celková hodnota, ktorú dostaneme vylúčením uvažovaného objektu, nie je menšia ako optimálna hodnota zistená počas doterajšieho priebehu výpočtu. Ak by totiž táto hodnota bola menšia, ďalšie vyhľadávanie by síce poskytlo nové riešenia, ale nepriviedlo by nás k optimálnemu riešeniu. Preto je ďalšie vyhľadávanie zbytočné. Z uvedených dvoch podmienok môžeme jednoducho určiť dve relevantné veličiny, ktoré treba vypočítať v každom kroku procesu výberu:

1. Celková váha tw výberu s, ktorú sme až do tohto kroku vypočítali.
2. Dosažiteľná hodnota av momentálneho výberu s.

Tieto dve veličiny môžeme vhodne reprezentovať prostredníctvom parametrov procedúry vyskúšaj.

Podmienku „zahrnutie je prijateľné" zo schémy (3.48) môžeme teraz vyjadriť v tvare

$$tw + a[i].w \leq limw \tag{3.50}$$

a príslušnú kontrolu optimálnosti pomocou algoritmu

$$
\begin{aligned}
&\textbf{if } av > maxv \textbf{ then} \\
&\quad \textbf{begin } \{\text{nové optimum, zaznamenaj ho}\} \\
&\qquad opts := s;\ maxv := av \\
&\quad \textbf{end}
\end{aligned} \tag{3.51}
$$

Posledný priraďovací príkaz vychádza zo skutočnosti, že dosažiteľná hodnota je taká, ku ktorej sme dospeli po preskúmaní všetkých n objektov.

Podmienku „vylúčenie je prijateľné" zo schémy (3.48) môžeme vyjadriť výrazom

$$av - a[i].v > maxv \tag{3.52}$$

Pretože táto podmienka sa často používa, nahradíme hodnotu $av - a[i].v$ symbolom $av1$, aby sa znížil počet potrebných výpočtov na minimum.

Úplný program optimálneho výberu je vytvorený podľa schémy (3.48), ktorá bola postupne vylepšovaná a zjemňovaná definíciami a schémami (3.49) až (3.52) a doplnená o príkazy priraďujúce začiatočné hodnoty globálnym premenným.

Všimnime si, ako jednoducho sa dá vyjadriť zahrnutie objektu do

Ukážka výstupu programu pre optimálny výber *Tabuľka 3.5*

Váha	10	11	12	13	14	15	16	17	18	19
Hodnota	18	20	17	19	25	21	27	23	25	24
10	•									
20							•			
30					•		•			
40	•				•		•			
50	•	•		•						
60	•	•	•	•	•					
70	•	•			•		•		•	
80	•	•	•		•		•	•		
90	•	•			•		•		•	•
100	•	•		•	•		•	•	•	
110	•	•	•		•	•	•		•	
120	•	•			•	•	•	•	•	•

množiny *s* a jeho vylúčenie z množiny *s*, ak použijeme príslušné množinové operátory. Výsledky programu 3.7 pre váhové tolerancie v rozsahu od 10 do 120 sú uvedené v *tab. 3.5*.

PROGRAM 3.7. *Optimálny výber*

```
program Výber (input, output);
{nájdenie optimálneho výberu objektu vzhľadom na určité kritérium
 výberu}
const n = 10;
type index = 1 . . n;
     objekt = record v, w: integer end;
var i: index;
    a: array [index] of objekt;
    limw, totv, maxv: integer;
    w1, w2, w3: integer;
    s, opts: set of index;
    z: array [boolean] of char;
procedure vyskúšaj (i: index; tw, av: integer);
  var av1: integer;
begin {vyskúšaj zahrnutie i-tého objektu}
  if tw + a[i] . w ≤ limw then
  begin s: = s + [i];
    if i < n then vyskúšaj (i + 1, tw + a[i] . w, av) else
         if av > maxv then
         begin maxv: = av; opts: = s
         end;
    s: = s − [i]
  end;
    {vyskúšaj vylúčenie i-tého objektu}
    av1: = av − a[i] . v;
    if av1 > maxv then
    begin if i < n then vyskúšaj (i + 1, tw, av1) else
         begin maxv: = av1; opts: = s
         end
    end
end {vyskúšaj};
```



```
begin totv: = 0;
  for i: = 1 to n do
    with a[i] do
    begin read (w, v); totv: = totv + v
    end;
  read (w1, w2, w3);
  z[true]: = '*'; z[false]: = ' ';
  write ('VÁHA');
  for i: = 1 to n do write (a[i] . w: 4);
  writeln; write ('HODNOTA');
  for i: = 1 to n do write (a[i] . v: 4);
  writeln;
  repeat limw: = w1; maxv: = 0; s: = [ ]; opts: = [ ];
    vyskúšaj (1, 0, totv);
    write (limw);
    for i: = 1 to n do write (' ', z[i in opts]);
    writeln; w1: = w1 + w2
  until w1 > w3
end.
```

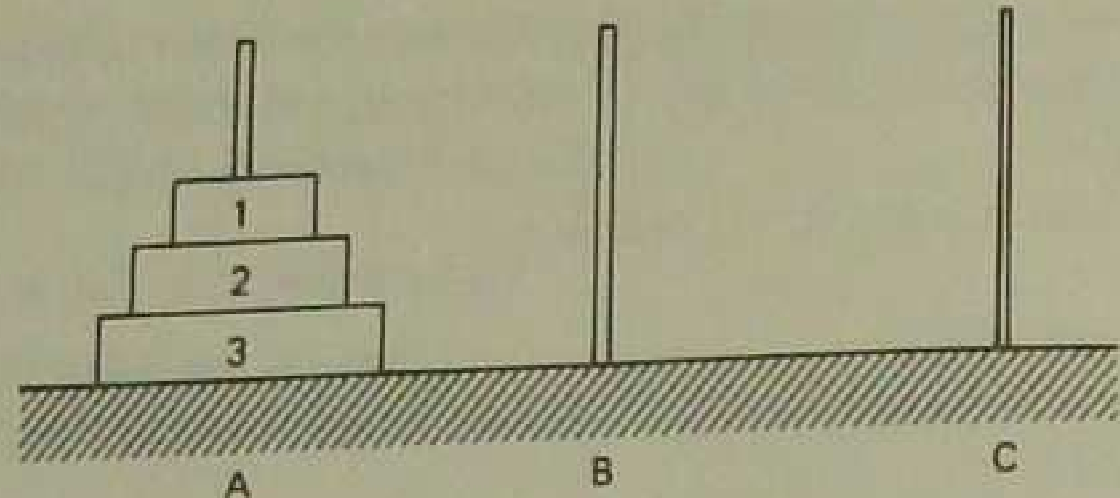
Táto schéma algoritmu prehľadávania s návratom spoločne s faktorom obmedzenia, ktorý redukuje narastanie vyhľadávacieho stromu, je známa aj pod pojmom algoritmus vetvenia s obmedzením.

Cvičenia

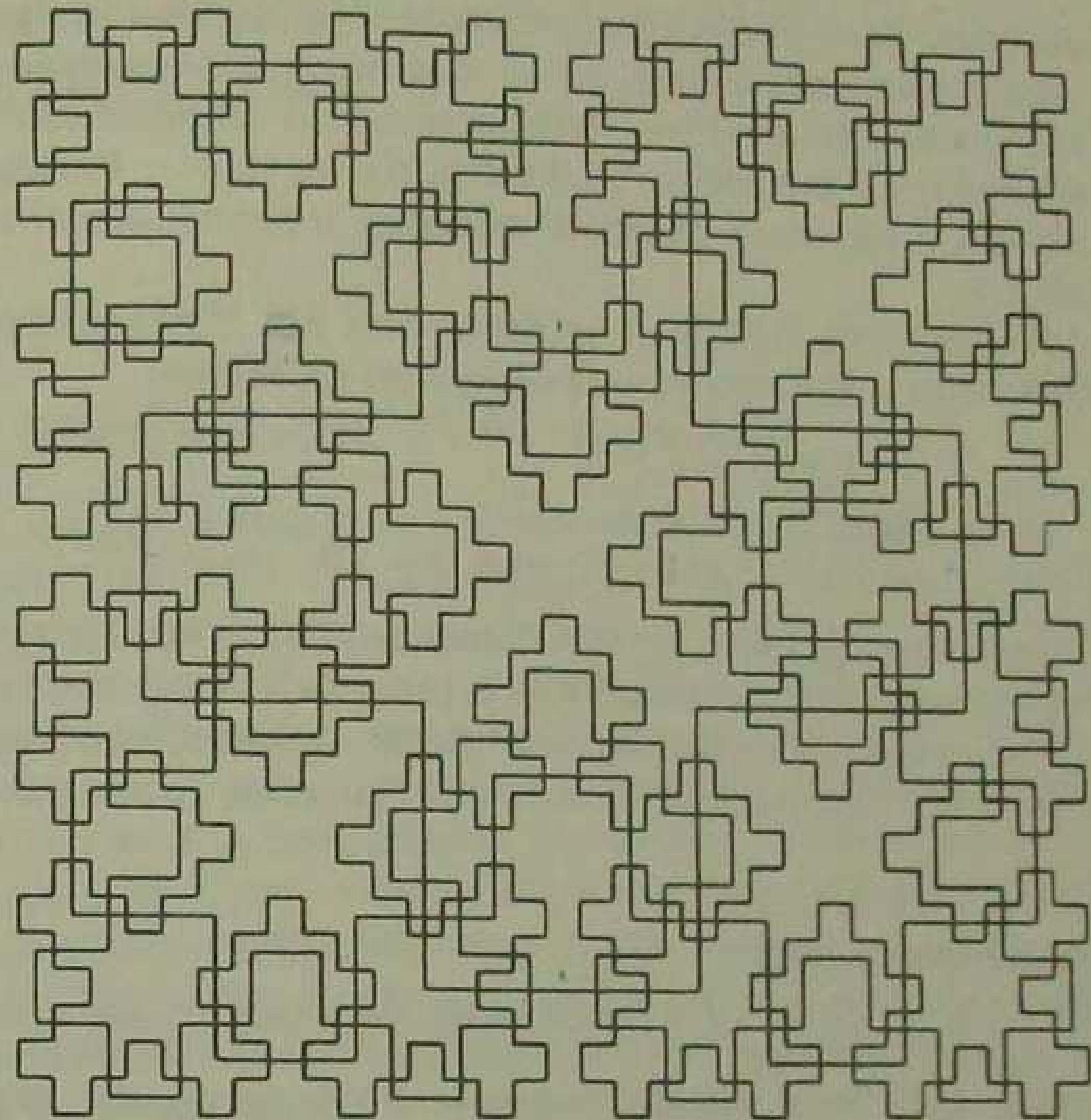
3.1. (*Hanojské veže*) Dané sú tri tyče a n kotúčov (diskov) rôznej veľkosti. Kotúče sa dajú nastoknúť na tyče, a tak je možné vytvárať veže. Nech je spočiatku n kotúčov nastoknutých na tyči A, a to v poradí od najväčšieho po najmenší, ako ukazuje *obr. 3.10* pre $n = 3$.

Premiestnite n kotúčov z tyče A na tyč C tak, aby boli uložené v pôvodnom poradí, ak sú dané tieto obmedzujúce podmienky:

1. V každom kroku sa môže premiestniť iba jediný kotúč z jednej tyče na iný.
2. Väčší kotúč sa nikdy nesmie položiť na menší kotúč.
3. Tyč B sa môže použiť ako pomocný odkladací priestor.



Obr. 3.10. Hanojské veže



Obr. 3.11. *W*-krivky prvého až štvrtého rádu

Vytvorte algoritmus, ktorý by uskutočnil uvedenú úlohu. Poznamenávame, že na vežu sa môžeme pozerať, ako keby pozostávala z jedného kotúča umiestneného na jej vrchole a z veže skladajúcej sa zo zvyšných kotúčov. Napíšte rekurzívnu verziu algoritmu riešiaceho uvedený problém.

3.2. Napíšte procedúru generujúcu všetkých $n!$ permutácií n prvkov $a_1, a_2, \ldots, a_n$ bez prídavného pomocného poľa. Nech sa po vygenerovaní každej ďalšej permutácie vyvolá parametrická procedúra Q, ktorá môže napr. zobraziť práve vygenerovanú permutáciu.

Pomôcka: Problém generovania všetkých permutácií prvkov $a_1, a_2, \ldots, a_m$ považujte za problém skladajúci sa z m podproblémov generovania všetkých permutácií prvkov $a_1, \ldots, a_{m-1}$, za ktorými nasleduje prvok a_m, pričom v rámci i-tého podproblému sa dva prvky a_i, a_m na začiatku vzájomne vymenili.

3.3. Odvoďte rekurzívnu schému z *obr. 3.11*, ktorý je superpozíciou štyroch kriviek W_1, W_2, W_3, W_4. Štruktúra je podobná štruktúre Sierpinskeho kriviek (3.21) a (3.22). Z rekurzívneho obrazca odvoďte rekurzívny program na nakreslenie týchto kriviek.

3.4. Pri riešení problému ôsmich dám sme sa zmienili o tom, že iba 12 z 92 možných riešení uvedeného problému pomocou programu 3.5 je zásadne rozdielnych. Ostatné riešenia získame, keď zoberieme do úvahy symetriu podľa osí alebo stredového bodu. Vytvorte program, ktorý zistí 12 základných riešení problému ôsmich dám. Uvedomte si pritom, že napr. vyhľadávanie v prvom stĺpci môže byť obmedzené na pozície 1 až 4.

3.5. Modifikujte program riešiaci problém stabilného manželstva tak, aby bol schopný zistiť optimálne riešenie (pre mužov alebo pre ženy). Touto modifikáciou sa algoritmus stabilného manželstva, ktorým sme sa zaoberali v článku 3.6, zmení na algoritmus vetvenia s obmedzením, reprezentovaný programom 3.7.

3.6. Istá železničná spoločnosť má na starosti n staníc $S_1, \ldots, S_n$. Rozhodla sa, že zlepší svoje informačné služby pre cestujúcich tak, že zavedie pre nich informačné terminály napojené na počítač. Cestujúci zadá prostredníctvom terminálu svoju východiskovú stanicu S_A a cieľovú stanicu S_B a okamžite obdrží plán vlakových spojení staníc S_A a S_B s optimálnym časom cestovania. Vytvorte

program, ktorý by vypočítal požadované informácie. Predpokladajte, že cestovný poriadok (ktorý je našou bankou údajov) je reprezentovaný vhodnou štruktúrou údajov obsahujúcou časy odchodov (= príchodov) všetkých možných vlakov. Prirodzene, nie všetky stanice sú pospájané priamymi traťami (pozri aj cvičenie 1.8).

3.7. Ackermannova funkcia A je definovaná pre všetky nezáporné celočíselné argumenty m, n nasledujúcim spôsobom:

$$\begin{aligned} &A(0, n) = n + 1 \\ &A(m, 0) = A(m - 1, 1) && (m > 0) \\ &A(m, n) = A(m - 1, A(m, n - 1)) && (m, n > 0) \end{aligned}$$

Navrhnite program, ktorý vypočítava hodnoty $A(m, n)$ bez použitia rekurzie. Pomôckou nech vám je program 2.11, t. j. nerekurzívna verzia algoritmu quicksort. Navrhnite všeobecnú množinu pravidiel pre transformáciu rekurzívnych programov na iteratívne.

Zoznam použitej literatúry

3-1. McVitie, D. G. — Wilson, L. B.: The Stable Marriage Problem, Comm. ACM, 14, No. 7 (1971), s. 486—492.

3-2. McVitie, D. G.: Stable Marriage Assignment for Unequal Sets, BIT, 10 (1970), s. 295—309.

3-3. Space Filling Cueves, or How to Waste Time on a Plotter. Software-Practice and Experience, 1, No. 4 (1971), s. 403—440.

3-4. Wirth, N.: Program Development by Stepwise Refinement. Comm. ACM, 14, No. 4 (1971), s. 221—227.

4 DYNAMICKÉ INFORMAČNÉ ŠTRUKTÚRY

4.1 REKURZÍVNE TYPY ÚDAJOV

V druhej kapitole sme zaviedli základné štruktúry údajov: pole, záznam a množinu. Tieto štruktúry považujeme za základné, v praxi sa vyskytujú najčastejšie a pomocou nich môžeme tvoriť zložitejšie štruktúry. Definovaním typu údajov umožníme pomocou tohto typu definovať určitú premennú. To znamená presne stanoviť rozsah hodnôt, ktoré táto premenná môže nadobúdať a stanoviť počet pamäťových miest potrebných na jej reprezentáciu. Premenné, ktoré sú definované takýmto spôsobom a majú presne definovanú štruktúru, ktorá je nemenná počas celého výpočtu, nazývame *statické*. Pochopiteľne, existuje mnoho problémov, ktoré vyžadujú oveľa zložitejšie informačné štruktúry. Charakteristickou črtou týchto problémov je, že ich štruktúry sa počas výpočtu menia. Preto ich nazývame *dynamické štruktúry*. Prirodzene, elementárne zložky týchto štruktúr sú na určitom stupni detailnosti statické, t. j. sú jedným zo známych základných typov údajov. Táto kapitola je venovaná problematike tvorby, analýzy a používania dynamických informačných štruktúr.

Medzi metódami zavádzania štruktúr do algoritmov a údajov existujú určité analógie. Ako v prípade všetkých analógií aj v tomto prípade sú určité rozdielnosti (v opačnom prípade by predsa išlo o totožnosť). Porovnaním metód štruktúrovania programov a údajov sa tieto odlišnosti objavia.

Základným neštruktúrovaným príkazom je priraďovací príkaz. Spomedzi štruktúr údajov mu zodpovedá neštruktúrovaný skalárny typ. Priraďovací príkaz a skalárny typ predstavujú základné prvky, pomocou ktorých môžeme tvoriť zložitejšie príkazy a typy údajov. Najjednoduchšie štruktúry, ktoré získame pomocou vymenovania alebo postup-

ného zoradenia prvkov, sú zložený príkaz a záznam. Obidve pozostávajú z konečného (obyčajne malého) počtu explicitne vymenovaných prvkov, ktoré môžu byť vzájomne odlišné. V prípade, že všetky prvky štruktúry sú rovnaké (totožné), nie je potrebné vymenovávať ich osobitne, na ich vymenovanie s výhodou použijeme príkaz **for** a štruktúru pole, ktorými najlepšie vyjadríme cyklus, resp. presný počet opakovaného výskytu nejakého prvku. Výber spomedzi dvoch alebo viacerých možností sa dá najvhodnejšie vyjadriť prostredníctvom príkazov vetvenia (podmienených príkazov), akým je napr. príkaz **if**, alebo pomocou príkazu **case**. Týmto príkazom zodpovedá štruktúra záznam s variantmi. Cyklus s neznámym počtom opakovaní (teda vlastne nekonečný cyklus) môžeme vyjadriť pomocou príkazov **while** alebo **repeat**. Zodpovedajúcou štruktúrou údajov je v tomto prípade postupnosť (súbor), ktorá je súčasne najjednoduchšou štruktúrou umožňujúcou konštrukciou typov s nekonečnou kardinalitou.

Zaujímavá je otázka, či existuje (alebo neexistuje) štruktúra údajov, ktorá by zodpovedala príkazu volania procedúry. Prirodzene, najzaujímavejšou a doteraz nezvyčajnou vlastnosťou procedúr je rekurzia. Hodnoty rekurzívneho typu údajov obsahujú jednu alebo viacero zložiek toho istého typu, akého sú samy. Táto vlastnosť rekurzívneho typu predstavuje analógiu k procedúre, ktorá obsahuje jedno alebo viaceré vyvolania seba samej. Podobne ako procedúry aj definície rekurzívnych typov údajov môžu byť buď priamo, alebo nepriamo rekurzívne.

Jednoduchým príkladom objektu, ktorý je vhodne reprezentovaný rekurzívne definovaným typom, je aritmetický výraz, jeden zo základných objektov programovacích jazykov. Rekurzia nám v tomto prípade umožňuje vyjadriť tzv. vkladanie do seba (nesting), t. j. vytvorenie podvýrazov (uzavretých v zátvorkách) ako operandov v rámci výrazov. Bude zrejme účelné, aby sme na tomto mieste definovali výraz.

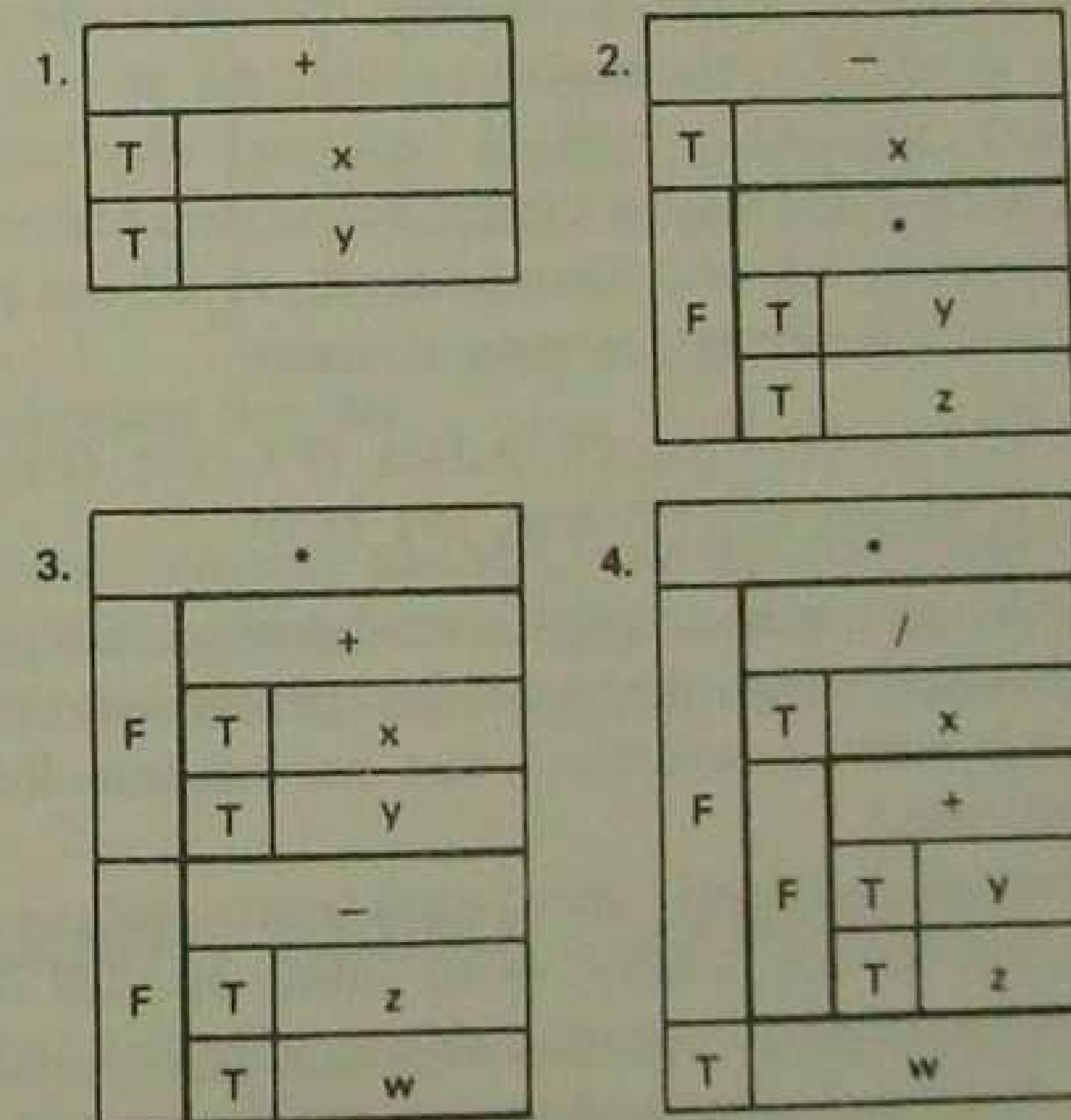
Výraz sa skladá z termu, za ktorým nasleduje operátor a ďalší term. (Dva termy tvoria operandy príslušného operátora.) Term môže byť vyjadrený buď pomocou premennej (reprezentovanej identifikátorom), alebo prostredníctvom výrazu uzavretého v zátvorkách.

Typ údajov, ktorého hodnoty reprezentujú takéto výrazy, môžeme veľmi jednoducho vyjadriť pomocou našich doterajších prostriedkov obohatených o rekurziu:


```
type výraz = record op: operátor;
                    opd1, opd2: term
             end;
type term = record                                  (4.1)
              if t then (id: alfa)
                 else (podvýraz: výraz)
            end
```

Z uvedenej definície vidíme, že každá premenná typu term sa skladá z dvoch zložiek. Prvou zložkou je rozlišovacia zložka označená symbolom t. Druhá zložka môže byť určená na základe hodnoty rozlišovacej zložky. Ak má rozlišovacia zložka hodnotu true, tak je druhou zložkou typu term identifikátor *id*, v opačnom prípade je druhou zložkou *podvýraz*. Majme napr. nasledujúce štyri výrazy:

1. $x + y$
2. $x - (y * z)$ (4.2)
3. $(x + y) * (z - w)$
4. $(x(y + z)) * w$



Obr. 4.1. Ukážka obsadenia pamäti v prípade rekurzívnej štruktúry záznam

Tieto výrazy môžeme zobraziť obrazcami, uvedenými na *obr. 4.1*, ktoré výstižne znázorňujú vloženie výrazov do seba, t. j. ich rekurzívnu štruktúru. Navyše tieto obrazce určujú rozmiestnenie výrazov v pamäti počítača.

Druhým príkladom rekurzívnej informačnej štruktúry môže byť rodokmeň. Nech je tento typ definovaný pomocou štruktúry záznam, pozostávajúcej z mena príslušnej osoby a rodokmeňa jej dvoch rodičov. Definícia vedie k nekonečnej štruktúre. V skutočnosti sú však rodokmene ohraničené, pretože od istej úrovne pra-pra-...-prarodičov (predkov) už ďalšie údaje chýbajú. Túto skutočnosť môžeme vziať do úvahy pri formulovaní definície typu rod (4.3), v ktorej použijeme štruktúru záznam s variantmi.

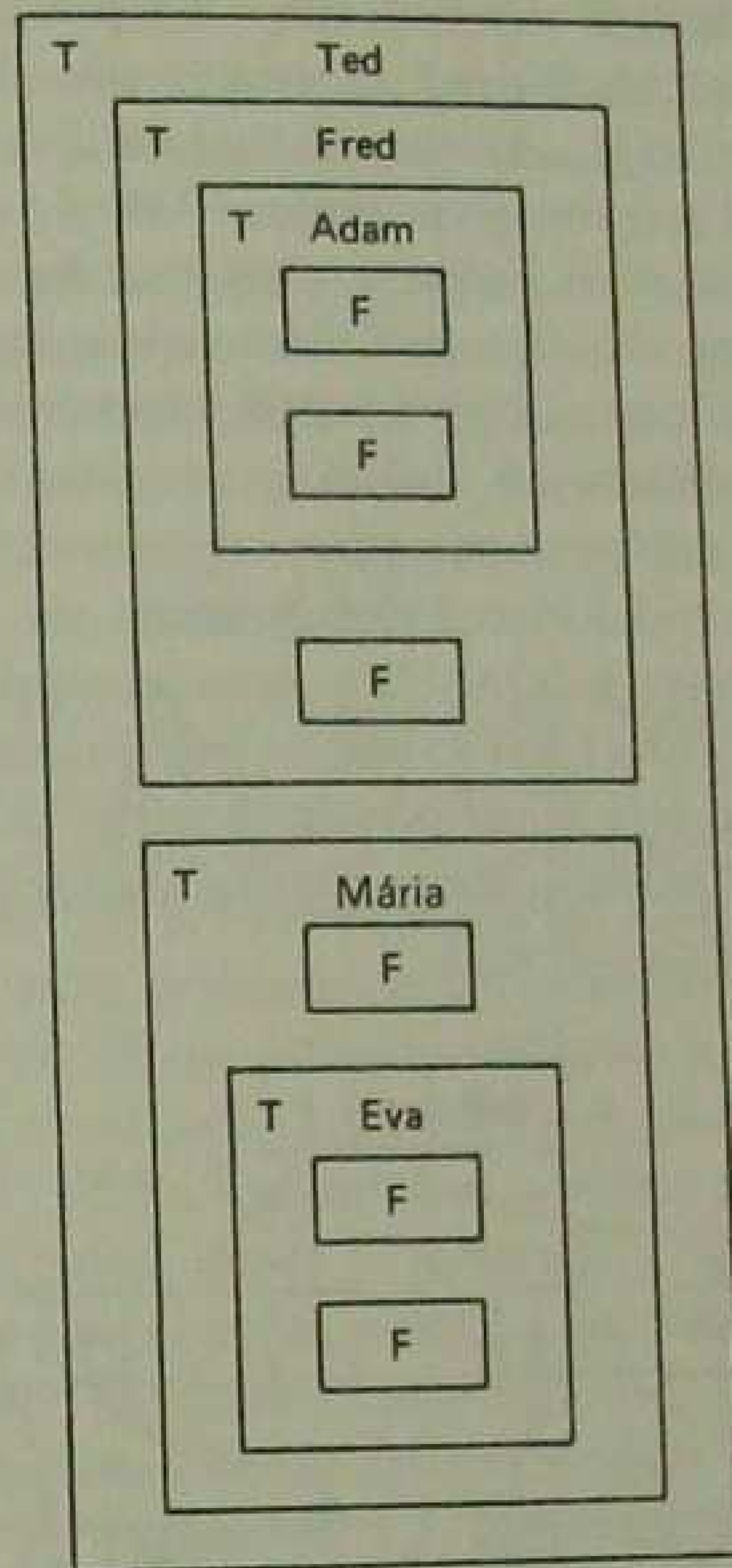
```
type rod = record
             if známy then
                  (meno: alfa;                              (4.3)
                   otec, matka: rod)
           end
```

(Poznamenávame, že každá premenná typu rod obsahuje aspoň jednu zložku, ktorou je rozlišovacia zložka známy. Ak je hodnotou tejto zložky konštanta true, tak premenná typu rod obsahuje ďalšie tri zložky, v opačnom prípade žiadnu.) Jednu konkrétnu hodnotu premennej typu rod znázorňuje *obr. 4.2*. Táto hodnota, vyjadrená prostredníctvom (rekurzívneho) konštruktora typu záznam

x = (T, Ted, (T, Fred, (T, Adam, (F), (F)), (F)),
(T, Mária, (F), (T, Eva, (F), (F)))

nám súčasne poskytuje obraz o možnom rozdelení pamäti potrebnej na reprezentáciu uvedenej premennej typu rod. (Vzhľadom na to, že ide o jednoduchý typ záznam, vynechali sme pri každom konštruktore identifikátor typu rod.)

Predpokladáme, že si čitateľ všimol významnú úlohu s variantmi, najmä však jej rozlišovacej zložky; je to jediný spôsob ohraničenia rekurzívnej štruktúry. Preto je neoddeliteľnou súčasťou každej rekurzívnej definície. Vidíme, že analógia medzi štruktúrovaním programu a údajov je v tomto prípade zvlášť výrazná. Aby sa výpočet takejto



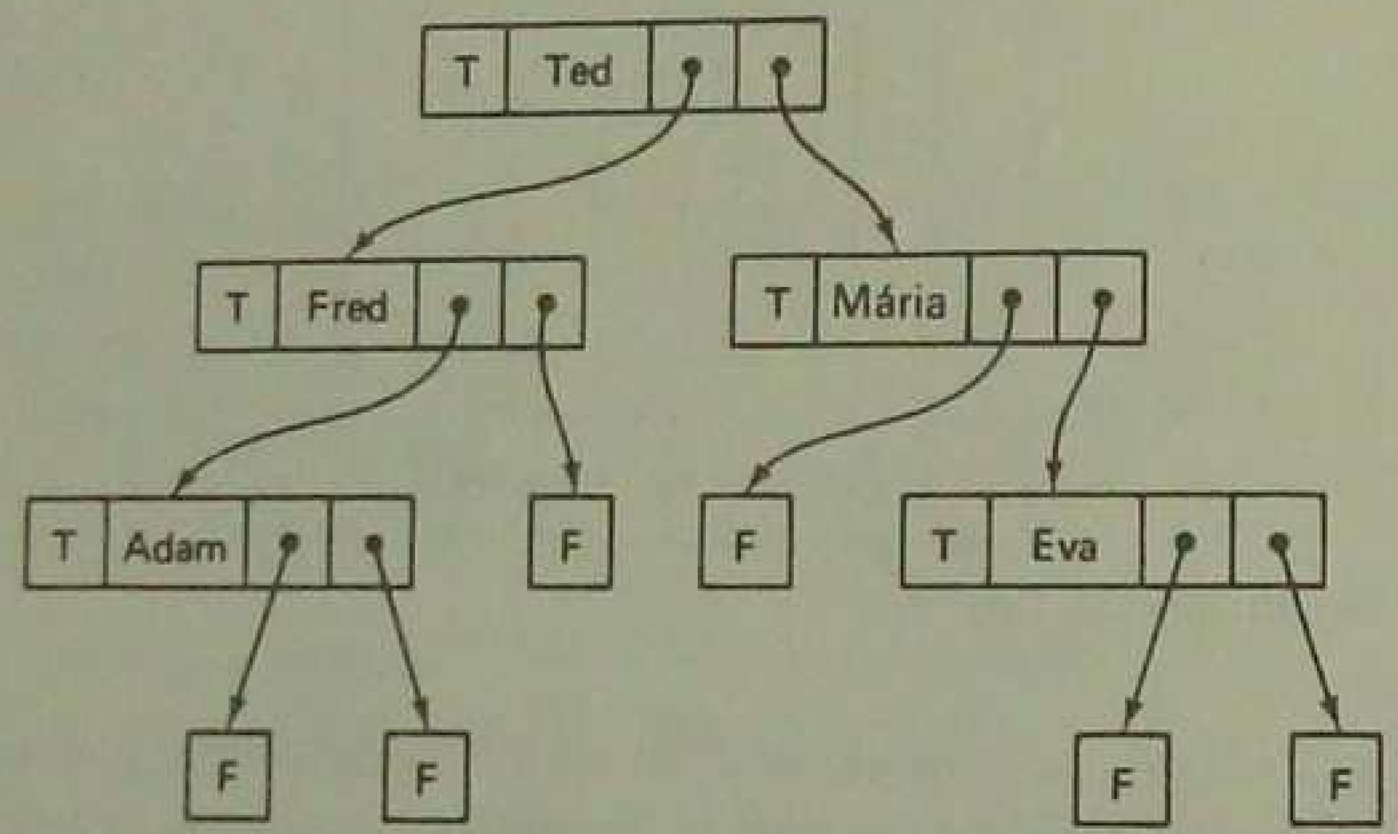
Obr. 4.2. Rodokmeňová štruktúra

procedúry vôbec niekedy ukončil, musí byť podmienený príkaz nevyhnutne súčasťou každej rekurzívnej procedúry. Je zjavné, že ukončenie priebehu výpočtu rekurzívnej procedúry zodpovedá konečnej kardinalite.

4.2 SMERNÍKY ALEBO REFERENCIE

Charakteristickou vlastnosťou rekurzívnych štruktúr, ktorou sa odlišujú od základných štruktúr (pole, záznam, množina), je ich premenlivá veľkosť. Preto nie je možné priradiť rekurzívne definovanej štruktúre

pevný počet pamäťových miest. Z toho ďalej vyplýva, že kompilátor v takomto prípade nie je schopný adresovať jednotlivé zložky rekurzívnych premenných. Najpoužívanejšia technika, ktorá rieši uvedený problém, vychádza z princípov dynamického prideľovania pamäti. Pamäť sa jednotlivým zložkám premennej prideľuje nie v čase prekladu programu, ale v tom okamihu, keď tieto začínajú existovať v priebehu realizácie programu. Kompilátor namiesto pridelenia pamäti potrebnej na reprezentáciu jednotlivých zložiek premennej vyhradí pevnú časť pamäti potrebnú na reprezentáciu adresy dynamicky pridelenej pamäti. Napríklad štruktúra rodokmeň, znázornená na *obr. 4.2*, bude reprezentovaná prostredníctvom jednotlivých, pravdepodobne nespojitých záznamov. Pre každú osobu bude existovať jeden záznam. Tieto osoby sú v rámci štruktúry rod pospájané pomocou adries priradených zložkám otec a matka. Túto situáciu možno najlepšie graficky zobraziť pomocou šípok alebo smerníkov (*obr. 4.3*).



Obr. 4.3. Štruktúra pospájaná pomocou smerníkov

Treba zdôrazniť, že použitie smerníkov v rekurzívnych štruktúrach je jednou z možných realizačných techník. Programátor nemusí o ich existencii nič vedieť. Pamäť sa môže automaticky prideľovať v okamihu prvej referencie novej zložky premennej. Prirodzene, ak je technika používania smerníkov (referencií) explicitne prístupná v rámci programovacieho jazyka, dajú sa vytvárať všeobecnejšie štruktúry údajov než

v prípade obyčajných rekurzívnych definícií. V týchto prípadoch sme schopní definovať nekonečné alebo cyklické štruktúry. Tiež vieme určiť, že sa isté štruktúry budú spoločne používať. Z týchto dôvodov sa vo vyšších programovacích jazykoch udomácnila možnosť explicitného narábania s adresami premenných. Pochopiteľne, že vlastné údaje musia byť v texte programu jasne odlišiteľné od referencií k týmto údajom. Z tohto dôvodu musíme zaviesť nový typ údajov, ktorého hodnotami budú smerníky (referencie) na iné údaje. Použijeme na to nasledujúci spôsob zápisu:

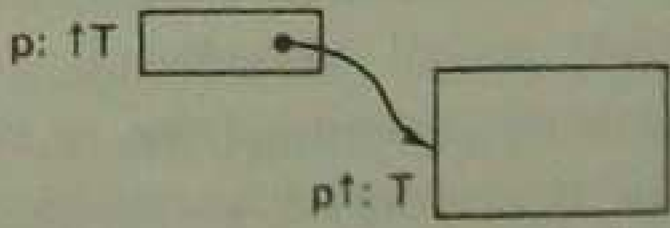
$$\text{type } T_p = \uparrow T \tag{4.4}$$

Uvedená deklarácia typu (4.4) vyjadruje skutočnosť, že hodnoty typu T_p sú smerníkmi na údaje typu T. Teda šípka v definícii (4.4) sa dá slovne vyjadriť ako „smerník na“. Je veľmi dôležité, aby z deklarácie T_p bolo očividné, na aký typ prvkov smerník ukazuje. Hovoríme, že T_p je viazaný na T. Táto väzba umožňuje odlíšiť smerníky použité vo vyšších programovacích jazykoch od adries použitých v asembleroch. Navyše predstavuje dôležitý prostriedok na zvýšenie bezpečnosti a spoľahlivosti pri programovaní (vďaka redundancii použitej notácie).

Hodnoty smerníkových typov sa generujú v okamihu, keď sa konkrétnej premennej pridelí pamäť. Budeme sa držať dohody, že každý takýto moment bude explicitne zachytený a zobrazený. To je však v protiklade s predpokladom, že prvá referencia premennej znamená súčasne i začiatok jej existencie, t.j. automatické pridelenie pamäti potrebnej na jej reprezentáciu. Na tieto účely zavedieme novú procedúru, ktorá sa nazýva new. Ak máme daný smerník p typu T_p, tak prostredníctvom príkazu

$$\text{new}(p) \tag{4.5}$$

vyhradíme pamäť pre premennú typu T. Vytvorí sa tak smerník typu T_p, ktorým je táto nová premenná prístupná, a súčasne sa hodnota tohto smerníka priradí premennej p (*obr. 4.4*). Hodnota smerníka je dostupná



Obr. 4.4. Dynamické pridelenie pamäti pre premennú $p\uparrow$

prostredníctvom smerníkovej premennej p. Premenná, ktorá je prístupná prostredníctvom smerníka p, sa označuje $p\uparrow$. Je ňou dynamicky vytvorená premenná typu T.

Uviedli sme, že rozlišovacia zložka v štruktúre záznam s variantmi má podstatný význam pri zabezpečení konečnej kardinality každého rekurzívneho typu. Najčastejšie sa vyskytujúci prípad je príklad rodokmeňa (pozri *obr. 4.3*). Rozlišovacia zložka môže nadobúdať dve hodnoty (typu boolean), true alebo false. V prípade, že hodnotou tejto zložky je konštanta false, záznamová štruktúra neobsahuje žiadne ďalšie zložky. Túto skutočnosť možno vyjadriť deklaračnou schémou (4.6),

$$\text{type } T = \textbf{record if } p \textbf{ then } S(T) \textbf{ end} \tag{4.6}$$

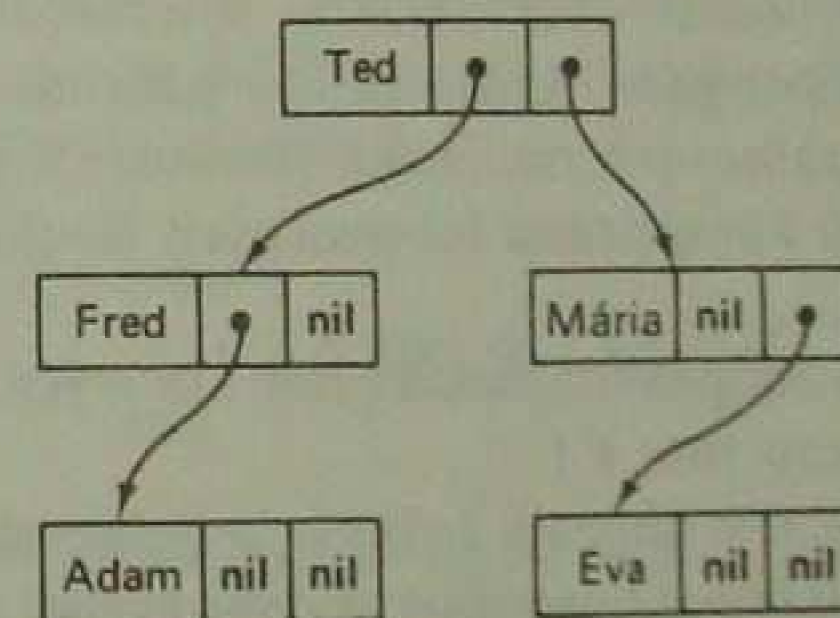
kde $S(T)$ označuje postupnosť definícií jednotlivých zložiek štruktúry typu T. Tým je súčasne vyjadrená aj jej rekurzívnosť. Všetky štruktúry odvodené na základe schémy (4.6) budú tvoriť stromovú štruktúru (alebo štruktúru zoznam), ktorá je podobná štruktúre na *obr. 4.3*. Osobitnou vlastnosťou takejto štruktúry je, že obsahuje iba smerníky na zložky údajov a rozlišovaciu zložku, t.j. žiadne ďalšie podstatné informácie. Implementácia využívajúca smerníky umožňuje jednoduchým spôsobom šetriť pamäť tak, že rozlišovaciu zložku začleníme do samotnej hodnoty smerníka. Zaužívaným riešením je zväčšenie rozsahu hodnôt typu T_p o hodnotu reprezentujúcu smerník, ktorý „nikam neukazuje" (nie je združený s nijakou premennou). Takúto hodnotu označujeme špeciálnym symbolom **nil**. Hovoríme, že konštanta **nil** je automaticky prvkom každého deklarovaného typu smerník. Zväčšenie rozsahu smerníkových hodnôt vyjadruje skutočnosť, že konečné štruktúry môžu byť generované bez prítomnosti podmienok (variantov) v ich (rekurzívnej) deklarácii.

Nové formulácie typov údajov deklarovaných v rámci definícií typov (4.1) a (4.3), založených na explicitných smerníkoch, sú uvedené v schémach (4.7) a (4.8). Všimnime si, že v definícii (4.8), ktorá pôvodne zodpovedala schéme (4.6), sa už nevyskytuje podmienková zložka štruktúry záznam, pretože $p\uparrow.známy$ = false sa dá teraz vyjadriť ako p = **nil**. Zmena identifikátora typu rod na osobu dokumentuje rozdiel v prístupe, ktorý sa dostal do popredia zavedením explicitných smerníkových hodnôt. Namiesto pôvodného uvažovania o štruktúre ako o celku a následného skúmania jej podštruktúry a jej prvkov sústredíme našu pozornosť v prvom rade na zložky štruktúry. Pritom ich vzájomné väzby (reprezentované prostredníctvom smerníkov) nie sú priamo viditeľné zo žiadnej pevne stanovenej deklarácie:

```
type výraz = record op: operátor;
                    opd1, opd2: ↑term
             end;
type term = record                                    (4.7)
              if t then (id: alfa)
                else (podvýraz: ↑výraz)
            end

type osoba = record meno: alfa;
                    otec, matka: ↑osoba               (4.8)
             end
```

Štruktúra údajov, ktorá reprezentuje rodokmeň znázornený na *obr. 4.2* a *obr. 4.3*, v ktorej sú smerníky na neznáme (alebo neexistujúce) osoby označené hodnotou **nil**, je opäť zobrazená na *obr. 4.5*. Pamäťové úspory sú v tomto prípade zjavné.



Obr. 4.5. Štruktúra so smerníkmi **nil**

Predpokladajme, v súlade s *obr. 4.5*, že Fred a Mária sú súrodenci, t.j. majú toho istého otca a matku. Túto situáciu dokážeme jednoducho vyjadriť tak, že nahradíme dve konštanty **nil** v príslušných zložkách záznamových štruktúr patričnými smerníkmi. Implementácia, ktorá

neumožňuje explicitné narábanie so smerníkmi, alebo ktorá využíva inú techniku narábania s pamäťou, núti programátora, aby každý záznam pre Adama a Evu zobrazil dvakrát. Táto „neefektívnosť" by nevadila v tých prípadoch, keď sa iba skúmajú jednotlivé zložky záznamovej štruktúry. V týchto prípadoch je totiž jedno, či sú dvaja otcovia (a dve matky) reprezentované doma záznamami alebo iba jediným. Problém však začína byť vážny v okamihu, keď nás začne zaujímať otázka výberovej aktualizácie. Manipulácia so smerníkmi ako s explicitnou údajovou položkou, a nie ako so skrytým implementačným prostriedkom, umožňuje programátorovi jasne a čitateľne vyjadriť, kde sa predpokladá spoločné používanie určitej pamäťovej oblasti.

Ďalším dôsledkom explicitného používania smerníkov je možnosť definovania a manipulovania s cyklickými štruktúrami údajov. Táto prídavná flexibilita síce zvyšuje výrazové možnosti programovacieho jazyka, ale súčasne núti programátora k ostražitosti, pretože narábanie s cyklickými štruktúrami údajov môže ľahko viesť k nekonečnému procesu.

Tento jav flexibility a výrazovej sily je dobre známy z programovania a pripomína najmä problémy spojené s príkazom **goto**. Skutočne, ak ďalej rozšírime analógiu medzi štruktúrami programu a údajov, tak čisto rekurzívnu štruktúru údajov možno postaviť na úroveň zodpovedajúcu procedúre, zatiaľ čo zavedenie smerníkov možno prirovnať k používaniu príkazu **goto**. Keďže príkaz **goto** umožňuje konštrukciu ľubovoľných programových riadiacich štruktúr (vrátane cyklov), tak aj smerníky dovoľujú kompozíciu ľubovoľných štruktúr údajov (vrátane cyklických).

Paralelný vývin zodpovedajúcich programových a údajových štruktúr stručne zobrazuje *tab. 4.1*.

V tretej kapitole sme ukázali, že iterácia je špeciálny prípad rekurzie a že vyvolanie rekurzívnej procedúry P definovanej podľa schémy (4.9),

$$\begin{array}{l} \textbf{procedure } P; \\ \textbf{begin} \\ \quad \textbf{if } B \textbf{ then begin } P_0;\ P \textbf{ end} \\ \textbf{end} \end{array} \tag{4.9}$$

kde P_0 je príkaz neobsahujúci vyvolanie procedúry P, je ekvivalentné

Zhodnosť programových a údajových štruktúr *Tabuľka 4.1*

Konštrukčný vzor	Príkaz programovacieho jazyka	Typ údajov
atomický prvok	priraďovací príkaz	skalár
enumerácia	zložený príkaz	záznam
cyklus so známym počtom opakovaní	príkaz cyklu **for**	pole
výber	podmienený príkaz	záznam s variantmi zjednotenia
cyklus s neznámym počtom opakovaní	príkazy cyklov **while** alebo **repeat**	postupnosť alebo súbor
rekurzia	príkaz vyvolania procedúry	rekurzívny typ údajov
všeobecný „graf"	príkaz **goto**	štruktúra pospájaná smerníkmi

a nahraditeľné iteráciou

$$\textbf{while } B \textbf{ do } P_0$$

Analógie zobrazené v *tab. 4.1* ukazujú, že podobný vzťah existuje medzi rekurzívnym typom údajov a postupnosťou. Skutočne, rekurzívny typ definovaný podľa schémy

$$\begin{array}{l} \textbf{type } T = \textbf{record} \\ \qquad\qquad \textbf{if } B \textbf{ then } (t_0\colon T_0;\ t\colon T) \\ \qquad\quad \textbf{end} \end{array} \tag{4.10}$$

pričom T_0 je typ definovaný nezávisle od typu T, je ekvivalentný a nahraditeľný typom sekvenčný súbor, definovaným ako

$$\textbf{file of } T_0$$

Z týchto dvoch príkladov vidíme, že rekurzia sa dá nahradiť iteráciou v programe i v definíciách údajov vtedy (a len vtedy), ak sa identifikátor procedúry alebo typu rekurzívne vyskytuje iba jediný raz, a to buď na konci, alebo na začiatku svojej definície.

Zvyšné časti tejto kapitoly sú venované problému generovania a manipulácie so štruktúrami údajov, ktorých zložky sú pospájané pomocou explicitných smerníkov. Dôraz sa kladie hlavne na štruktúry s jednoduchou schémou definície; pritom spôsob narábania so zložitejšími štruktúrami sa dá ľahko odvodiť z jednoduchých štruktúr, ako sú lineárny zoznam alebo zreťazená postupnosť (t. j. najjednoduchší prípad) a stromy. Skutočnosť, že sa zaoberáme iba týmito prostriedkami štruktúrovania údajov, ešte neznamená, že sa v praxi nemôžu vyskytovať i komplikovanejšie štruktúry. Skutočne, nasledujúci príbeh, ktorý uverejnili zürišské noviny v júli roku 1922, je dôkazom toho, k akým konfliktným situáciám môže dôjsť aj v rámci takých pravidelných štruktúr, akými sú napr. (rodinné) stromy. Príbeh, ktorý novinám poskytol jeden muž, bol jeho životným osudom a znie takto:

Oženil som sa s jednou vdovou, ktorá mala dospelú dcéru. Môj otec, ktorý nás dosť často navštevoval, sa zaľúbil do mojej nevlastnej dcéry a oženil sa s ňou. Tým sa môj otec stal mojím zaťom a moja nevlastná dcéra mojou matkou. O niekoľko mesiacov porodila moja manželka syna, ktorý sa stal švagrom môjho otca a bol súčasne mojím strýkom. Manželka môjho otca, t. j. moja nevlastná dcéra, mala tiež syna. Tento bol mojím bratom a súčasne i vnukom. Moja manželka je mojou starou matkou, pretože je matka mojej matky. Na základe toho som jednak manželom svojej ženy a súčasne jej nevlastným vnukom; inými slovami, som svojím vlastným starým otcom.

4.3 LINEÁRNE ZOZNAMY

4.3.1 ZÁKLADNÉ OPERÁCIE

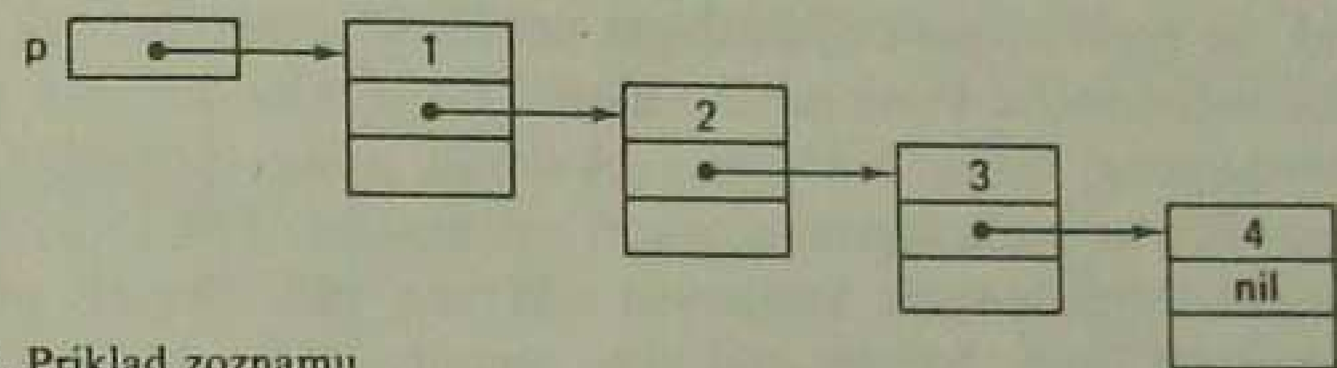
Najjednoduchší spôsob vzájomného združenia alebo pospájania prvkov nejakej množiny je ich zoradenie do zoznamu alebo frontu. V týchto prípadoch je potrebný iba jeden spojovací článok pre každý prvok množiny, ktorý stačí na určenie nasledujúceho prvku v množine.

Predpokladajme, že typ T je definovaný podľa schémy (4.11). Potom každá premenná typu T pozostáva z troch zložiek, a to zo zložky

identifikujúcej hodnotu kľúča, zo smerníka určujúceho nasledovníka a prípadne ďalšej informácie, ktorá charakterizuje príslušnú premennú, ale je v definícii (4.11) vynechaná.

```
type T = record kľúč: integer;
                 ďalší: ↑T;
                 ...........                    (4.11)
         end
```

Zoznam premenných typu T spoločne so smerníkom ukazujúcim na jeho prvú premennú (priradeným premennej p), je znázornený na *obr. 4.6*.



Obr. 4.6. Príklad zoznamu

Pravdepodobne najjednoduchšou operáciou, ktorá sa vykonáva so zoznamom, akú predstavuje aj *obr. 4.6*, je vloženie nového prvku na začiatok zoznamu. Táto operácia sa dá uskutočniť nasledujúcou postupnosťou elementárnych operácií: najprv sa pridelí (vyhradí) pamäť pre prvok typu T (slúži na to známa procedúra new), potom sa smerník na tento prvok priradí pomocnej premennej typu smerník (nech sa volá q). Nasledujúce priradenia smerníkov završia operáciu vloženia nového prvku do zoznamu (pozri text programu (4.12)).

$$\text{new}(q);\ q\uparrow.\mathit{ďalší} := p;\ p := q \tag{4.12}$$

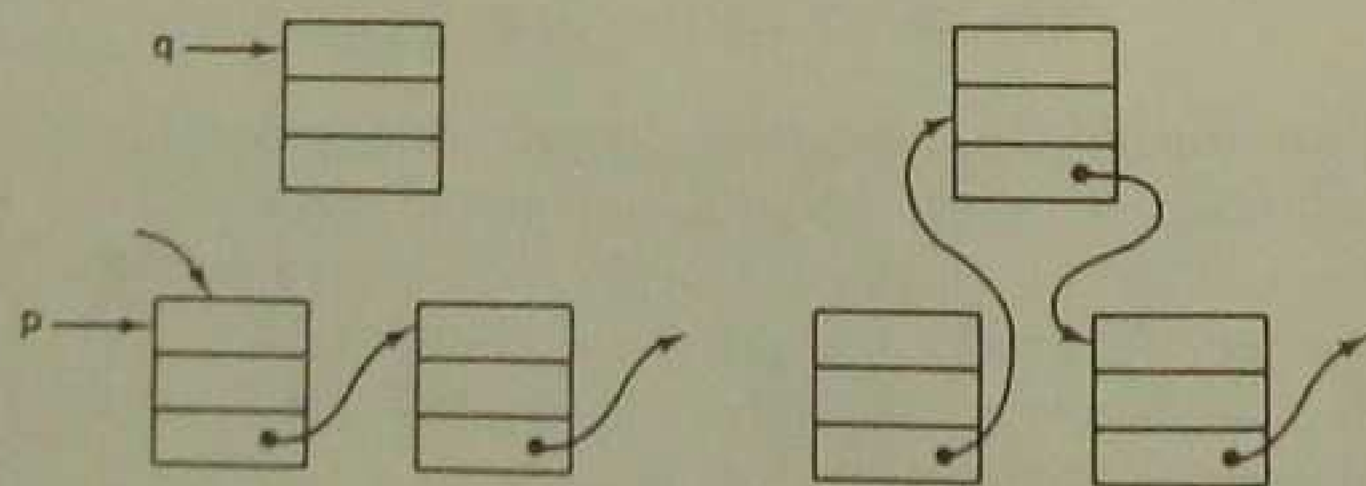
Upozorňujeme, že poradie týchto troch príkazov je záväzné. Operácia pridania prvku do zoznamu súčasne naznačuje, ako možno takýto zoznam vytvoriť: Začíname prázdnym zoznamom a postupne, pomocou uvedenej operácie, pridávame do zoznamu nové prvky. Tento proces generovania zoznamu ukazuje algoritmus (4.13); počet prvkov zoznamu je vyjadrený symbolom n.

```
p := nil; {začíname s prázdnym zoznamom}
while n > 0 do
  begin new(q); q↑.další := p; p := q;                    (4.13)
    q↑.kľúč := n; n := n − 1
  end
```

Takýto algoritmus predstavuje najjednoduchší spôsob tvorby zoznamu. Uvedomme si však, že výsledné usporiadanie prvkov zoznamu je obrátené vzhľadom na poradie, v akom sa prvky pridávajú do zoznamu. Táto skutočnosť je v niektorých praktických aplikáciách nevyhovujúca, pretože obvykle vyžadujeme, aby sa novopridaný prvok vyskytoval na konci zoznamu. Aj keď koniec zoznamu môžeme jednoducho zistiť tak, že prehľadáme celý zoznam, predsa len tento naivný prístup vyžaduje určité úsilie, ktoré možno ušetriť zavedením druhého smerníka, povedzme q, ktorý v každom okamihu určuje posledný prvok zoznamu. Táto metóda je použitá napr. v programe (4.4), ktorý realizuje algoritmus generovania krížových odkazov jednotlivých objektov programového textu. Nevýhodou tejto metódy je, že vloženie prvého prvku do zoznamu sa líši od všetkých ostatných vložení.

Explicitné použitie smerníkov umožňuje realizovať určité operácie veľmi jednoduchým spôsobom; v inom prípade by ich bolo potrebné riešiť oveľa ťažkopádnejšie. Spomedzi základných operácií nad štruktúrou zoznamu sú to: pridanie a zrušenie prvkov (výberová aktualizácia zoznamu) a, samozrejme, prechod zoznamu. Najskôr preskúmame operáciu pridania prvku do zoznamu.

Predpokladajme, že prvok určený smerníkom (premennou) q treba vložiť (pridať) do zoznamu za prvok určený smerníkom p. Potrebné



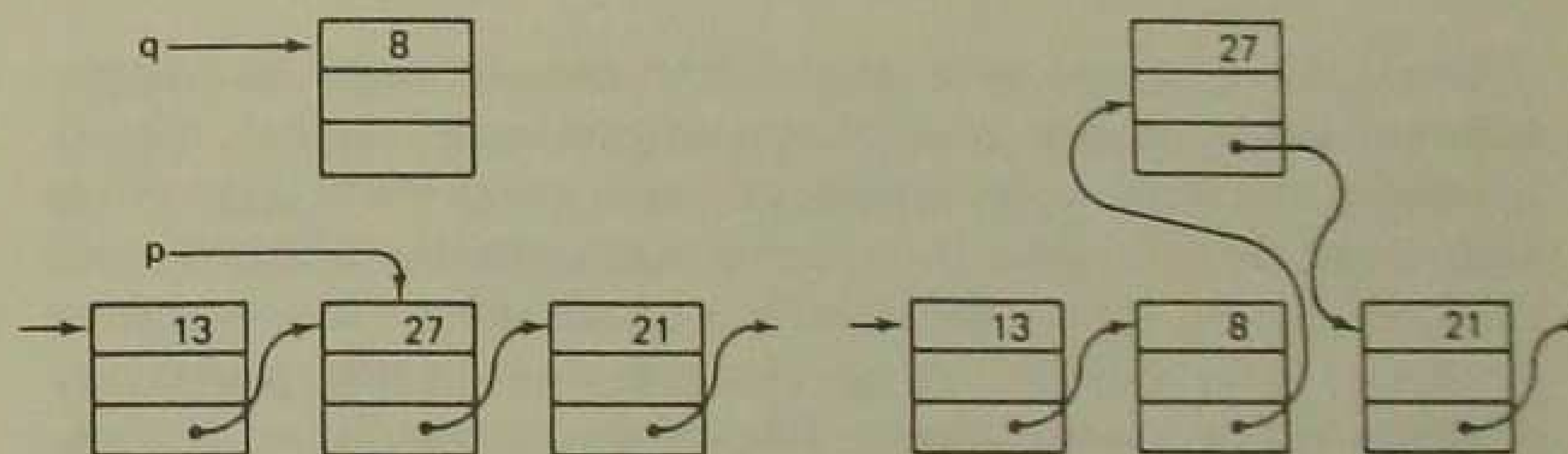
Obr. 4.7. Pridanie prvku do zoznamu za prvok $p\uparrow$

smerníkové priradenia sú vyjadrené príkazmi (4.14) a ich účinok znázorňuje *obr. 4.7.*

$$q\uparrow.\mathit{další} := p\uparrow.\mathit{další};\ p\uparrow.\mathit{další} := q \qquad (4.14)$$

Ak by sme však chceli pridať prvok do zoznamu pred prvok určený smerníkom $p\uparrow$, zistili by sme, že jednosmerná spojovacia reťaz spôsobí problémy, pretože neumožňuje prístup k predchodcom jednotlivých prvkov zoznamu. Tento problém však vyrieši jednoduchý trik, vyjadrený príkazmi (4.15) (*obr. 4.8*). Predpokladajme, že kľúč nového prvku má hodnotu $k = 8$.

```
new(q); q↑ := p↑;                              (4.15)
p↑.kľúč := k; p↑.další := q
```



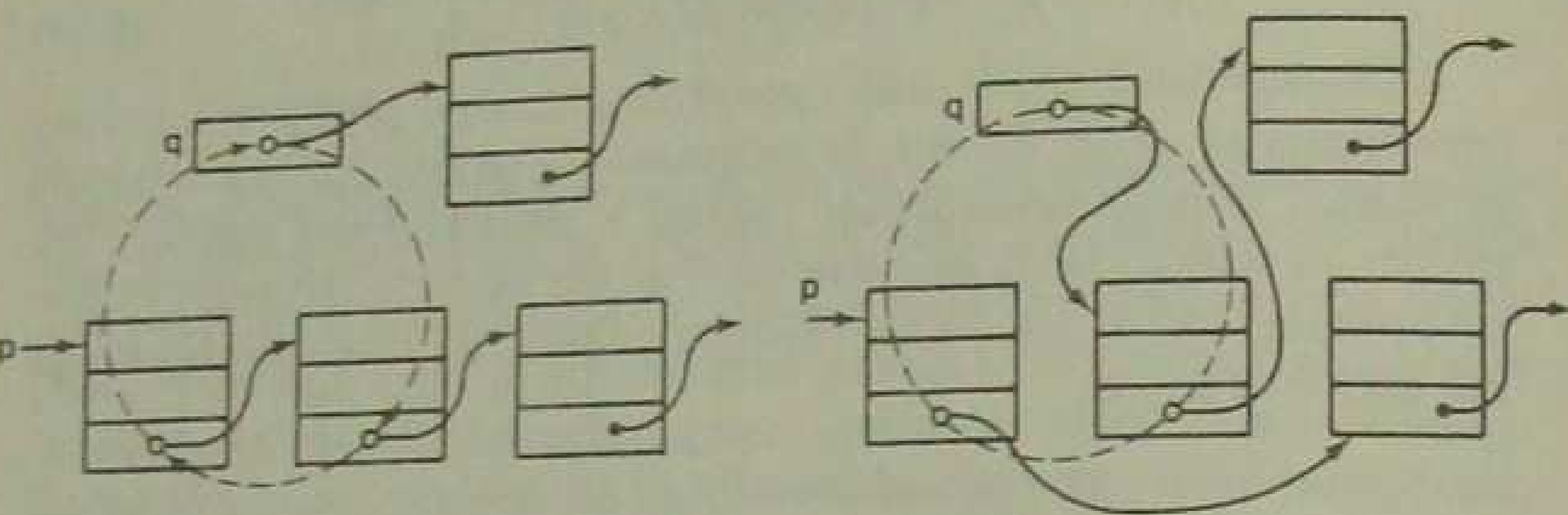
Obr. 4.8. Pridanie prvku do zoznamu pred prvok $p\uparrow$

Trik spočíva v tom, že nový prvok sice v skutočnosti pridáme za prvok určený premennou $p\uparrow$, ale vzápäti sériou priraďovacích príkazov vzájomne zameníme hodnoty novopridaného prvku a prvku určeného smerníkovou hodnotou $p\uparrow$.

Ďalšou základnou operáciou na štruktúre zoznam je zrušenie prvku. Zrušenie nasledovníka $p\uparrow$ je opäť priamočiare. V schéme (4.16) je táto operácia ukázaná spoločne s operáciou, ktorá vzápäti tento odobratý prvok pridá, ale na začiatok iného zoznamu (označeného smerníkovou premennou q). Symbolom r je označená pomocná premenná typu $\uparrow T$.

```
r := p↑.další; p↑.další := r↑.další;            (4.16)
r↑.další := q; q := r
```

Obr. 4.9 zobrazuje proces uvedený v schéme (4.16) a súčasne dokumentuje cyklickú výmenu jeho troch smerníkov.



Obr. 4.9. Zúženie prvku v zozname a jeho pridanie do iného zoznamu

Oveľa ťažšou operáciou je však zrušenie samotného prvku, označeného smerníkovou premennou (namiesto jeho nasledovníka), pretože sa tu opäť stretávame s problémom, s ktorým sme sa stretli pri pridávaní prvku pred prvok označený smerníkom $p\uparrow$: nemáme prístup k predchodcovi daného prvku. Jednoduchý a zaužívaný spôsob, pomocou ktorého sa tento problém rieši, je: Hodnotu nasledovníka presunieme do prvku určeného smerníkom p a nasledovníka odstránime z reťaze, pretože už nie je potrebný (vyskytoval by sa v zozname dvakrát). Túto techniku však môžeme použiť iba v prípade, keby mal prvok $p\uparrow$ nasledovníka, t. j. keby nebol posledný v zozname.

Treťou základnou operáciou so zoznamom je prechod zoznamom. Predpokladajme, že na každom prvku v zozname, ktorého prvý prvok je určený hodnotou smerníka $p\uparrow$, chceme aplikovať operáciu $P(x)$. Túto úlohu môžeme vyjadriť nasledujúcim algoritmom:

```
while zoznam určený smerníkom p nie je prázdny do
begin vykonaj operáciu P;
      prejdi na nasledovníka
end
```

Podrobne, t. j. pomocou zaužívanej notácie jazyka pascal, možno tento algoritmus vyjadriť schémou


```
while p ≠ nil do
   begin P(p↑); p: = p↑.další
   end                                          (4.17)
```

Z podstaty príkazu **while** a zoznamu vyplýva, že operácia P bude aplikovaná na všetky prvky zoznamu (a na žiadne iné).

Veľmi častou operáciou na zoznamoch je vyhľadanie prvku, ktorého kľúč má určitú hodnotu x. Táto operácia je podobne ako v prípade súboru, čisto sekvenčného charakteru. Vyhľadávanie končí buď nájdením hľadaného prvku, alebo dosiahnutím konca zoznamu. Predpokladajme, že začiatok zoznamu je opäť určený smerníkom p. Potom prvý pokus o formuláciu algoritmu vyhľadávania prvku môže mať takýto tvar:

```
while (p ≠ nil) ∧ (p↑.kľúč ≠ x) do p: = p↑.další     (4.18)
```

Musíme si však uvedomiť, že vzťah $p = \textbf{nil}$ znamená, že neexistuje prvok $p\uparrow$. Teda vyhodnotenie podmienky ukončenia cyklu môže spôsobiť referenciu na neexistujúcu premennú (na rozdiel od premennej s nedefinovanou hodnotou), a tým abnormálne ukončenie realizácie programu. Tejto situácii sa môžeme vyhnúť buď prostredníctvom explicitného prerušenia vyhľadávacieho cyklu vyjadreného príkazom **goto** (pozri schému (4.19)), alebo zavedením pomocnej boolovskej premennej, určujúcej nájdenie hľadaného kľúča (pozri algoritmus (4.20)).

```
while p ≠ nil do
   if p↑.kľúč = x then goto nájdený              (4.19)
                  else p: = p↑.další
```

Ako si čitateľ už určite všimol, použitie príkazu **goto** vyžaduje prítomnosť návestia na patričnom mieste v programe (v našom prípade je návestie označené symbolom nájdený). Poznamenávame, že jeho nekompatibilita s príkazom **while** vyplýva z toho, že v tomto prípade došlo k „oklamaniu" alebo „zneužitiu" príkazu **while**. Príkaz, ktorého opakované vykonávanie špecifikuje práve podmienka v príkaze **while**, sa nebude vykonávať dovtedy, kým je splnené $p \neq \textbf{nil}$ (čo je v rozpore s princípmi príkazu **while**).

```
b: = true;
while (p ≠ nil) ∧ b do
    if p↑.kľúč = x then b: = false                    (4.20)
                   else p: = p↑.ďalší
{(p = nil) ∨ ¬ b}
```

4.3.2 USPORIADANÉ ZOZNAMY A REORGANIZOVANÉ ZOZNAMY

Algoritmus (4.20) veľmi pripomína procedúry na prehľadávanie poľa alebo súboru. V skutočnosti je vlastne súbor lineárnym zoznamom, v rámci ktorého sú spojenia medzi jednotlivými prvkami buď nešpecifikované, alebo dané implicitne. Pretože primitívne operátory so súborom neumožňujú pridávanie nových prvkov (okrem pridávania na koniec súboru, t.j. za posledný prvok súboru) alebo zrušenie prvkov (okrem zrušenia všetkých prvkov súboru), ponecháva sa voľba reprezentácie na implementátora. Implementátor môže s výhodou zvoliť sekvenčné prideľovanie pamäti jednotlivým prvkom štruktúry. Tieto sa potom budú nachádzať v spojitých pamäťových oblastiach. Lineárne zoznamy s explicitnými smerníkmi predstavujú *väčšiu flexibilitu*, a preto sa odporúča používať ich, najmä ak sa vyžaduje táto dodatočná flexibilita.

Na objasnenie uvedeného použijeme príklad, ktorý sa bude v tejto kapitole ešte viackrát vyskytovať a ktorý nám bude súčasne slúžiť na ilustráciu alternatívnych riešení a techník. Vybrali sme takýto problém: Postupným čítaním nejakého textu budeme z neho vyberať jednotlivé slová a súčasne počítať a zaznamenávať frekvenciu ich výskytov. Tento proces sa nazýva *konkordancia*.

Bežným riešením je konštrukcia zoznamu všetkých slov, vyskytujúcich sa v texte, a to tak, že zoznam (spočiatku prázdny) po prečítaní každého slova prehľadáme a zistíme, či obsahuje dané slovo. Ak áno, jeho frekvenčné číslo sa zvýši; v opačnom prípade slovo pridáme do zoznamu. Tento proces nazývame jednoducho vyhľadávaním, aj keď môže zjavne zahŕňať i operáciu pridávania.

Aby sme mohli sústrediť našu pozornosť na podstatné problémy manipulácie so zoznamom, predpokladajme, že slová už boli nejakým

spôsobom vybraté zo skúmaného textu, zakódované ako celé čísla a sprístupnené vo forme vstupného súboru.

Formuláciu procedúry, ktorú nazývame vyhľadaj, môžeme priamočiaro odvodiť na základe algoritmu (4.20). Premennou koreň sa odvolávame na začiatok zoznamu, do ktorého pridávame nové slová, v súlade so schémou (4.12). Úplný algoritmus prezentuje program 4.1, ktorý navyše obsahuje procedúru tabelovania vytvoreného zoznamu. Tabelovanie je príkladom procesu, v ktorom sa určitá akcia vykoná jedenkrát pre každý prvok zoznamu, ako to možno vidieť i zo schémy (4.17).

PROGRAM 4.1. *Priame pridávanie prvkov do zoznamu*

```
program Zoznam (input, output);
{priame pridávanie do zoznamu}
  type ref = ↑slovo;
       slovo = record kľúč: integer;
                      počet: integer;
                      ďalší: ref
       end;
  var k: integer; koreň: ref;
  procedure vyhľadaj (x: integer; var koreň: ref);
    var w: ref; b: boolean;
  begin w: = koreň; b: = true;
    while (w ≠ nil) ∧ b do
        if w↑.kľúč = x then b: = false else w: = w↑.ďalší;
    if b then
    begin {nový prvok} w: = koreň; new (koreň);
      with koreň↑ do
      begin kľúč: = x; počet: = 1; ďalší: = w
      end
    end else
      w↑.počet: = w↑.počet + 1
  end {vyhľadaj};
  procedure vytlačzoznam (w: ref);
  begin while w ≠ nil do
      begin writeln (w↑.kľúč, w↑.počet);
          w: = w↑.ďalší
```

```
      end
   end {vytlačzoznam};
begin koreň: = nil; read (k);
   while k ≠ 0 do
      begin vyhľadaj (k, koreň); read (k)
      end;
   vytlačzoznam (koreň)
end.
```

Algoritmus lineárneho vyhľadávania v programe 4.1 pripomína procedúru vyhľadávania v poli a súbore a predovšetkým jednoduchú techniku umožňujúcu zjednodušenie podmienky ukončenia cyklu. Touto technikou je použitie *zarážky*. Zarážka sa dá rovnako výhodne použiť aj pri vyhľadávaní v zozname; je reprezentovaná fiktívnym prvkom na konci zoznamu. Nová procedúra, ktorá je zachytená v schéme (4.21), nahrádza vyhľadávaciu procedúru programu 4.1. Novým objektom tejto procedúry je globálna premenná *zarážka* a inicializácia premennej koreň je nahradená príkazmi

new (*zarážka*); *koreň*: = *zarážka*;

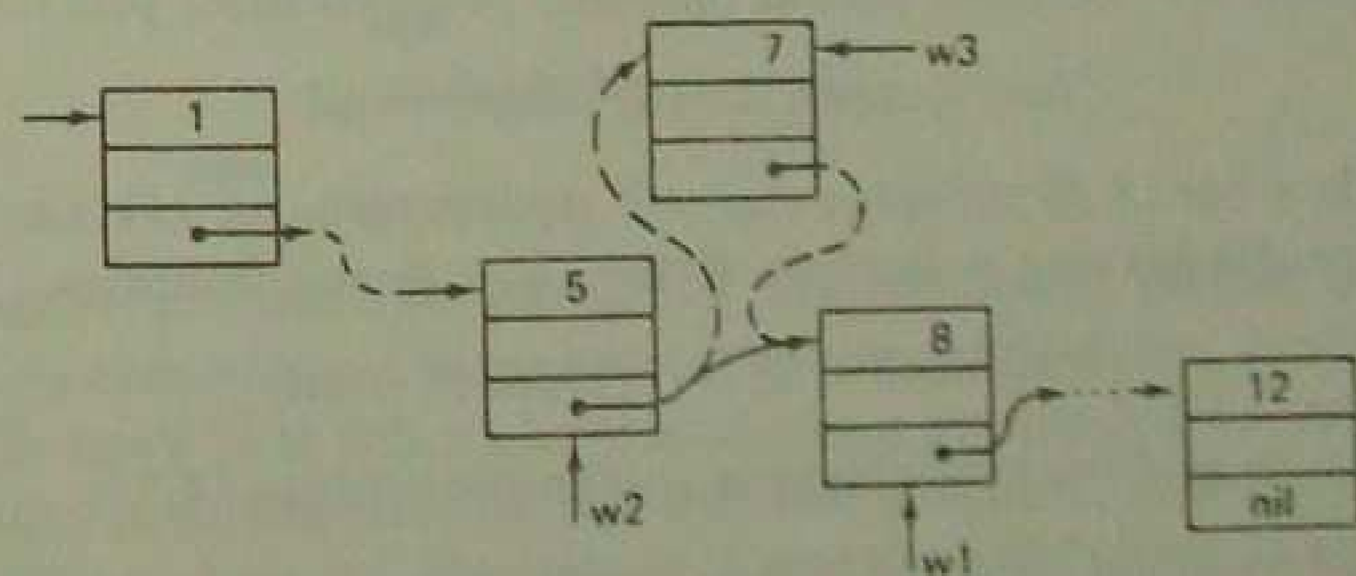
Príkazy generujú prvok, ktorý sa použije ako zarážka.

```
procedure vyhľadaj (x: integer; var koreň: ref);
   var w: ref;
begin w: = koreň; zarážka↑.kľúč: = x;
   while w↑.kľúč ≠ x do w: = w↑.ďalší;
   if w ≠ zarážka then w↑.počet: = w↑.počet + 1 else                 (4.21)
      begin {nový prvok} w: = koreň; new (koreň);
         with koreň↑ do
         begin kľúč: = x; počet: = 1; ďalší: = w
         end
      end
end {vyhľadaj}
```

Tento príklad nedokumentuje silu a flexibilitu použitia zoznamovej štruktúry najlepšie, pretože lineárne vyhľadávanie v celom zozname je prijateľné iba v prípade malého počtu prvkov zoznamu. Jednoduchým

vylepšením je vyhľadávanie v usporiadanom zozname. Ak je zoznam usporiadaný (povedzme podľa zväčšujúcich sa hodnôt kľúčov), vyhľadávací proces sa dá ukončiť najneskôr vtedy, keď sa vyskytne prvý kľúč s väčšou hodnotou, ako je hodnota nového kľúča. Usporiadanie v zozname dosiahneme tým, že každý prvok vložíme na správne miesto, a nie na začiatok zoznamu. Vidíme, že usporiadanie získame prakticky zadarmo. Je to spôsobené tým, že operácia pridania prvkov do usporiadaného zoznamu sa dá realizovať veľmi efektívne, s maximálnym využitím flexibility takejto štruktúry. Štruktúry pole a súbor takéto možnosti neposkytujú. (Poznamenávame však, že v usporiadaných zoznamoch sa nedá použiť nijaký ekvivalent metódy binárneho vyhľadávania v poliach.)

Vyhľadávanie v usporiadanom zozname je typickým príkladom situácie opísanej schémou (4.15), ktorá rieši problém pridania prvku pred určitý prvok (bol označený smerníkom *p*). Teraz nás však bude zaujímať pridanie prvku pred prvý taký prvok, ktorého kľúč má väčšiu hodnotu. Zvolená technika, ktorú použijeme, sa však líši od tej, ktorú sme ukázali v schéme (4.15). Namiesto kopírovania hodnôt dvoch prvkov využijeme dva smerníky *w*1 a *w*2, ktoré v každom okamihu procesu prechodu zoznamom sprístupňujú jeho dva prvky: predchádzajúci prvok *w*2 a bežný prvok *w*1. To znamená, že smerníkom *w*2, ktorý zaostáva za smerníkom *w*1 o jeden krok, je určená príslušná pozícia na vloženie prvého prvku. Smerník *w*1 určuje prvý prvok, ktorého kľúč má väčšiu hodnotu ako kľúč pridávaného prvku. Proces pridania prvku je vo všeobecnosti znázornený na *obr. 4.10*.



Obr. 4.10. Pridanie prvku do usporiadaného zoznamu

Skôr ako budeme pokračovať, musíme zvážiť dve okolnosti:

1. Smerník na nový prvok ($w3$) musíme priradiť premennej $w2$. *ďalší* okrem prípadu, keď je zoznam ešte prázdny. V záujme jednoduchosti a efektívnosti algoritmu nebudeme tieto dva prípady rozlišovať prostredníctvom podmieneného príkazu. Zvolíme radšej inú stratégiu, ktorou je pridanie fiktívneho prvku na začiatok zoznamu.

2. Prehľadávanie zoznamu pomocou dvoch smerníkov vyžaduje, aby zoznam obsahoval aspoň jeden prvok (okrem fiktívneho). To znamená, že treba rozlíšiť pridanie prvého prvku od pridávania ostatných prvkov a podľa toho realizovať príslušný algoritmus.

Návrh algoritmu, vychádzajúceho z uvedených skutočností a zásad, je vyjadrený schémou (4.23). Procedúra vyhľadaj využíva existenciu procedúry pridaj, ktorá je lokálne deklarovaná v rámci procedúry vyhľadaj. Procedúru pridaj, ktorá generuje a inicializuje nový prvok w, znázorňuje schéma (4.22).

```
procedure pridaj (w: ref);
  var w3: ref;
begin new(w3);
  with w3↑ do
    begin kľúč: = x; počet: = 1; ďalší: = w                    (4.22)
    end;
  w2↑.ďalší: = w3
end {pridaj}
```

Príkaz *koreň*: = **nil**, priraďujúci začiatočnú hodnotu premennej koreň v programe 4.1, môžeme potom nahradiť postupnosťou príkazov

new(*koreň*); *koreň*↑.*ďalší*: = **nil**

V súlade s *obr. 4.10* môžeme stanoviť podmienku, na základe ktorej proces prehľadávania pokračuje. Skladá sa z dvoch faktorov:

$(w1\uparrow.kľúč < x) \wedge (w1\uparrow.ďalší \neq \text{nil})$

Výslednú procedúru vyhľadávania prezentuje schéma (4.23).

```
procedure vyhľadaj (x: integer; var koreň: ref);
  var w1, w2: ref;
begin w2: = koreň; w1: = w2↑.ďalší;
  if w1 = nil then pridaj (nil) else
  begin
    while (w1↑.kľúč < x) ∧ (w1↑.ďalší ≠ nil)
    do begin w2: = w1; w1: = w2↑.ďalší
      end;                                                      (4.23)
    if w1↑.kľúč = x then
      w1↑.počet: = w1↑.počet + 1
    else
      pridaj(w1)
  end
end {vyhľadaj};
```

Žiaľ, uvedený návrh obsahuje formálnu logickú chybu. Algoritmus, aj napriek našej starostlivosti, má jedno chybné miesto (dostal sa nám tam „chrobák"), ktoré by sa mohol pozorný čitateľ pokúsiť identifikovať skôr, než budeme pokračovať. Pre tých, ktorí neprijmú funkciu „detektíva", stačí konštatovanie, že algoritmus (4.23) vždy premiestni prvok, ktorý sa pridal ako prvý, na koniec zoznamu. Chybu môžeme opraviť tak, že zoberieme do úvahy skutočnosť, ktorá nastane pri ukončení prehľadávania vzhľadom na druhý faktor podmienky ukončenia. V tomto prípade musíme nový prvok pridať za prvok $w1\uparrow$ namiesto pred neho. Potom môžeme príkaz pridaj($w1$) nahradiť algoritmom

```
begin if w1↑.ďalší = nil then
        begin w2: = w1; w1 = nil
        end;                                                    (4.24)
        pridaj(w1)
end
```

Je to smutné, ale dôverčivý čitateľ bol opäť oklamaný, pretože algoritmus (4.24) ešte stále nie je správny. Aby sme zistili chybu, predpokladajme, že nový kľúč leží medzi posledným a predposledným kľúčom. Keď proces prehľadávania dosiahne koniec zoznamu, nadobudnú obidva faktory podmienky pokračovania procesu prehľadávania hodnotu false, čím sa prvok pridá za posledný prvok zoznamu. Ak by sa ten istý

kľúč vyskytol ešte raz, bude vložený správne do zoznamu a v rámci tabelovania sa objaví dvakrát. Oprava spočíva v zmene podmienky

$$w1\uparrow.\mathit{\check{d}al\check{s}\acute{\imath}} = \textbf{nil}$$

v algoritme (4.24) na podmienku

$$w1\uparrow.\mathit{k\acute{l}\acute{u}\check{c}} < x$$

V zmysle zrýchlenia procesu vyhľadávania môžeme podmienku pokračovania procesu vo vnútri príkazu cyklu **while** opäť zjednodušiť tak, že použijeme techniku zarážky. Použitie tohto prostriedku však vyžaduje začiatočnú prítomnosť fiktívneho prvého prvku a zarážky na konci zoznamu. Zoznam musí byť preto inicializovaný prostredníctvom príkazov

```
new(koreň); new(zarážka);
koreň↑.ďalší := zarážka;
```

Súčasne sa tým procedúra vyhľadávania veľmi zjednoduší. Jej novú verziu predstavuje schéma (4.25).

```
procedure vyhľadaj (x: integer; var koreň: ref);
  var w1, w2, w3: ref;
begin w2 := koreň; w1 := w2↑.ďalší;
      zarážka↑.kľúč := x;
  while w1↑.kľúč < x do
    begin w2 := w1; w1 := w2↑.ďalší
    end;
  if (w1↑.kľúč = x) ∧ (w1 ≠ zarážka) then
    w1↑.počet := w1↑.počet + 1 else                    (4.25)
    begin new(w3); {vlož w3 medzi w1 a w2}
      with w3↑ do
        begin
          kľúč := x; počet := 1; ďalší := w1
        end;
      w2↑.ďalší := w3
    end
end {vyhľadaj}
```


Teraz už môžeme odpovedať na otázku, aký úžitok má vyhľadávanie v usporiadanom zozname. Vzhľadom na to, že situácia nie je veľmi zložitá, nemožno očakávať nejaké ohromujúce zlepšenie.

Predpokladajme, že sa všetky slová vyskytujú v texte s rovnakou frekvenciou. V takomto prípade je zisk z lexikografického usporiadania skutočne nulový, pretože nezáleží na pozícii slova, ale zaujíma nás predovšetkým celkový počet prístupov k slovám. Pritom vieme, že frekvencia výskytu je rovnaká pre všetky slová. Získame však pri každej operácii pridanie nového slova do zoznamu. Namiesto pôvodného prehľadávania celého zoznamu potrebujeme teraz prehľadať iba polovicu zoznamu. Z toho vyplýva, že pridávanie do usporiadaného zoznamu je výhodné najmä vtedy, ak máme generovať konkordanciu s veľmi rozdielnymi slovami v porovnaní s frekvenciou ich výskytu. Predchádzajúce príklady nám preto slúžia predovšetkým ako cvičenia, a nie ako nejaké „návody" pre praktické aplikácie.

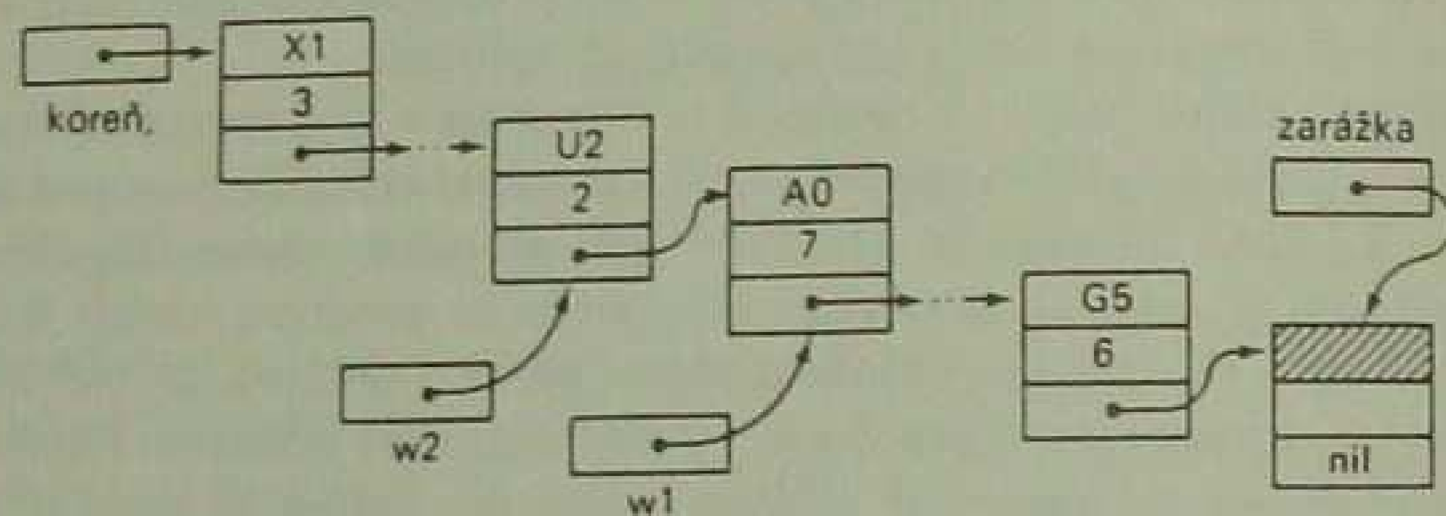
Usporiadanie údajov v zozname sa odporúča v tých prípadoch, keď je počet prvkov pomerne malý (povedzme menej ako sto), mení sa a nie je daná žiadna informácia o frekvenciách ich sprístupnenia. Typickým príkladom je tabuľka symbolov kompilátorov programovacích jazykov. Každá deklarácia spôsobí pridanie nového symbolu do tabuľky. Po výstupe z rozsahu platnosti sa symbol z tabuľky odoberie. Použitie jednoduchých zoznamov je vhodné najmä pre aplikácie, ktoré majú krátke programy. Dokonca aj v tomto prípade možno dosiahnuť významné zlepšenie v prístupovej metóde, a to prostredníctvom veľmi jednoduchej techniky, o ktorej sa zmienime v ďalšom. Táto technika je veľmi peknou ukážkou flexibility zreťazenej zoznamovej štruktúry.

Charakteristickou vlastnosťou programov je, že výskyty toho istého identifikátora sú časté (hromadia sa), t. j. za jedným výskytom nasleduje ďalší alebo ďalšie výskyty toho istého slova. Táto skutočnosť nás núti reorganizovať zoznam po každom prístupe tak, že nájdené slovo zakaždým presunieme na začiatok zoznamu. Tým minimalizujeme dĺžku prechodu zoznamom pri nasledujúcom vyhľadávaní. Takáto prístupová metóda sa nazýva vyhľadávanie v zozname s reorganizáciou alebo, trocha zveličene, samoorganizujúce sa vyhľadávanie v zozname. V procedúre, ktorej algoritmus vybudujeme na základe doterajších úvah a ktorú môžeme zaradiť do programu 4.1, použijeme hneď od začiatku

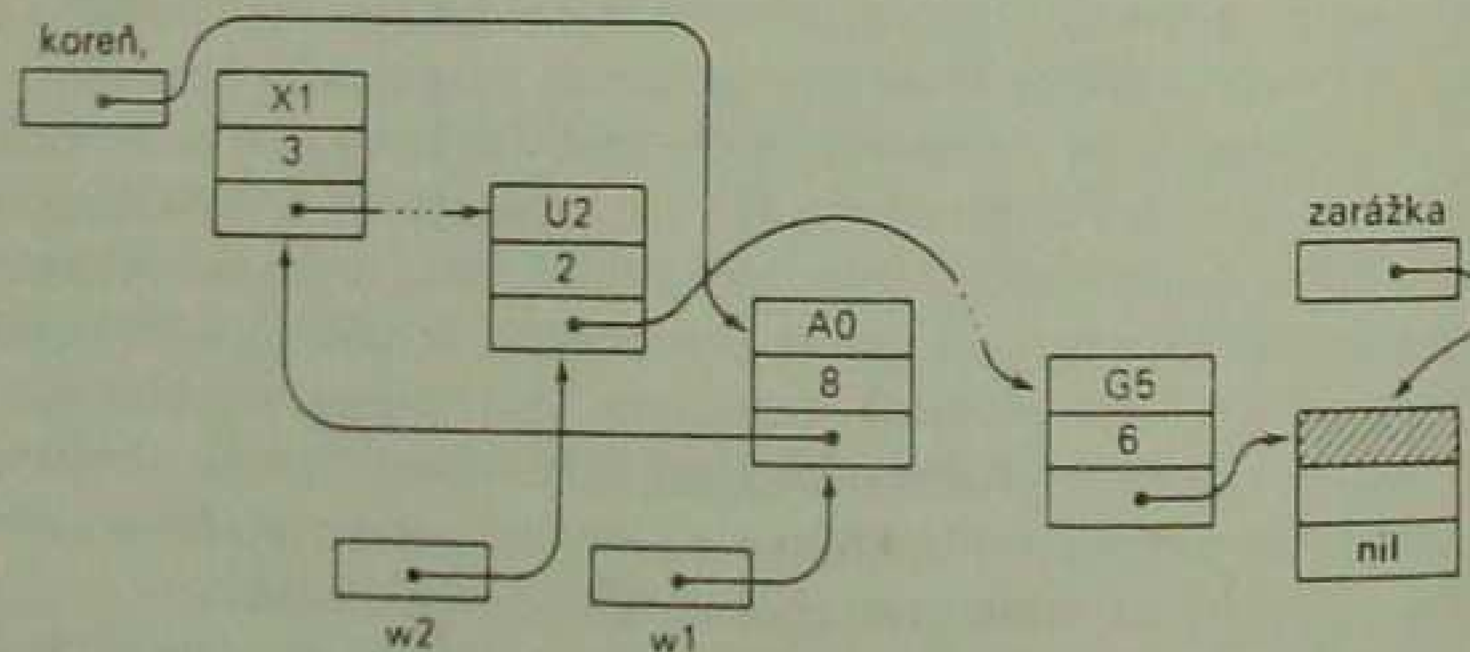
techniku zarážky. V skutočnosti použitie zarážky nielenže zrýchľuje proces vyhľadávania, ale v tomto prípade i zjednodušuje program. Zoznam, pochopiteľne, nie je na začiatku prázdny, ale obsahuje zarážku. Inicializačné príkazy majú takýto tvar:

new (*zarážka*); *koreň* : = *zarážka*;

Poznamenávame, že podstatný rozdiel medzi našim novým algoritmom a priamym vyhľadávaním v zozname (4.21) predstavuje preusporiadanie zoznamu po každom nájdení prvku. Tento prvok sa potom premiestni zo svojej starej pozície na začiatok zoznamu. Akcia vyžaduje opäť prítomnosť dvoch pomocných smerníkov *w*1, *w*2, pomocou ktorých je stále prístupný predchodca *w*2↑ identifikovaného prvku *w*1↑. To zase vyžaduje špeciálnu manipuláciu s prvým prvkom (t. j. s prázdnym zoznamom). Aby sme si utvorili predstavu o procese preusporiadania, všimneme si *obr. 4.11*, ktorý ukazuje pozíciu smerníkov *w*1 a *w*2 v okamihu, keď bol *w*1↑ identifikovaný ako hľadaný prvok. Situáciu po správnom preusporiadaní znázorňuje *obr. 4.12*. Úplný algoritmus novej procedúry vyhľadávania dokumentuje schéma (4.26).



Obr. 4.11. Zoznam pred preusporiadaním



Obr. 4.12. Zoznam po preusporiadaní

```
procedure vyhľadaj (x: integer; var koreň: ref);
  var w1, w2: ref;
begin w1 : = koreň; zarážka↑.kľúč : = x;
  if w1 = zarážka then
  begin {prvý prvok} new (koreň);
    with koreň↑ do
      begin kľúč : = x; počet : = 1; ďalší : = zarážka
      end
  end else
  if w1↑.kľúč = x then w1↑.počet : = w1↑.počet + 1 else
  begin {vyhľadávanie}
    repeat w2 : = w1; w1 : = w2↑.ďalší
    until w1↑.kľúč = x;
    if w1 = zarážka then                                   (4.26)
    begin {vloženie prvku}
      w2 : = koreň; new (koreň);
      with koreň↑ do
        begin kľúč : = x; počet : = 1; ďalší : = w2
        end
    end else
    begin {nájdený, a teda preusporiadanie}
      w1↑.počet : = w1↑.počet + 1;
      w2↑.ďalší : = w1↑.ďalší; w1↑.ďalší : = koreň; koreň : = w1
    end
  end
end {vyhľadaj}
```

Zlepšenie tejto metódy vyhľadávania veľmi závisí od stupňa zhlukovania symbolov vo vstupnom súbore. Pre daný koeficient zhlukovania môžeme očakávať zlepšenie najmä v prípade veľkých zoznamov. Aby sme si mohli vytvoriť predstavu o očakávanom zlepšení, prezrime si

výsledky empirických meraní uskutočnených pomocou uvedeného programu pre generovanie konkordancie na malom texte a na pomerne veľkom texte. Výsledky získané pomocou metód lineárneho usporiadania (4.21) a reorganizácie zoznamu (4.26) sú zachytené v *tab. 4.2*. Žiaľ, podstatné zlepšenie nastane vtedy, keď použijeme úplne inú organizáciu údajov. K tomuto príkladu sa ešte vrátime v článku 4.4.

Porovnanie vyhľadávacích metód v zoznamoch *Tabuľka 4.2*

	Test 1	Test 2
Počet rôznych kľúčov	53	582
Počet výskytov kľúčov	315	14 341
Čas vyhľadávania pre usporiadaný zoznam	6 207	3 200 622
Čas vyhľadávania pre reorganizovaný zoznam	4 529	681 584
Koeficient zlepšenia	1.37	4.70

4.3.3 APLIKÁCIA: TOPOLOGICKÉ TRIEDENIE

Vhodným príkladom použitia flexibilnej, dynamickej štruktúry údajov je proces topologického triedenia. Je to proces triedenia množiny prvkov, na ktorej je definované čiastočné usporiadanie, t.j. takej, v rámci ktorej je definované usporiadanie iba nad niektorými dvojicami prvkov, nie však nad všetkými. Toto je bežná situácia v praxi. Nasledujúce štyri príklady sú ukážkami čiastočných usporiadaní:

1. Slová v slovníku alebo v glosári sú definované pomocou iných slov. Ak je slovo w definované pomocou slova v, označujeme to výrazom $v \prec w$. Topologické triedenie slov v slovníku znamená také usporiadanie slov, že sa v slovníku už nebudú vyskytovať žiadne dopredné odkazy.

2. Úloha (napr. inžiniersky projekt) sa rozdelí na podúlohy. Dohotoveniu určitých podúloh musí obyčajne predchádzať realizácia iných podúloh. Ak má byť podúloha v dohotovená pred podúlohou w (t.j. podúloha v predchádza podúlohu w), zapisujeme to $v \prec w$. Topologické triedenie v tomto prípade znamená usporiadať podúlohy tak, aby pri začatí určitej podúlohy boli všetky podúlohy, ktoré sú nevyhnutné na jej realizáciu, už dokončené.

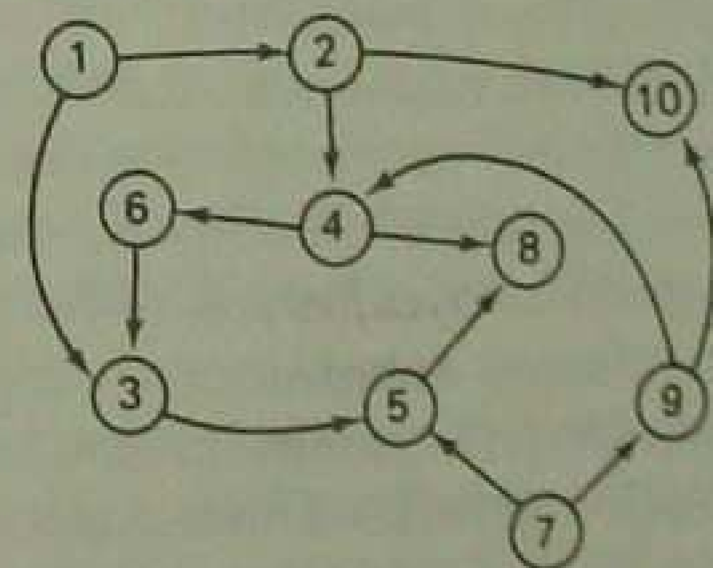
3. Pri zostavovaní plánu výuky na univerzite je dôležité myslieť na to, že látka z určitých predmetov sa musí prebrať skôr, ako sa začne preberať ďalší (nový) predmet, pretože tento predpokladá vedomosti získané z predchádzajúcich predmetov. Ak teda predmet v musí predchádzať predmetu w, vyjadríme to opäť vzťahom $v \prec w$. Topologické triedenie v tomto prípade znamená usporiadať vyučovacie predmety takým spôsobom, že všetky predmety nevyhnutné na zvládnutie iného predmetu budú do zoznamu zaradené pred týmto predmetom.

4. V programoch sa často vyskytujú procedúry, ktoré obsahujú príkazy volania iných procedúr. Ak procedúra w volá procedúru v, môžeme napísať známy vzťah $v \prec w$. Topologické triedenie znamená usporiadať deklarácie procedúr tak, aby sa nevyskytovali žiadne dopredné referencie.

Čiastočné usporiadanie množiny S je vo všeobecnosti dané reláciou medzi prvkami množiny S. Označujeme ju symbolom $\prec$, ktorý znamená „predchádza". Pre tri rôzne prvky x, y, z z množiny S táto relácia spĺňa nasledujúce vlastnosti (axiómy):

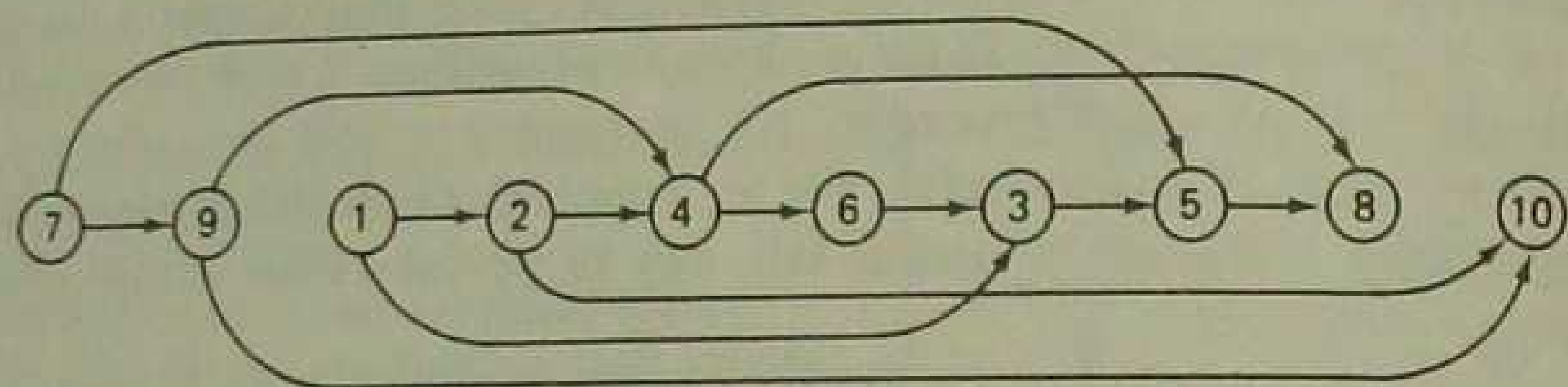
1. Ak platí $x \prec y$ a $y \prec z$, tak platí $x \prec z$ (tranzitivnosť).
2. Ak platí $x \prec y$, tak neplatí $y \prec x$ (asymetria).
3. Neplatí $x \prec x$ (ireflektívnosť). (4.27)

Z pochopiteľných dôvodov budeme predpokladať, že množiny S, ktoré chceme topologicky triediť, sú konečné. Môžeme ich zobraziť pomocou diagramu alebo grafu, ktorého vrcholy predstavujú prvky množiny S a orientované hrany reprezentujú relácie usporiadania. Príklad možno vidieť na *obr. 4.13*.



Obr. 4.13. Čiastočne usporiadaná množina

Problém topologického triedenia spočíva vo vnorení čiastočného usporiadania do lineárneho. Graficky to znamená, že vrcholy grafu sa zoradia tak, aby všetky šípky hrán smerovali doprava, ako to vidieť na *obr. 4.14*. Vlastnosti 1 a 2 čiastočného usporiadania zaručujú, že graf neobsahuje žiadne cykly. Toto je skutočne nevyhnutná podmienka realizácie lineárneho zoradenia prvkov čiastočne usporiadanej množiny.



Obr. 4.14. Linearizácia čiastočne usporiadanej množiny

Ako budeme postupovať pri hľadaní jedného z možných lineárnych usporiadaní? Postup je veľmi jednoduchý. Začneme výberom prvkov, ktorým nepredchádza žiadny iný prvok (musí byť aspoň jeden, pretože inak by graf obsahoval cyklus, čo by bol spor s axiómami 1 a 2). Tieto prvky odstránime z množiny S a zaradíme na začiatok vytváraného zoznamu. Zostávajúca množina je stále čiastočne usporiadaná, a preto môžeme na ňu aplikovať ten istý algoritmus až dovtedy, kým nebude prázdna. Aby sme tento algoritmus mohli presnejšie opísať, musíme urobiť isté rozhodnutia, týkajúce sa štruktúr údajov, reprezentácie množiny S a jej usporiadania.

Voľba reprezentácie množiny S bude veľmi ovplyvnená operáciami, ktoré budeme na nej vykonávať, najmä však operáciou výberu prvkov, ktoré nemajú predchodcu. Každý prvok môžeme preto charakterizovať pomocou troch atribútov: identifikačného kľúča, množiny nasledovníkov a celkového počtu predchodcov. Pretože počet prvkov n množiny S nie je vopred daný, je účelné reprezentovať množinu S pomocou zoznamu. V dôsledku toho budeme pri deklaráciách prvku potrebovať ďalšiu zložku, ktorou bude smerník na nasledujúci prvok zoznamu. Budeme predpokladať, že kľúče sú vyjadrené celočíselnými hodnotami (ale nie nevyhnutne po sebe idúcimi celými číslami 1 až n). Podobne

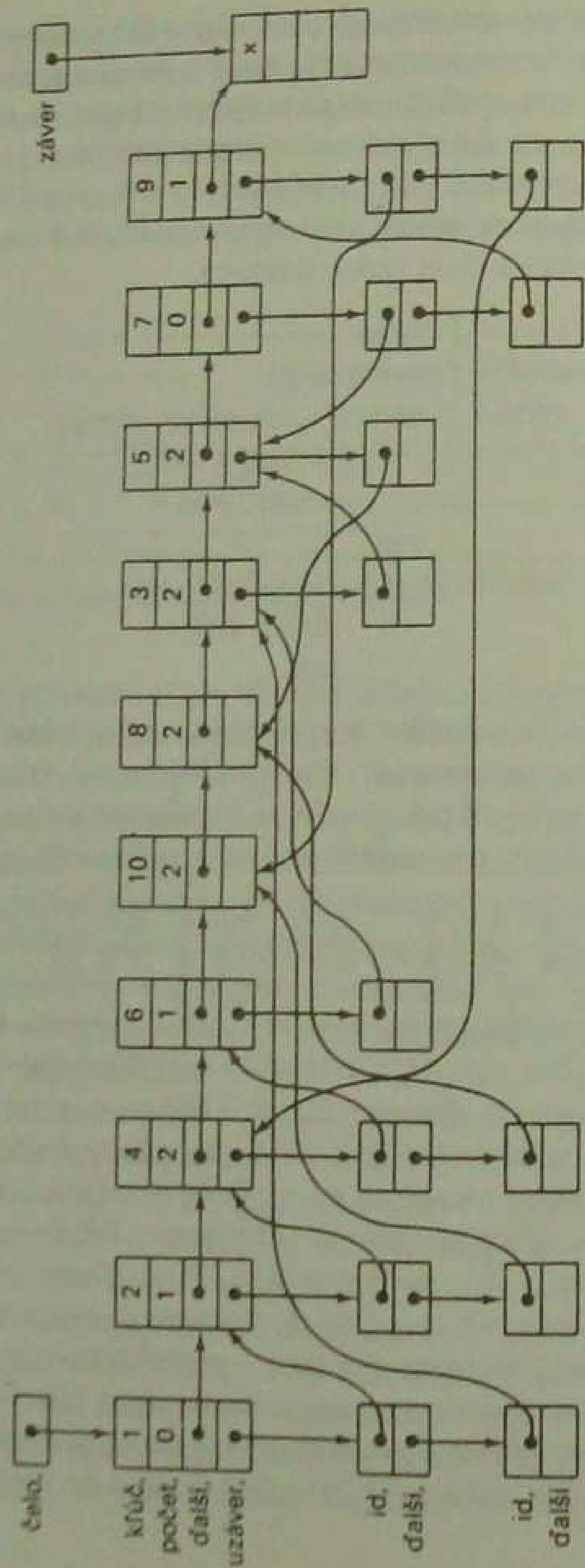
výhodne môžeme prostredníctvom zoznamov reprezentovať i množinu nasledovníkov jednotlivých prvkov. Každý prvok zoznamu nasledovníkov je definovaný svojím identifikačným údajom a smerníkom na ďalší prvok zoznamu. Ak nazveme štruktúru opisujúcu prvok hlavného zoznamu, t. j. zoznamu, v ktorom sa každý prvok objaví iba raz, vedúci a štruktúru opisujúcu prvky zoznamu nasledovníkov uzatvárajúci, dostaneme tieto deklarácie typov údajov:

```
type vref = ↑vedúci
     uref = ↑uzatvárajúci
     vedúci = record kľúč, počet: integer,
                     uzáver: uref;
                     ďalší: vref                    (4.28)
              end;
     uzatvárajúci = record id: vref;
                           ďalší: uref
                    end
```

Predpokladajme, že množina S a jej relácie usporiadania sú spočiatku reprezentované postupnosťou dvojíc kľúčov vo vstupnom súbore. Vstupné údaje pre príklad na *obr. 4.13* sú zobrazené v nasledujúcej schéme (4.29). Kvôli zrozumiteľnosti sú v nej navyše aj symboly $\prec$:

$$\begin{array}{ccccccc} 1\prec 2 & 2\prec 4 & 4\prec 6 & 2\prec 10 & 4\prec 8 & 6\prec 3 & 1\prec 3 \\ 3\prec 5 & 5\prec 8 & 7\prec 5 & 7\prec 9 & 9\prec 4 & 9\prec 10 & \end{array} \qquad (4.29)$$

V úvodnej časti programu topologického triedenia musíme načítať údaje zo vstupného súboru a vytvoriť z nich štruktúru zoznam. To sa uskutoční postupným čítaním dvojíc kľúčov x, y ($x \prec y$). Nech sú štruktúry typu vedúci, ktorých zložkami sú aj uvedené kľúče x, y, prístupné pomocou smerníkov p a q. Tieto záznamové štruktúry musíme nájsť pomocou vyhľadávania v zozname. Ak by ich zoznam ešte neobsahoval, treba ich doňho pridať. Túto úlohu bude vykonávať funkcia L. Nový prvok sa pridá do zoznamu nasledovníkov prvku x spoločne s identifikátorom y a počet predchodcov prvku y sa zvýši o jednotku. Tento algoritmus nazývame vstupná fáza (4.30). *Obr. 4.15* znázorňuje štruktúru údajov vytváranú týmto algoritmom počas spracúvania vstupného súboru (4.29). V algoritme sa volá funkcia $L(w)$,



Obr. 4.15. Zoznamová štruktúra generovaná programom topologického triedenia

ktorej výstupná hodnota predstavuje smerník na prvok zoznamu vedúci, ktorého kľúč má hodnotu w (pozri aj program 4.2). Predpokladáme, že postupnosť dvojíc kľúčov vo vstupnom súbore je ukončená dodatočným kľúčom s hodnotou nula:

```
{vstupná fáza} read (x);
new (čelo); záver: = čelo; z: = 0;
while x ≠ 0 do
  begin read (y); p: = L(x); q: = L(y);
        new (t); t↑.id: = q; t↑.ďalší: = p↑.uzáver;          (4.30)
        p↑.uzáver: = t; q↑.počet: = q↑.počet + 1;
        read (x)
  end
```

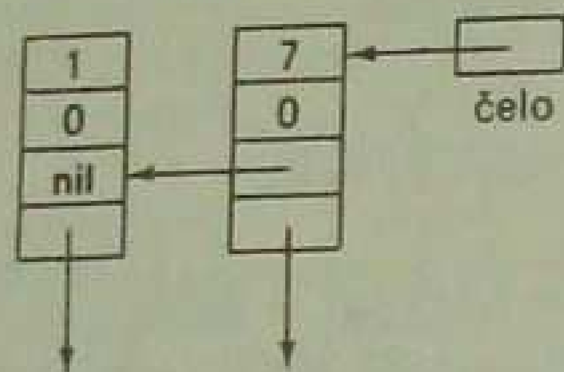
Len čo sa prostredníctvom vstupnej fázy vytvorí štruktúra znázornená na *obr. 4.15*, môže sa začať skutočný proces topologického triedenia podľa uvedeného algoritmu. Vzhľadom na to, že tento algoritmus v každom kroku cyklu vyberá prvok s nulovým počtom predchodcov, bude rozumné, ak tieto prvky najskôr zreťazíme. Na tento zámer môžeme využiť zložku *ďalší*, pretože pôvodné zreťazenie vedúcich už nebudeme potrebovať. Takéto zmeny zreťazenia sa pomerne často vyskytujú pri spracúvaní zoznamov.

Detaily môžeme vidieť v algoritme (4.31), ktorého zvláštnosťou je, že nové zreťazenie prvkov generuje v opačnom smere.

```
{vyhľadanie vedúcich bez predchodcov}
  p: = čelo; čelo = nil;
  while p ≠ záver do
    begin q: = p; p: = q↑.ďalší;
          if q↑.počet = 0 then                                  (4.31)
          begin {pridaj q↑ do novej reťaze}
             q↑.ďalší: = čelo; čelo: = q
          end
    end
```

Vizuálne si zmenu zreťazenia môžeme priblížiť pomocou *obr. 4.15* a *obr. 4.16*, ak zreťazenie vedúcich určené smerníkovou zložkou *ďalší* na *obr. 4.15* nahradíme zreťazením vedúcich, ktorí nemajú predchodcov

(znázornení sú na *obr. 4.16*). Nezobrazené smerníky pritom ostávajú nezmenené.



Obr. 4.16. Zoznam vedúcich, ktorí nemajú predchodcov

Keď sme vykonali všetky dodatočné rozhodnutia, týkajúce sa voľby reprezentácie čiastočne usporiadanej množiny *S*, môžeme konečne pristúpiť ku skutočnému problému topologického triedenia, t. j. generovania výstupnej postupnosti. Prvá verzia algoritmu môže vyzerať takto:

```
q: = čelo;
while q ≠ nil do
begin {výstup tohto prvku a jeho zrušenie}
    writeln (q↑.kľúč); z: = z − 1;
    t: = q↑.uzáver; q: = q↑.ďalší;
    „zníženie počtu predchodcov pri všetkých nasledovni-
    koch tohto prvku v zozname uzatvárajúcich t; ak zlož-       (4.32)
    ka počet nadobudne pri niektorom z nich hodnotu 0,
    treba tento prvok premiestniť do zoznamu vedúcich q"
end
```

Príkaz, ktorý je uvedený formálne v algoritme (4.32) a ktorý musíme v ďalšej fáze zjemniť, si vyžiada ďalšie prehľadanie zoznamu (pozri schému (4.17)). Pomocná premenná *p* bude v každom kroku algoritmu označovať prvok zoznamu vedúcich, ktorého počet predchodcov treba znížiť o jednotku a otestovať, či sa nerovná nule.

```
while t ≠ nil do
begin p: = t↑.id; p↑.počet: = p↑.počet − 1
    if p↑.počet = 0 then
```



```
        begin {pridaj prvok p↑ do zoznamu vedúcich}
          p↑.ďalší: = q; q: = p                               (4.33)
        end
    t: = t↑.ďalší
end
```

Týmto máme vlastne dokončený program topologického triedenia. Všimnime si, že celočíselná premenná *z* v programe 4.2 slúži ako počítadlo vedúcich, generovaných vo vstupnej fáze. Hodnota tohto počítadla sa vo výstupnej fáze zníži o jednotku vždy vtedy, keď sa prvok zoznamu vedúcich premiestni do výstupnej postupnosti. To znamená, že pri skončení programu sa hodnota premennej *z* rovná nule. V prípade, že by tak nebolo, znamená to, že nám zvýšili nejaké prvky, z ktorých každý má nejakého predchodcu. V takomto prípade však množina *S* určite nie je čiastočne usporiadaná.

Uvedená výstupná fáza topologického triedenia je príkladom procesu, ktorý pracuje s pulzujúcim zoznamom, t. j. s takým, do ktorého sa prvky pridávajú, resp. sa z neho vyberajú v nepredvídanom poradí. Preto je tiež príkladom procesu, ktorý plne využíva flexibilitu zoznamovej štruktúry s explicitnými smerníkmi.

PROGRAM 4.2. *Topologické triedenie*

```
program Topotriedenie (input, output);
type vref = ↑vedúci;
     uref = ↑uzatvárajúci
     vedúci = record kľúč: integer;
                     počet: integer;
                     uzáver: uref;
                     ďalší: vref
              end;
     uzatvárajúci = record id: vref;
                           ďalší: uref
                    end;
var čelo, záver, p, q: vref;
    t: uref; z: integer;
    x, y: integer;
```

```
function L(w: integer): vref;
  {určenie vedúceho s kľúčom w}
  var h: vref;
begin h: = čelo; záver↑.kľúč: = w; {zarážka}
  while h↑.kľúč ≠ w do h: = h↑.ďalší;
  if h = záver then
    begin {v zozname nie je prvok s kľúčom w}
      new(záver); z: = z + 1;
      h↑.počet: = 0; h↑.uzáver: = nil; h↑.ďalší: = záver
    end;
    L: = h
end {L};
begin {inicializácia zoznamu vedúcich fiktívnymi prvkami}
  new(čelo); záver: = čelo; z: = 0;
{vstupná fáza} read(x);
  while x ≠ 0 do
  begin read(y); writeln(x, y);
    p: = L(x); q: = L(y);
    new(t); t↑.id: = q; t↑.ďalší: = p↑.uzáver;
    p↑.uzáver: = t; q↑.počet: = q↑.počet + 1;
    read(x)
  end;
{vyhľadanie vedúcich s počtom = 0}
  p: = čelo; čelo: = nil;
  while p ≠ záver do
  begin q: = p; p: = p↑.ďalší;
    if q↑.počet = 0 then
    begin q↑.ďalší: = čelo; čelo: = q
    end
  end;
{výstupná fáza} q: = čelo;
  while q ≠ nil do
  begin writeln(q↑.kľúč); z: = z - 1;
    t: = q↑.uzáver; q: = q↑.ďalší;
    while t ≠ nil do
    begin p: = t↑.id; p↑.počet: = p↑.počet - 1;
```



```
      if p↑.počet = 0 then
      begin {pridaj p↑ do q-zoznamu}
        p↑.ďalší: = q; q: = p
      end;
      t: = t↑.ďalší
    end
  end;
  if z ≠ 0 then
    writeln ('TÁTO MNOŽINA NIE JE ČIASTOČNE
              USPORIADANÁ')
end.
```

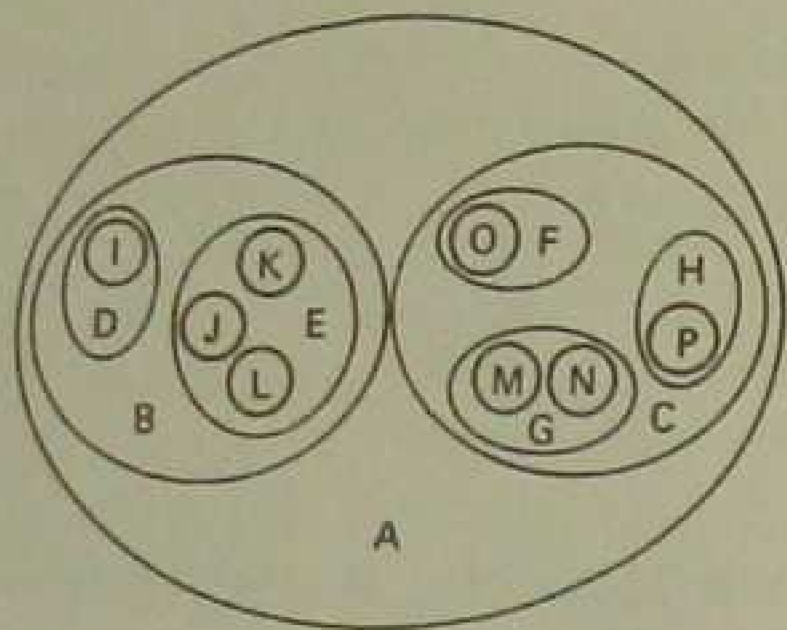
4.4 STROMOVÉ ŠTRUKTÚRY

4.4.1 ZÁKLADNÉ POJMY A DEFINÍCIE

Zistili sme, že postupnosti a zoznamy sa dajú vhodne definovať takto: Postupnosť (zoznam) so základným typom T je buď prázdna postupnosť (zoznam), alebo zreťazenie typu T a postupnosti so základným typom T.

V tejto definícii sme použili rekurziu ako prostriedok na definovanie princípu štruktúrovania, a to radenia do postupnosti alebo iterácie. Postupnosti i terácie sú také bežné, že sa obyčajne považujú za základné vzory štruktúry a správania sa. Treba si však uvedomiť, že sa dajú definovať prostredníctvom rekurzie, pričom opak nie je pravdou; rekurzia je efektívny a elegantný prostriedok na definovanie oveľa dômyselnejších štruktúr. Príklad takejto štruktúry je strom. Nech strom je definovaný nasledujúcim spôsobom: *Štruktúra strom* so základným typom T je buď prázdna štruktúra, alebo vrchol typu T spolu s konečným počtom pripojených disjunktných stromových štruktúr so základným typom T, ktoré nazývame *podstromy*.

Z podobnosti definícií postupnosti a stromových štruktúr vyplýva, že postupnosť (zoznam) je vlastne stromová štruktúra, v ktorej má každý vrchol aspoň jeden podstrom. Postupnosť (zoznam) sa preto tiež nazýva *degenerovaný strom*.

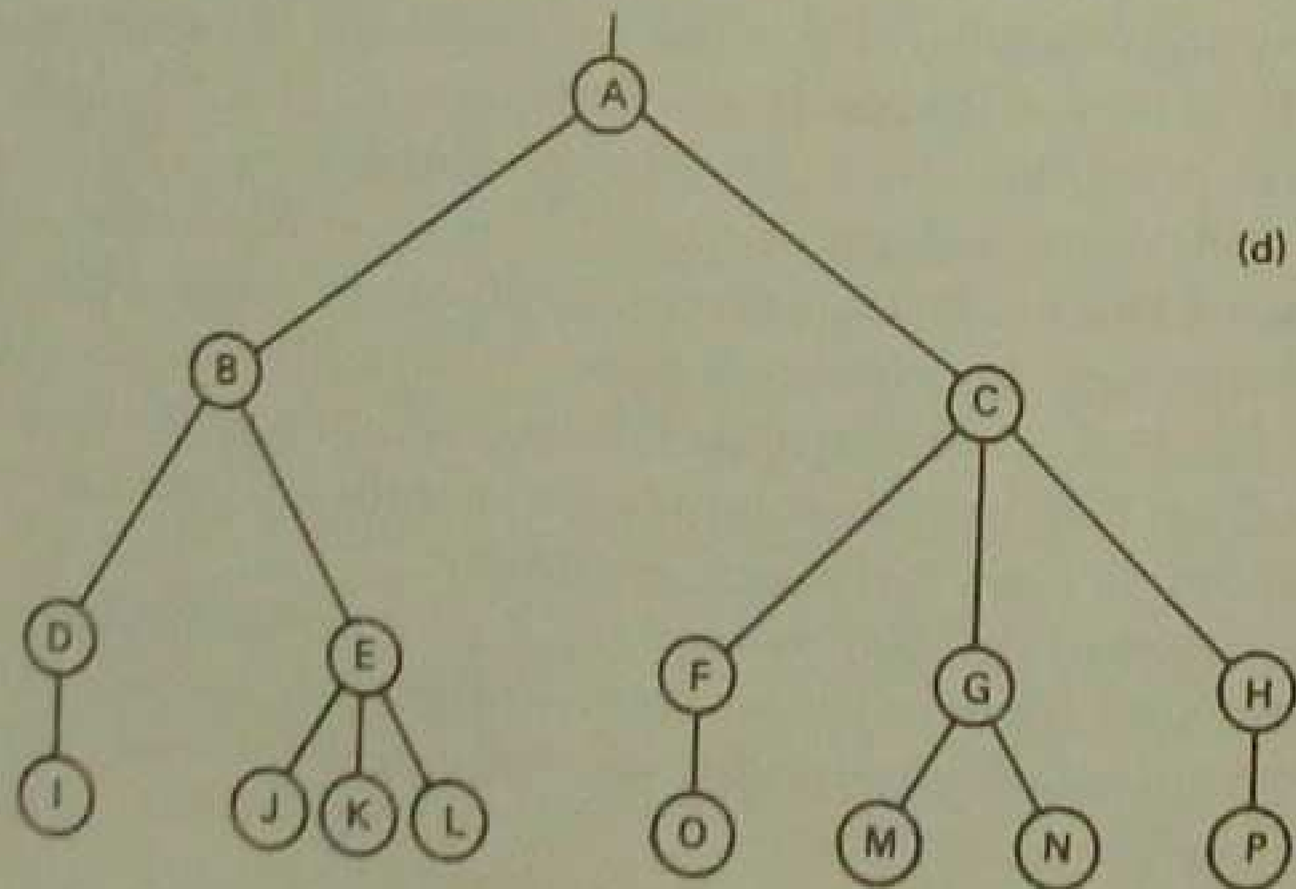


(a)

(A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P)))) (b)

A
B
D
I
E
J
K
L
C
F
O
G
M
N
H
P

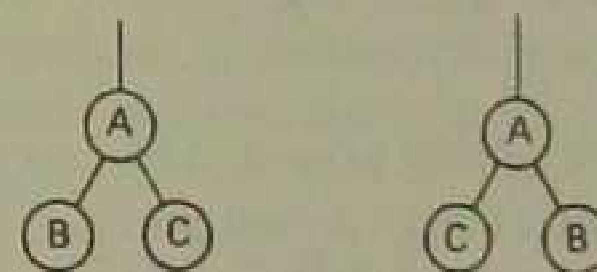
(c)



(d)

Existuje viacero spôsobov reprezentácie stromovej štruktúry. Na ilustráciu zvoľme nasledujúci príklad: Majme základný typ T reprezentovaný množinou písmen. Príslušná stromová štruktúra, vyjadrená viacerými spôsobmi, je na *obr. 4.17.*

Uvedené štyri reprezentácie zobrazujú tú istú štruktúru, graf, a sú preto ekvivalentné. Grafová štruktúra explicitne zobrazuje vzťahy vetvenia, ktoré v podstate viedli ku vzniku pojmu strom. Je to síce dosť nezvyklé, ale stromy sa najčastejšie zobrazujú v opačnom smere, t. j. obrátene, alebo inak povedané, od koreňa k vetvám. Najvyšší vrchol (A) sa bežne nazýva koreň stromu. Aj keď vieme, že skutočné stromy sú obyčajne komplikovanejšie ako naše abstrakcie, budeme naše stromové štruktúry v ďalšom nazývať jednoducho stromami.



Obr. 4.18. Dva rozdielne binárne stromy

Usporiadaný strom je taký, v ktorom sú vetvy každého vrcholu usporiadané. Z toho vyplýva, že usporiadané stromy na *obr. 4.18* sú dva rozdielne objekty. Vrchol y, ktorý je bezprostredne pod vrcholom x, sa nazýva (priamy) nasledovník vrcholu x; ak je vrchol x na i-tej úrovni, tak y bude na $(i + 1)$-ej úrovni. O vrchole x zase hovoríme, že je (priamy) predchodca vrcholu y. Koreň stromu je podľa definície na prvej úrovni. Maximálnu úroveň ľubovoľného prvku v strome nazývame jeho hĺbkou alebo výškou.

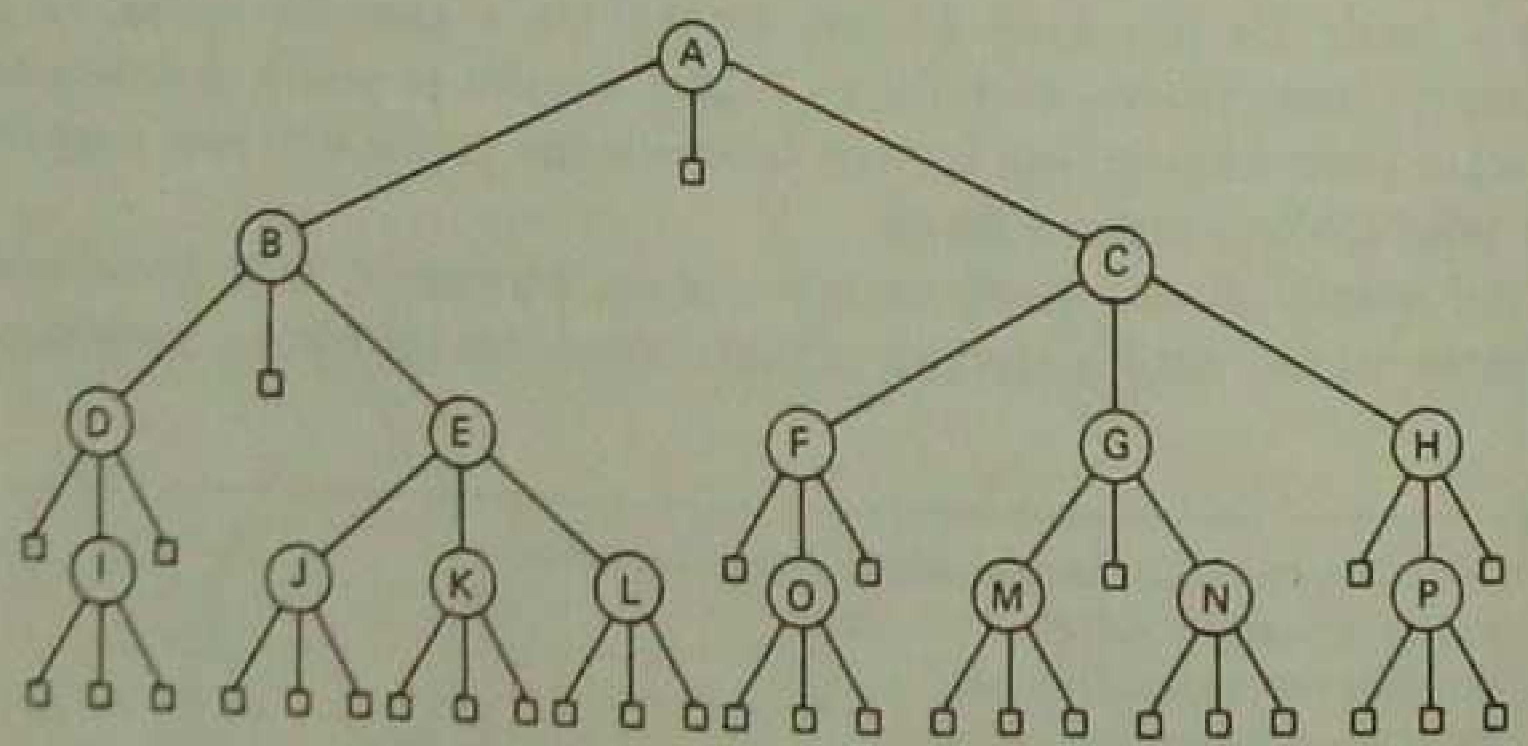
Ak nejaký prvok nemá nasledovníkov, hovoríme, že je koncovým prvkom alebo listom stromu. Prvok, ktorý nie je listom, nazývame

Obr. 4.17. Reprezentácia stromovej štruktúry pomocou:
a) množín vnorených do seba
b) zátvoriek vnorených do seba
c) viacúrovňového zarovnávania
d) grafu

vnútorným vrcholom. Počet (priamych) nasledovníkov vnútorného vrcholu nazývame jeho stupňom. Maximálny stupeň spomedzi všetkých vrcholov určuje stupeň stromu. Počet vetiev (hrán) alebo vrcholov, ktoré treba v strome prejsť, aby sme sa dostali od koreňa k vrcholu x sa nazýva *dĺžka cesty* vrcholu x. Táto dĺžka sa pre koreň rovná jednotke, pre priameho potomka koreňa dvojke atď. Vo všeobecnosti platí, že dĺžka cesty vrcholu na i-tej úrovni sa rovná hodnote i. Dĺžka cesty celého stromu je definovaná ako súčet dĺžok ciest jednotlivých vrcholov stromu. Často sa táto dĺžka nazýva tiež dĺžkou vnútornej cesty. Dĺžka vnútornej cesty stromu znázorneného napr. na *obr. 4.17* je 52. Priemerná dĺžka cesty P_I je

$$P_I = \frac{1}{n} \sum_i n_i \, . \, i \tag{4.34}$$

kde n_i je počet vrcholov na i-tej úrovni. Aby sme mohli definovať dĺžku vonkajšej cesty, rozšírime každý vrchol stromu, ktorý nemá podstrom, o špeciálny vrchol. Predpokladajme pritom, že všetky vrcholy stromu sú rovnakého stupňa, a to stupňa stromu. Rozšírenie stromu v tomto zmysle znamená vyplniť prázdne vetvy pomocou špeciálnych vrcholov, pričom tieto, pochopiteľne, nemajú nasledovníkov. Rozšírenie stromu z *obr. 4.17* prostredníctvom špeciálnych vrcholov je znázornené na *obr. 4.19*. Špeciálne vrcholy sú zobrazené pomocou štvorčekov.



Obr. 4.19. Ternárny strom rozšírený o špeciálne vrcholy

Dĺžku vonkajšej cesty môžeme teraz definovať ako súčet dĺžok ciest všetkých špeciálnych vrcholov. Ak je počet špeciálnych vrcholov na i-tej úrovni m_i, tak priemerná dĺžka vonkajšej cesty P_E je

$$P_E = \frac{1}{m} \sum_i m_i \, . \, i \tag{4.35}$$

Dĺžka vonkajšej cesty stromu, znázorneného na *obr. 4.19*, je 153.

Počet špeciálnych vrcholov m, ktoré treba pridať do stromu stupňa d, priamo závisí od počtu pôvodných vrcholov n. Uvedomme si, že do každého vrcholu vstupuje jediná hrana. V rozšírenom strome potom musí byť $m + n$ hrán. Na druhej strane z každého pôvodného vrcholu vystupuje d hrán, pričom zo špeciálnych vrcholov žiadna. Celkový počet hrán bude preto $dn + 1$, pričom tá, ktorá zvyšuje, je hrana vstupujúca do koreňa stromu. Z uvedených dvoch výsledkov dostávame nasledujúci vzťah medzi počtom špeciálnych vrcholov (m) a počtom pôvodných vrcholov (n):

$$dn + 1 = m + n$$

alebo (4.36)

$$m = (d - 1)\, n + 1$$

Maximálny počet vrcholov v strome výšky h dosiahneme, ak všetky vrcholy budú mať d podstromov okrem tých, ktoré sú na h-tej úrovni a nemajú žiadne podstromy. Pre strom stupňa d potom platí: na prvej úrovni sa nachádza 1 vrchol (koreň), na druhej úrovni bude d potomkov koreňa, na tretej úrovni bude d^2 potomkov d vrcholov z druhej úrovne atď. Na základe toho dostávame vzťah

$$N_d(h) = 1 + d + d^2 + \ldots + d^{h-1} = \sum_{i=0}^{h-1} d^i \tag{4.37}$$

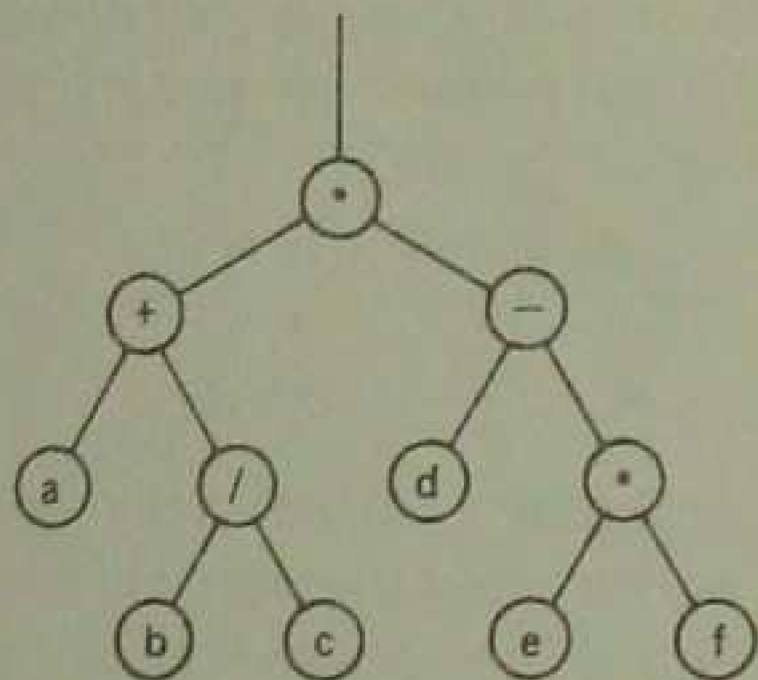
ktorý znamená maximálny počet vrcholov v strome výšky h a stupňa d. Pre $d = 2$ dostaneme

$$N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1 \tag{4.38}$$

Osobitný význam majú usporiadané stromy druhého stupňa. Nazývajú sa binárne stromy. Definujeme ich ako konečnú množinu prvkov

(vrcholov), ktorá je buď prázdna, alebo sa skladá z koreňa (vrchola) a z dvoch disjunktných stromov nazývaných ľavý a pravý podstrom koreňa. V nasledujúcom sa budeme zaoberať výlučne binárnymi stromami. Preto termín strom budeme používať na označenie pojmu usporiadaný binárny strom. Stromy s vyšším stupňom ako binárne, nazývame viaccestné stromy a stretneme sa s nimi v piatom článku tejto kapitoly.

Známymi príkladmi binárnych stromov sú rodokmene, v ktorých matka a otec tvoria nasledovníkov (!) príslušnej osoby; priebeh tenisového turnaja, kde je každá hra zobrazená vrcholom stromu označujúcim víťaza hry dvoch súperov, ktorí predstavujú nasledovníkov tohto vrcholu; alebo aritmetický výraz, ktorého binárne operátory tvoria vrcholy vetvenia v strome a operandy ich podstromy (*obr. 4.20*).

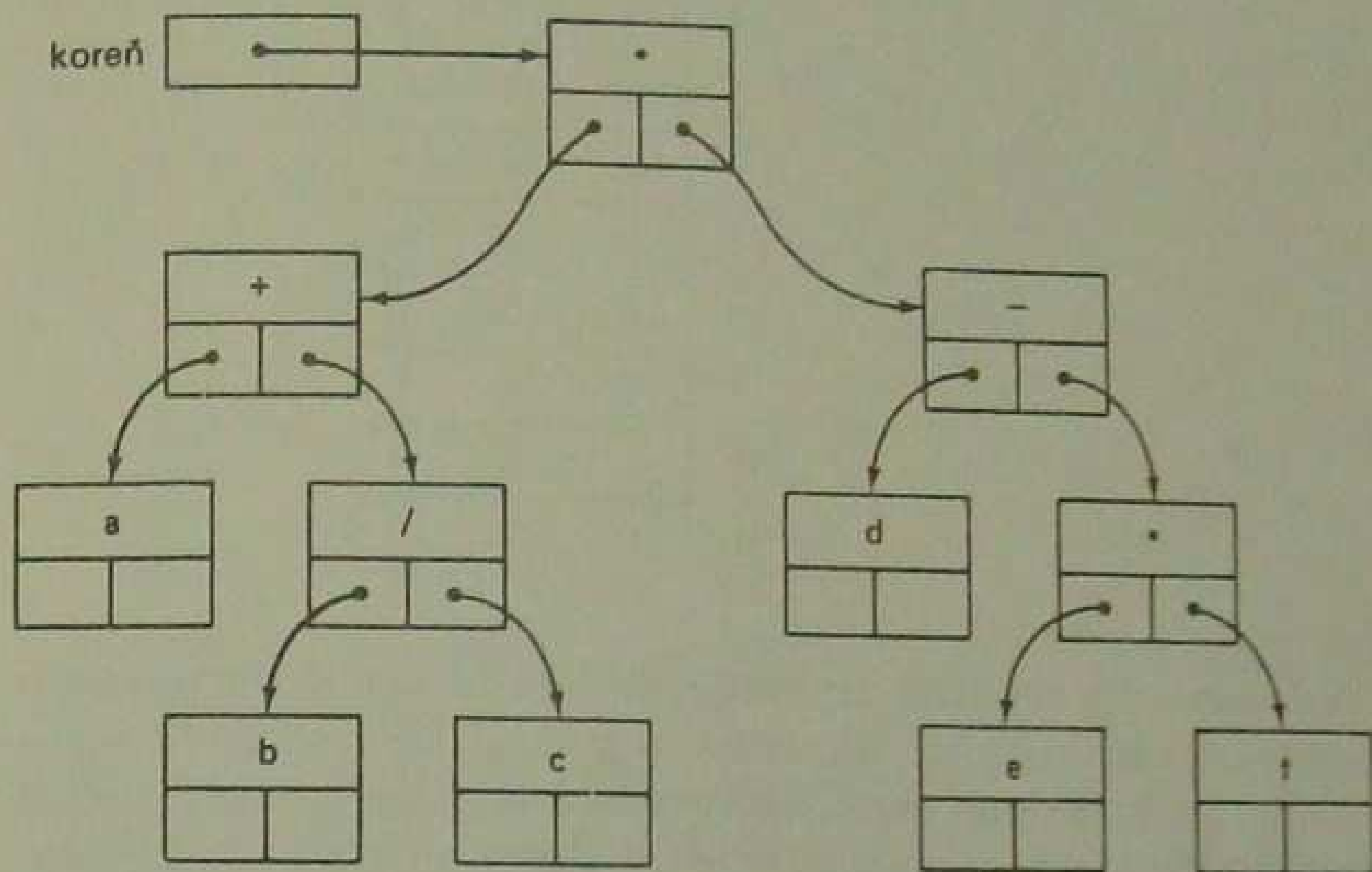


Obr. 4.20. Reprezentácia výrazu $(a + b/c) * (d - e * f)$

Vráťme sa teraz k problematike reprezentácie stromov. Vzhľadom na to, že ide o rozvetvené rekurzívne štruktúry, bude zaiste výhodné, ak využijeme dobre známe schopnosti smerníkov. Je pochopiteľné, že nemá zmysel definovať premenné s pevnou stromovou štruktúrou. Namiesto toho definujeme vrcholy stromu ako premenné s pevnou štruktúrou, t. j. pevne stanoveného typu, v rámci ktorého stupeň stromu určuje počet smerníkov na podstromy príslušného vrcholu. Referenciu na prázdny strom vyjadrujeme konštantou **nil**. Potom strom znázornený na *obr. 4.20* pozostáva z vrcholov, ktorých typ je definovaný takto:


```
type vrchol = record op: char;
                     ľavý, pravý: ↑vrchol
              end                                  (4.39)
```

Reprezentácia takéhoto stromu je na *obr. 4.21*.



Obr. 4.21. Strom zobrazený ako štruktúra údajov

Prirodzene, existujú viaceré spôsoby reprezentácie abstraktnej myšlienky stromovej štruktúry pomocou známych typov údajov, akými sú napr. polia. Zobrazenie stromu prostredníctvom poľa je najzaužívanejším spôsobom pri tých programovacích jazykoch, ktoré neumožňujú dynamické prideľovanie pamäti premenným a ich referenciu pomocou smerníkov. V takomto prípade môžeme strom, znázornený na *obr. 4.20*, reprezentovať premennou typu pole, deklarovanou ako

```
t: array [1 .. 11] of
      record op: char;                             (4.40)
             ľavý, pravý: integer
      end
```

Hodnoty jednotlivých prvkov poľa udáva *tab. 4.3*.

Tabuľka 4.3

Reprezentácia stromu pomocou poľa

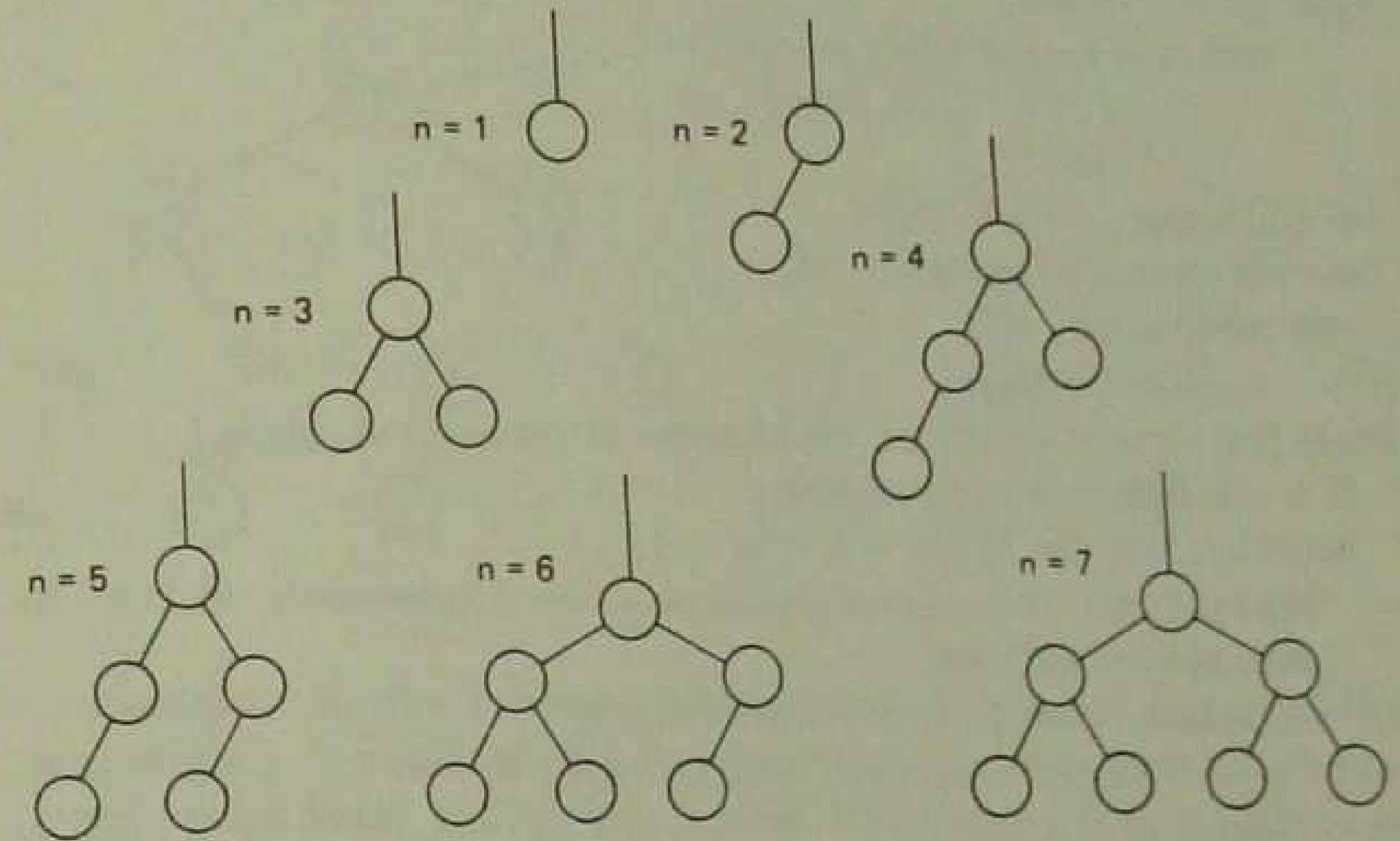
1	*	2	3
2	+	6	4
3	−	9	5
4	/	7	8
5	*	10	11
6	*a*	0	0
7	*b*	0	0
8	*c*	0	0
9	*d*	0	0
10	*e*	0	0
11	*f*	0	0

Vzhľadom na uvedenú explicitnú deklaráciu nebudeme v tomto prípade hovoriť o strome, ale radšej o poli, aj keď zodpovedajúcou abstraktnou štruktúrou (na reprezentáciu ktorej sme použili pole) je strom. Ďalšie možnosti reprezentácie stromov v programovacích jazykoch a systémoch, ktoré neposkytujú možnosť dynamického prideľovania pamäti, už nebudeme rozoberať, pretože predpokladáme, že všetky moderné programovacie jazyky a systémy takýto prostriedok majú, alebo ho budú mať.

Skôr ako začneme skúmať výhody používania stromov a spôsoby aplikovania rôznych operácií na ne, uvedieme príklad vytvorenia stromu programom. Predpokladajme, že strom, ktorý chceme vytvoriť, obsahuje vrcholy, ktorých typ je definovaný v deklarácii (4.39). Nech sú tieto vrcholy očíslované od 1 po n, pričom ich hodnoty získame načítaním zo vstupného súboru.

Aby bol problém o niečo zložitejší, zostrojíme strom, ktorý má n vrcholov a minimálnu výšku. Ak chceme vytvoriť strom s minimálnou výškou, bude potrebné, aby sme na každej úrovni, okrem tej najnižšej, umiestnili maximálny možný počet vrcholov. To sa dá jednoducho dosiahnuť tým, že jednotlivé vrcholy umiestnime rovnomerne na pravú

a ľavú pozíciu nasledovníka príslušného vrcholu. Takejto distribúcii vrcholov hovoríme štruktúrovanie stromu a pre dané n (v našom prípade: $n = 1, 2, \ldots, 7$) je znázornené na *obr. 4.22*.



Obr. 4.22. Dokonale vyvážené stromy

Pravidlo rovnomernej distribúcie známeho počtu n vrcholov sa dá najlepšie formulovať rekurzívne takto:

1. Zvoľme jeden vrchol za koreň stromu.
2. Vytvorme ľavý podstrom s počtom vrcholov $nl = n$ **div** 2.
3. Vytvorme pravý podstrom s počtom vrcholov $nr = n - nl - 1$.

Toto pravidlo, vyjadrené v tvare rekurzívnej procedúry, tvorí súčasť programu 4.3, ktorý načíta zo vstupného súboru hodnoty vrcholov (1 až n) a skonštruuje dokonale vyvážený strom.

Definícia dokonale vyváženého stromu vyzerá takto: *Strom je dokonale vyvážený*, ak pre každý vrchol platí, že počet vrcholov v jeho ľavom a pravom podstrome sa líši najviac o jednotku.

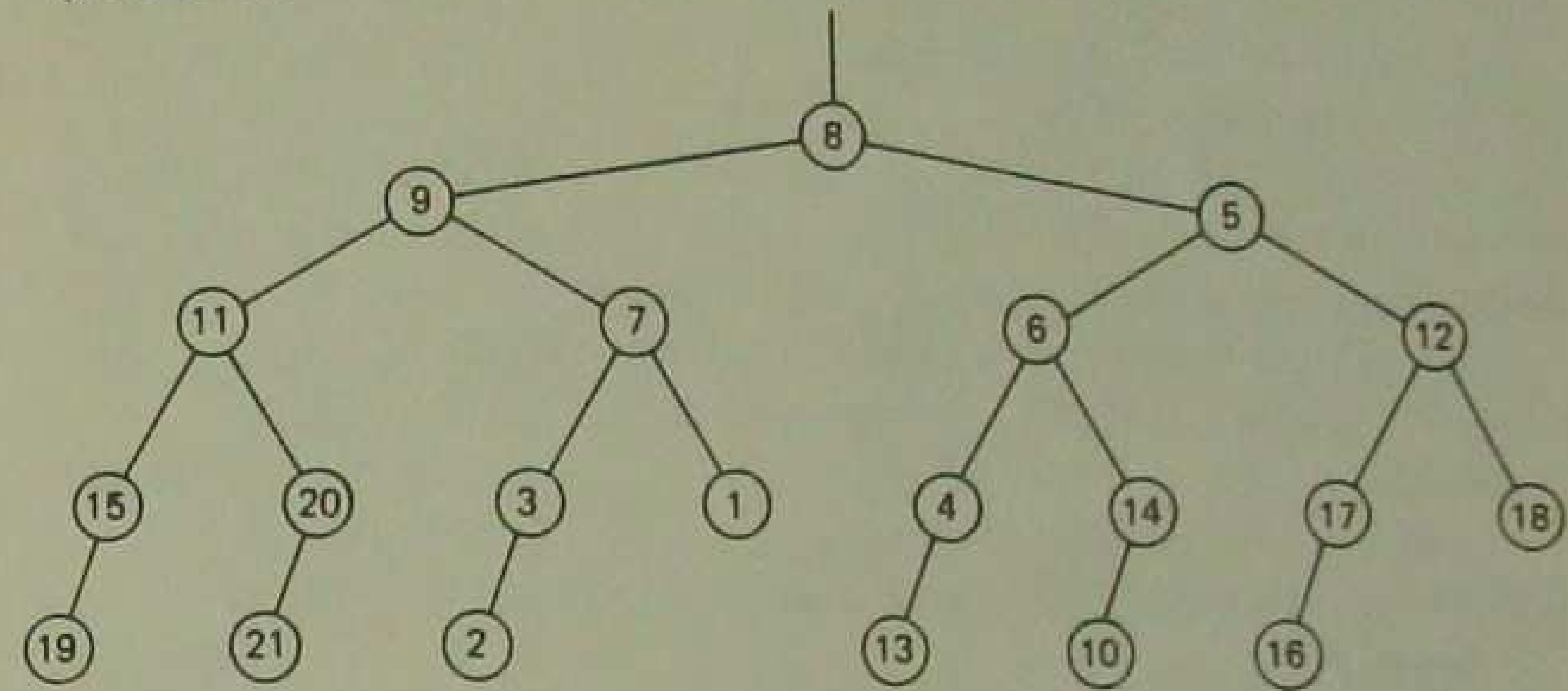
Predpokladajme napr., že vstupný súbor obsahuje 21 hodnôt vrcholov stromu:

21 8 9 11 15 19 20 21 7 3 2
1 5 6 4 13 14 10 12 17 16 18

PROGRAM 4.3. *Vytvorenie dokonale vyváženého stromu*

```
program Vytvorstrom (input, output);
type ref = ↑vrchol;
     vrchol = record kľúč: integer;
                     ľavý, pravý: ref
              end;
var n: integer; koreň: ref;
function strom (n: integer): ref;
  var novývrchol: ref;
      x, nl, nr: integer;
begin {vytvorenie dokonale vyváženého stromu s n vrcholmi}
  if n = 0 then strom: = nil else
  begin nl: = n div 2; nr: = n - nl - 1;
    read (x); new (novývrchol);
    with novývrchol↑ do
      begin kľúč: = x; ľavý: = strom (nl);
                       pravý: = strom (nr);
      end;
    strom: = novývrchol
  end
end {strom};
procedure vytlačstrom (t: ref; h: integer);
  var i: integer;
begin {vytlačenie stromu t so zarovnaním z}
  if t ≠ nil then
  with t↑ do
  begin vytlačstrom (ľavý, h + 1);
    for i: = 1 to h do write (' ');
    writeln (kľúč);
  vytlačstrom (pravý, h + 1)
  end
end {vytlačstrom};
begin {hodnota prvého celého čísla n udáva počet vrcholov}
  read (n); koreň: = strom (n); vytlačstrom (koreň, 0)
end.
```


Prostredníctvom programu 4.3 dostaneme dokonale vyvážený strom (*obr. 4.23*).



Obr. 4.23. Strom vytvorený pomocou programu 4.3

Všimnime si, ako rekurzívne procedúry zjednodušujú a spresňujú program 4.3. Je pochopiteľné, že rekurzívne algoritmy sú zvlášť vhodné v tých prípadoch, keď aj informačné štruktúry, s ktorými program manipuluje, sú rekurzívne definované. Je to vidieť i v procedúre, ktorá realizuje výstupnú tlač vytvoreného stromu: prázdny strom sa nevytlačí, podstrom na L-tej úrovni sa tlačí na tri etapy; v prvej sa vytlačí jeho ľavý podstrom, v druhej sám vrchol, primerane zarovnaný pomocou L predsunutých medzier, a nakoniec sa vytlačí jeho pravý podstrom.

Výhoda rekurzívneho algoritmu sa prejaví najmä vtedy, keď ho porovnáme s nerekurzívnou verziou. Odporúčame preto čitateľovi, aby predtým, než sa pozrie na program (4.41), použil všetok svoj um a dôvtip na napísanie nerekurzívnej verzie algoritmu uvedeného generátora stromu. Tento nerekurzívny algoritmus (4.41) uvádzame bez ďalších poznámok, pretože chceme, aby sa čitateľ sám pokúsil odhaliť, ako a prečo pracuje.

```
program Vytvorstrom (input, output);
type ref = ↑vrchol;
     vrchol = record kľúč: integer;
                     ľavý, pravý: ref
              end;
```

```
var i, n, nl, nr, x: integer;
    koreň, p, q, r, dmy: ref;
    s: array [1..30] of {zásobník}
        record n: integer;
               rf: ref
        end;
begin {prvé celé číslo určuje počet vrcholov}
  read (n); new (koreň); new (dmy); {fiktívny vrchol}
  i: = 1; s[1].n: = n; s[1].rf: = koreň;
  repeat n: = s[i].n; r: = s[i].rf; i: = i - 1;                    (4.41)
    {odobratie z vrcholu zásobníka}
    if n = 0 then r↑.pravý: = nil else
    begin p: = dmy;
      repeat nl: = n div 2; nr: = n - nl - 1;
        read (x); new (q); q↑.kľúč: = x;
        i: = i + 1; s[i].n: = nr; s[i].rf: = q;
        {vloženie do zásobníka}
        n: = nl; p↑.ľavý: = q; p: = q
      until n = 0
      q↑.ľavý: = nil; r↑.pravý: = dmy↑.ľavý
    end
  until i = 0;
  vytlačstrom (koreň↑.pravý, 0)
end.
```

4.4.2 ZÁKLADNÉ OPERÁCIE NA BINÁRNYCH STROMOCH

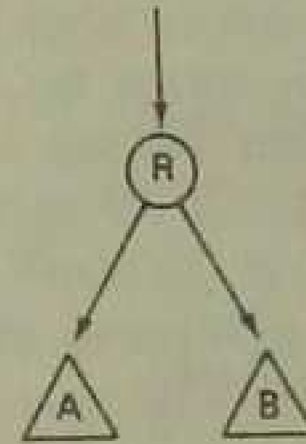
Existuje mnoho úloh, ktoré sa dajú riešiť na stromovej štruktúre; najbežnejšou je uskutočnenie danej operácie P na každom prvku stromu. Operáciu P chápeme ako parameter všeobecnejšej úlohy navštívenia všetkých vrcholov stromu, ktoré sa zvyčajne nazýva prechod stromu.

Ak sa na túto úlohu pozeráme ako na jednoduchý sekvenčný procec, je jasné, že jednotlivé vrcholy v strome budú navštívené v určitom špecifickom poradí a možno sa na ne pozerať, ako keby boli lineárne

usporiadané. V skutočnosti je opis mnohých algoritmov oveľa jednoduchší, ak vychádzame z toho, že spracovanie každého ďalšieho prvku v strome je možné na základe príslušného usporiadania.

Rozlišujeme tri základné usporiadania, ktoré sú prirodzeným dôsledkom stromovej štruktúry. Podobne ako samotná štruktúra aj ony sa dajú vhodne rekurzívne vyjadriť. V súlade s binárnym stromom na *obr. 4.24*, v ktorom symbolom R označujeme vrchol stromu, a symbolmi A, B ľavý a pravý podstrom, sú to tieto tri usporiadania:

1. Priame (preorder): R, A, B (najprv sa navštívil koreň a potom podstromy).
2. Vnútorné (inorder): A, R, B.
3. Spätné (postorder): A, B, R (koreň sa navštívi až po podstromoch).



Obr. 4.24. Binárny strom

Zoberme si druhý príklad, t.j. binárny strom na *obr. 4.20*. Prechodom tohto stromu a zaznamenávaním jednotlivých znakov, s ktorými prídeme do styku pri návšteve konkrétneho vrcholu, dostaneme tieto usporiadania:

1. Priame: $* + a/bc - d * ef$.
2. Vnútorné: $a + b/c * d - e * f$.
3. Spätné: $abc/ + def * - *$.

Rozoznávame tri druhy výrazov: priamy prechod stromu, reprezentujúceho daný výraz, poskytuje *prefixový zápis*, spätný prechod *postfixový* a vnútorný prechod *infixový zápis*. Všimnime si, že v týchto prípadoch nepotrebujeme zátvorky vyjadrujúce prioritu operátorov.

Pokúsme sa teraz sformulovať uvedené tri spôsoby prechodu stromom pomocou troch konkrétnych programov. Na označenie príslušného stromu v nich použijeme explicitný parameter t a na vyjadrenie

operácie, ktorá sa má uskutočniť na každom vrchole, implicitný parameter P. Predpokladajme tieto definície:

```
type ref = ↑vrchol
   vrchol = record ...                                    (4.42)
              ľavý, pravý: ref
            end
```

Tri metódy prechodu stromom môžeme bez ťažkostí formulovať ako rekurzívne procedúry; opäť sa potvrdzuje skutočnosť, že operácie na rekurzívne definovaných štruktúrach údajov sa najvhodnejšie definujú prostredníctvom rekurzívnych algoritmov.

```
procedure priamyprechod (t: ref);
begin if t ≠ nil then                                     (4.43)
   begin P(t);
      priamyprechod (t↑.ľavý);
      priamyprechod (t↑.pravý);
   end
end
procedure vnútornýprechod (t: ref);
begin if t ≠ nil then
   begin vnútornýprechod (t↑.ľavý);                      (4.44)
      P(t);
      vnútornýprechod (t↑.pravý);
   end
end
procedure spätnýprechod (t: ref);
begin
   if t ≠ nil then
      begin spätnýprechod (t↑.ľavý);                     (4.45)
         spätnýprechod (t↑.pravý);
         P(t)
      end
end
```

Všimnite si, že pri volaní každej z uvedených troch procedúr je smerník t posielaný svojou hodnotou. To vyjadruje skutočnosť, že

podstatnou veličinou je referencia na uvažovaný podstrom, a nie premenná, ktorej hodnotou je smerník a ktorá by sa mohla zmeniť, keby bol smerník t posielaný referenciou.

Príkladom algoritmu prechodu stromom môže byť procedúra, ktorá vytlačí príslušný strom na výstupný protokol a súčasne uskutoční patričné grafické zarovnanie úrovne každého vrcholu (pozri program 4.3).

Binárne stromy sa často používajú na reprezentáciu množiny údajov, ktorej prvky sa majú vyberať na základe konkrétneho kľúča. Ak usporiadame jednotlivé vrcholy stromu takým spôsobom, že pre každý vrchol t_i budú všetky kľúče ľavého podstromu vrcholu t_i menšie ako kľúč vrcholu t_i a všetky kľúče v pravom podstrome väčšie ako kľúč vrcholu t_i, dostaneme strom, ktorý nazývame vyhľadávací. Vyhľadanie konkrétneho prvku sa v takomto strome uskutočňuje veľmi jednoduchým spôsobom: Začneme pri koreni stromu a ďalej pokračujeme cestou (buď v ľavom, alebo pravom podstrome), ktorú nám určuje výsledok porovnania hodnoty kľúča hľadaného prvku s hodnotou kľúča príslušného vrcholu. Vieme už, že z n prvkov môžeme zostrojiť binárny strom, ktorého výška nie je väčšia ako $\log n$. Ak by tento strom bol dokonale vyvážený, tak sa vyhľadávanie dá uskutočniť na základe porovnaní, ktorých celkový počet nie je väčší ako $\log n$. Vo všeobecnosti je strom oveľa vhodnejšou štruktúrou na organizovanie uvedených údajov než zoznam.

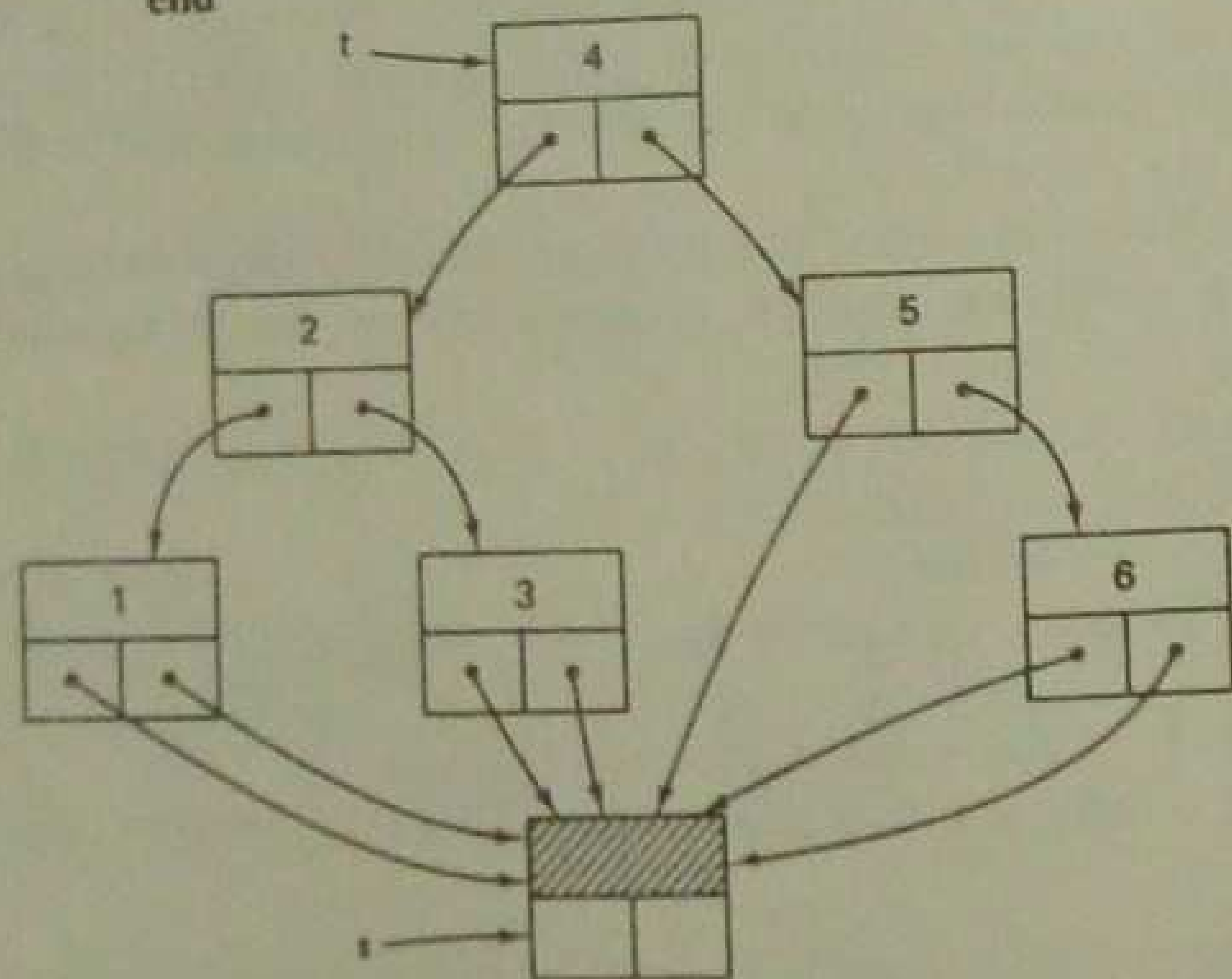
Vyhľadávací proces od koreňa až po hľadaný vrchol môžeme jednoducho a prehľadne vyjadriť prostredníctvom iteračného algoritmu (4.46).

```
function loc (x: integer; t: ref): ref;
  var nájdený: boolean;
begin nájdený: = false;
  while (t ≠ nil) ∧ ¬ nájdený do                          (4.46)
  begin
      if t↑.kľúč = x then nájdený: = true else
      if t↑.kľúč > x then t: = t↑.ľavý else t: = t↑.pravý
  end;
  loc: = t
end
```

Funkcia loc(*x*, *t*) vráti hodnotu **nil**, ak sa hľadaný kľúč s hodnotou *x* nenachádza v strome s koreňom *t*. Podobne ako v prípade vyhľadávania v zoznamoch, je možné zjednodušiť a zefektívniť pomerne zložitú podmienku ukončenia vyhľadávania. Riešenie je opäť založené na princípe použitia zarážky. Aplikácia smerníkov nám umožní ukončiť vetvenie všetkých vrcholov stromu v jedinom vrchole — zarážke. Takto upravená štruktúra predstavuje strom, ktorého všetky listy sú zviazané a ukotvené v jednom bode, ktorým je práve spomínaná zarážka (*obr. 4.25*). Zarážku môžeme považovať za spoločného a jediného zástupcu všetkých špeciálnych vrcholov, o ktoré sa pôvodný strom rozšíril (pozri *obr. 4.19*). Výsledná zjednodušená vyhľadávacia procedúra je vyjadrená algoritmom (4.47).

```
function loc(x: integer; t: ref): ref;
begin s↑.kľúč := x; {zarážka}
  while t↑.kľúč ≠ x do                                    (4.47)
    if x < t↑.kľúč then t := t↑.ľavý
                   else t := t↑.pravý;

  loc := t
end
```



Obr. 4.25. Vyhľadávací strom so zarážkou

Poznamenávame, že ak v tomto prípade funkcia loc(*x*, *t*) nenájde v strome *t* kľúč s hodnotou *x*, vráti hodnotu *s*, t. j. smerník, na zarážku. Takýto smerník vlastne reprezentuje smerník s hodnotou **nil**.

4.4.3 VYHĽADÁVANIE A PRIDÁVANIE DO STROMOV

Ťažko možno ukázať silu a účinnosť techniky dynamického prideľovania pamäti a metódy prístupu prostredníctvom smerníkov na tých príkladoch, v ktorých sa vytvorená množina údajov v priebehu výpočtu nemení. Oveľa vhodnejšími sú príklady aplikácií, v ktorých sa samotná stromová štruktúra mení, t. j. narastá alebo sa zmenšuje. Tieto príklady tiež dokumentujú nemožnosť použitia takých štruktúr, akými sú napr. polia a potvrdzujú výhodnosť použitia stromových štruktúr s prvkami pospájanými pomocou smerníkov.

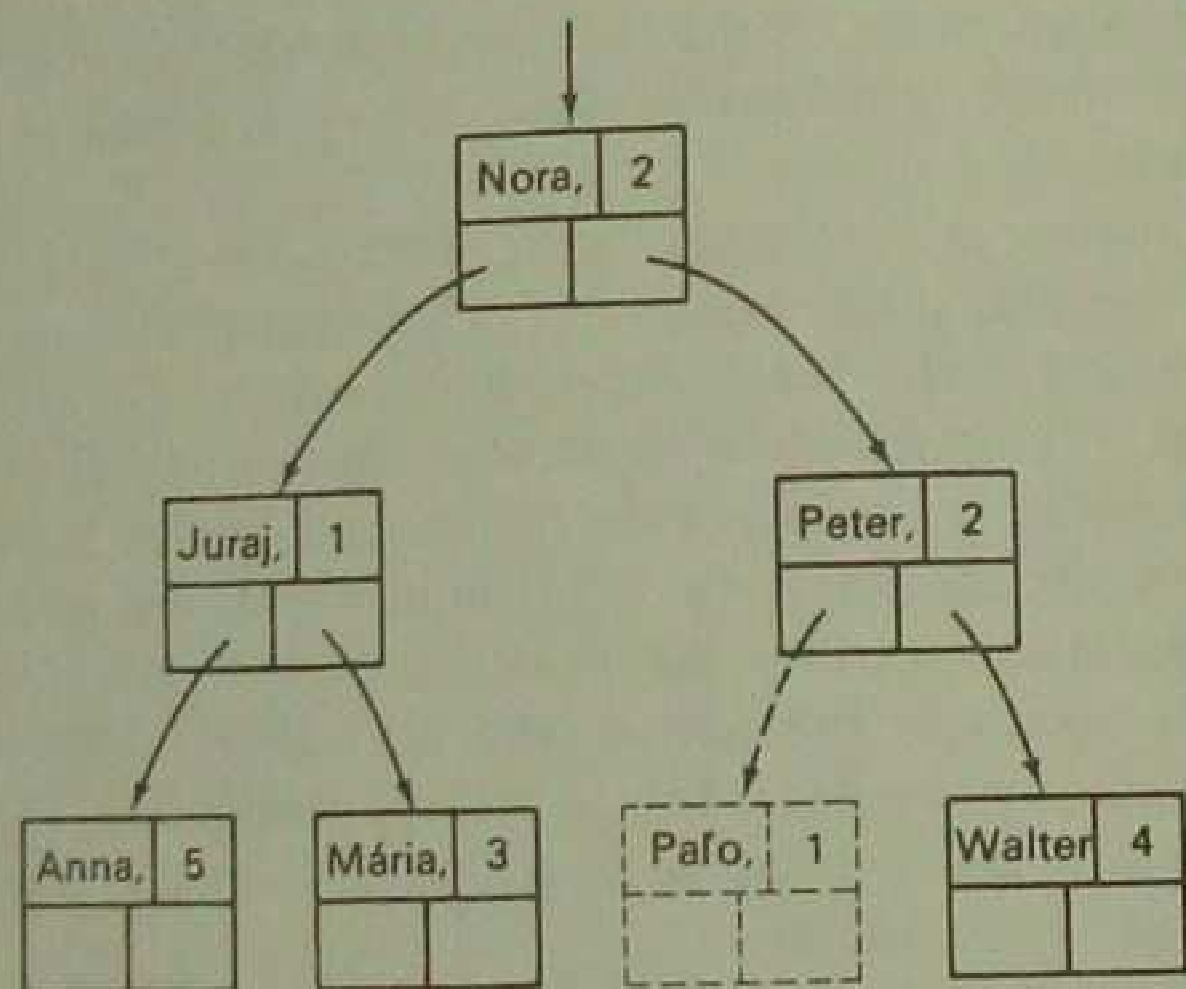
Majme najprv príklad o stromoch, ktoré počas výpočtu programu narastajú, ale nikdy sa nezmenšujú. Vhodným príkladom môže byť problém konkordancie, ktorý sme skúmali v súvislosti so zoznamovými štruktúrami. Na osvieženie si ho stručne zopakujeme: Bola daná postupnosť slov a bolo potrebné zistiť frekvencie výskytov každého slova. To znamená, že zakaždým treba príslušné slovo vyhľadať v strome, ktorý je spočiatku prázdny. Ak sa nájde, zvýši sa údaj o počte jeho výskytov; v opačnom prípade sa (ako nové slovo) pridá do stromu (pričom údaju o počte výskytov sa priradí hodnota 1). Tento proces nazývame *vyhľadávanie v strome spojené s pridávaním*. Predpokladajme tieto definície typov údajov:

```
type ref = ↑slovo;
  slovo = record kľúč: integer;
                 počet: integer;                          (4.48)
                 ľavý, pravý: ref
          end
```

Predpokladajme navyše, že sa vstupné údaje nachádzajú v súbore *f*. Nech je koreň stromu označený premennou typu smerník. Potom môžeme náš program vyjadriť nasledujúcim algoritmom

```
reset(f)
while ¬ eof(f) do                                              (4.49)
  begin read(f, x); vyhľadaj(x, koreň)
  end
```

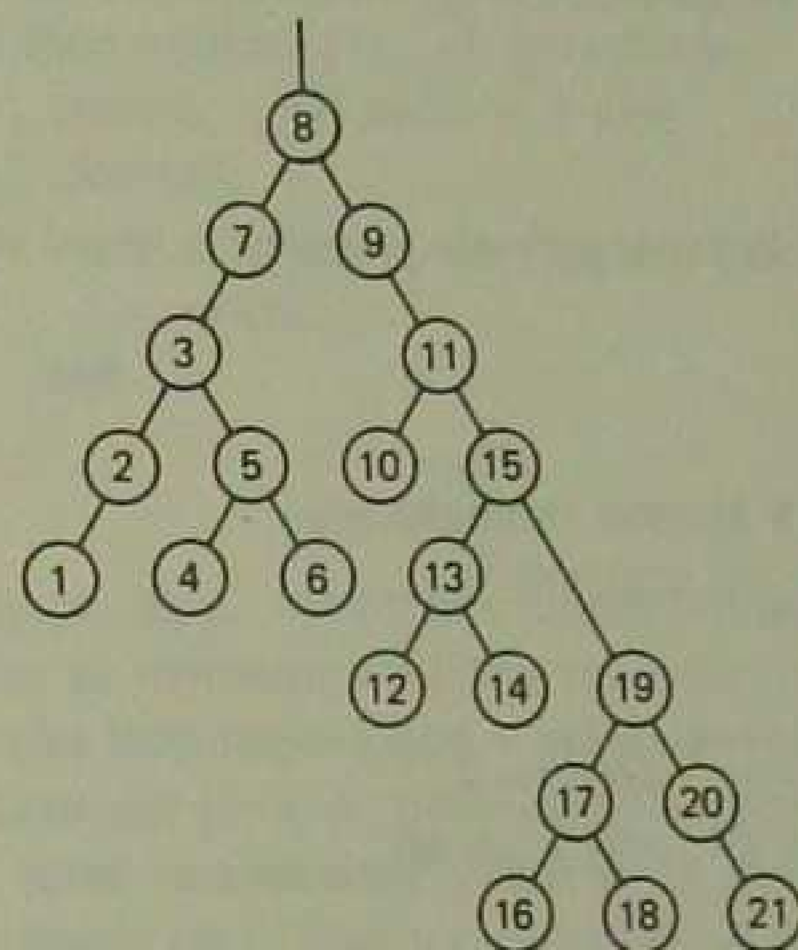
Zisťujeme, že vyhľadávanie je opäť priamočiare. Ak by však viedlo do „slepej uličky" (t.j. do prázdneho podstromu určeného hodnotou **nil** príslušného smerníka), museli by sme dané slovo pridať do stromu namiesto prázdneho podstromu. Uvažujme napr. o binárnom strome na *obr. 4.26*, do ktorého chceme pridať slovo Paľo. Výsledok znázornený prerušovanou čiarou je na tom istom obrázku.



Obr. 4.26. Pridanie prvku do usporiadaného binárneho stromu

Úplná operácia sa nachádza v programe 4.4. Vyhľadávací proces je formulovaný rekurzívnou procedúrou vyhľadaj. Všimnime si, že skutočný parameter *p* sa pri vyvolaní tejto procedúry posiela referenciou, a nie hodnotou. Toto je dôležité, pretože pri pridaní prvku do stromu sa musí hodnota smerníka (určujúceho tento nový prvok) priradiť premennej, ktorej hodnotou bola predtým konštanta **nil**. Ak pre program 4.4 použijeme tú istú vstupnú postupnosť 21 čísel, akú sme použili

v prípade programu 4.3, ktorý vytvoril strom na *obr. 4.23*, dostaneme binárny vyhľadávací strom znázornený na *obr. 4.27*.



Obr. 4.27. Vyhľadávací strom vytvorený programom 4.4

PROGRAM 4.4. *Stromové vyhľadávanie a pridávanie*

```
program Stromovévyhľadávanie(input, output);
{vyhľadávanie slov v binárnom strome a ich pridávanie}
type ref = ↑slovo;
     slovo = record kľúč: integer;
                    počet: integer;
                    ľavý, pravý: ref
             end;
var koreň: ref;
    k: integer;
procedure vytlačstrom(w: ref; l: integer);
  var i: integer;
begin if w ≠ nil then
      with w↑ do
      begin vytlačstrom(ľavý, l + 1);
```

```
        for i: = 1 to l do write(' ');
        writeln (kľúč);
        vytlačstrom (pravý, l + 1)
      end
end;

procedure vyhľadaj (x: integer; var p: ref);
begin
  if p = nil then
  begin
        {slovo sa v strome nenachádza;
         pridaj ho do stromu}
    new (p);
    with p↑ do
      begin kľúč: = x; počet: = 1;
            ľavý: = nil; pravý: = nil
      end
  end else
  if x < p↑.kľúč then vyhľadaj (x, p↑.ľavý) else
  if x > p↑.kľúč then vyhľadaj (x, p↑.pravý) else
    p↑.počet: = p↑.počet + 1
end {vyhľadaj};
begin
  koreň: = nil;
  while ¬ eof(input) do
    begin read (k);
          vyhľadaj (k, koreň)
    end;
  vytlačstrom (koreň, 0)
end.
```

Použitie zarážky v schéme (4.50) opäť o niečo zjednodušuje riešenie úlohy. Na začiatku realizácie programu však musíme premennej *koreň* priradiť hodnotu smerníka, ukazujúceho na zarážku (namiesto hodnoty **nil**), a pred začatím každého vyhľadávania treba hľadanú hodnotu x priradiť kľúčovej zložke zarážky.


```
procedure vyhľadaj (x: integer; var p: ref);
begin
  if x < p↑.kľúč then vyhľadaj (x, p↑.ľavý) else
  if x > p↑.kľúč then vyhľadaj (x, p↑.pravý) else                    (4.50)
  if p ≠ s then p↑.počet: = p↑.počet + 1 else
    begin {pridaj} new (p);
      with p↑ do begin kľúč: = x; ľavý: = s; pravý: = s;
                       počet: = 1
                 end
    end
end
```

Ešte raz, a naposledy, vytvoríme alternatívnu verziu uvedeného programu, pričom sa vyhneme použitiu rekurzie. Odstránenie rekurzie však nebude až také jednoduché ako v prípade vyhľadávania, ktoré nie je spojené s pridávaním prvkov, pretože pridávanie vyžaduje informáciu o prejdenej ceste (maximálne jeden krok dozadu). Táto funkcia zapamätania prejdenej cesty bola v programe 4.4 automaticky zabezpečená prostredníctvom parametra, posielaného referenciou.

Aby sme mohli príslušný prvok správne pridať do stromu, musíme poznať odkaz na jeho predchodcu a navyše vedieť, či má byť prvok pridaný ako ľavý alebo pravý podstrom tohto predchodcu. Za týmto účelom zavedieme dve premenné $p2$ a d (na označenie smeru).

```
procedure vyhľadaj (x: integer; koreň: ref);
  var p1, p2: ref; d: integer;
begin p2: = koreň; p1: = p2↑.pravý; d: = 1;
  while (p1 ≠ nil) ∧ (d ≠ 0) do
  begin p2: = p1;
    if x < p1↑.kľúč then
      begin p1: = p1↑.kľúč; d: = −1 end else
    if x > p1↑.kľúč then
      begin p1: = p1↑.pravý; d: = 1 end else
    d: = 0                                                            (4.51)
  end;
  if d = 0 then p1↑.počet: = p1↑.počet + 1 else
    begin {pridaj} new (p1);
```

```
        with p1↑ do
        begin kľúč := x; ľavý := nil; pravý := nil;
              počet := 1
        end;
        if d < 0 then p2↑.ľavý := p1 else p2↑.pravý := p1
    end
end
```

Smerníky $p1$ a $p2$ sa použijú tým istým spôsobom ako v prípade zoznamového vyhľadávania a pridávania, t.j. smerníkom $p2$ bude vždy označený predchodca prvku $p1\uparrow$. Aby sme mohli, vzhľadom na tieto okolnosti, zahájiť vyhľadávací proces, je potrebné zaviesť pomocný fiktívny prvok, označený smerníkom *koreň*. Začiatok skutočného vyhľadávacieho stromu je určený smerníkom *koreň*↑.*pravý*. Preto musí program začínať príkazmi

new (*koreň*); *koreň*↑.*pravý* := **nil**

namiesto pôvodného priradenia

koreň := **nil**

Aj keď hlavným cieľom tohto algoritmu je vyhľadávanie, dá sa výhodne použiť i na triedenie. V skutočnosti algoritmus pripomína triedenie metódou priameho vkladania a pretože namiesto poľa využíva stromovú štruktúru, odpadá potreba dodatočného preusporiadania prvkov, ovplyvnených pridaním alebo premiestnením nejakého prvku. Stromové triedenie sa dá naprogramovať tak, že je skoro tak efektívne ako najlepšie známe metódy vnútorného triedenia. Pravda, musíme urobiť niektoré opatrenia. Prípad výskytu zhodného kľúča treba rozhodne riešiť iným spôsobom. Ak sa prípad $x = p\uparrow.kľúč$ spracúva zhodne s prípadom $x > p\uparrow.kľúč$, daný algoritmus reprezentuje stabilnú metódu triedenia, t.j. prvky so zhodnými kľúčmi sa budú pri prehľadávaní stromu v normálnom poradí vyskytovať v tej istej postupnosti, ako boli pridávané.

Prirodzene, existujú aj lepšie metódy triedenia, ale v aplikáciách, ktoré si vyžadujú vyhľadávanie i triedenie, sa algoritmus stromového

vyhľadávania a pridávania veľmi odporúča. Uplatní sa aj pri organizovaní objektov, ktoré sa majú uchovávať a vyberať, napr. v rámci kompilátorov jazykov a bánk údajov. Vhodným príkladom môže byť konštrukcia zoznamu krížových odkazov daného textu. Pozrime sa na tento problém podrobnejšie.

Našou úlohou je zostrojiť program, ktorý (okrem toho, že číta textový súbor f a tlačí ho na výstupný súbor spolu s číslami riadkov) zbiera a uchováva jednotlivé slová tohto textu spolu s číslami riadkov, v ktorých sa vyskytujú. Po skončení tohto procesu (t.j. po prečítaní a spracovaní celého súboru f) treba vytlačiť tabuľku obsahujúcu všetky nazhromaždené slová v utriedenom tvare, vrátane zoznamu ich výskytov.

Vyhľadávací strom (nazývaný aj lexikografický strom) je nepochybne najvhodnejší kandidát na reprezentáciu slov vyskytujúcich sa v texte. Každý vrchol stromu bude obsahovať jednak príslušné slovo textu ako svoju kľúčovú hodnotu, jednak začiatok zoznamu čísel riadkov výskytu. Každý záznam výskytu budeme nazývať *položka*. Vidíme, že v tomto príklade sa stretávame so stromovými štruktúrami a súčasne i lineárnymi zoznamami. Program sa skladá z dvoch častí (pozri program 4.5), a to z prehľadávacej fázy a tlačenia tabuľky. Druhá fáza je priamočiarou aplikáciou procedúry prechodu stromom, pričom navštívenie každého vrcholu znamená vytlačenie jeho kľúčovej hodnoty (textového slova) a prehľadanie k nemu pripojeného zoznamu čísel riadkov (položiek).

Predtým než uvedieme kompletný program generátora krížových odkazov (program 4.5), uvedieme niekoľko dôležitých poznámok:

1. Slovo chápeme ako ľubovoľnú postupnosť písmen a číslic, začínajúcu písmenom.

2. Iba prvých $c1$ znakov sa uchováva vo forme kľúča. Preto dve slová, ktoré sa neodlišujú vo svojich prvých $c1$ znakoch, sa považujú za identické.

3. Týchto $c1$ znakov sa v zhustenom tvare priradí do poľa *id* (typu alfa). Ak je $c1$ dostatočne malé, mnohé počítače dokážu porovnať dve takéto zhustené polia jedinou inštrukciou.

4. Premennou $k1$ označujeme index, ktorý udržiava nasledujúcu invariantnú podmienku, platnú pre pole znakov a:

$$a[i] = ' ' \quad \text{pre} \quad i = k1 + 1, \ldots, c1$$

Slová, ktoré sa skladajú z menšieho počtu znakov, ako určuje konštanta $c1$, sú doplnené zodpovedajúcim počtom medzier.

5. Požadujeme, aby sa čísla riadkov v zozname krížových odkazov vytlačili vo vzostupnom poradí. Preto musia byť zoznamy položiek generované v tom istom poradí, v akom sú prehľadávané pri tlači. Táto požiadavka nás núti k tomu, aby sme pripojili dva smerníky ku každému vrcholu, z ktorých prvý ukazuje na prvú položku a druhý na poslednú položku v zozname.

6. Prehľadávací program je vytvorený takým spôsobom, že slová, uvedené v úvodzovkách a ako poznámky, sa nezahŕňajú do zoznamu krížových odkazov. Predpokladáme pritom, že tieto jazykové konštrukcie neprekročia maximálny prístupný počet znakov v riadku.

PROGRAM 4.5. *Generátor krížových odkazov*

```
program Križrefgen (f, output);
{generátor krížových odkazov využívajúci binárny strom}
const c1 = 10;      {dĺžka slov}
      c2 = 8;       {počet čísel v riadku}
      c3 = 6;       {počet číslic v čísle}
      c4 = 9999;    {maximálny počet riadkov textu}
type alfa = packed array [1 .. c1] of char;
     slovref = ↑slovo;
     polref = ↑položka;
     slovo = record kľúč: alfa;
                    prvý, posledný: polref;
                    ľavý, pravý: slovref
             end;
     položka = packed record
                   lno: 0 .. c4;
                   další: polref
               end;
var koreň: slovref;
    k, k1: integer;
    n: integer;      {číslo bežného riadku}
    id: alfa;
    f: text;
    a: array [1 .. c1] of char;
procedure vyhľadaj (var w1: slovref);
  var w: slovref; x: polref;
begin w: = w1;
  if w = nil then
  begin new (w); new (x);
    with w↑ do
    begin kľúč: = id; ľavý: = nil; pravý: = nil;
          prvý: = x; posledný: = x
    end;
    x↑.lno: = n; x↑.další: = nil; w1: = w
  end else
  if id < w↑.kľúč then vyhľadaj (w↑.ľavý) else
  if id > w↑.kľúč then vyhľadaj (w↑.pravý) else
  begin new (x); x↑.lno: = n; x↑.další: = nil;
        w↑.posledný↑.další: = x; w↑.posledný: = x
  end
end {vyhľadaj};
procedure vytlačstrom (w: slovref);
  procedure vytlačslovo (w: slovo);
    var l: integer; x: polref;
  begin write (' ', w.kľúč);
      x: = w.prvý; l: = 0;
      repeat if l = c2 then
             begin writeln;
                   l: = 0; write (' ': c1 + 1)
             end;
             l: = l + 1; write (x↑.lno: c3); x: = x↑.další
      until x = nil;
      writeln
  end {vytlačslovo};
begin
  if w ≠ nil then
  begin vytlačstrom (w↑.ľavý);
```

```
        vytlačslovo (w↑);
        vytlačstrom (w↑. pravý)
  end
end {vytlačstrom};
begin koreň : = nil ; n : = 0; k1 : = c1;
  page (výstup);
  reset (f);
  while ¬ eof (f) do
  begin if n = c4 then n : = 0;
    n : = n + 1; write (n: c3);   {ďalši riadok}
    write (' ');
    while ¬ eoln (f) do
    begin {analýza neprázdneho riadku}
      if f↑ in ['A' .. 'Z'] then
      begin k : = 0;
        repeat
          if k < c1 then
          begin k : = k + 1; a[k] : = f↑;
          end ;
          write (f↑); get (f)
        until ¬ (f↑ in ['A' .. 'Z', '0' .. '9']);
        if k ≧ k1 then k1 : = k else
        repeat
          a[k1] : = ' '; k1 : = k1 − 1
        until k1 = k;
        pack (a, 1, id);
        vyhľadaj (koreň)
      end else
      begin {kontrola úvodzoviek a poznámky}
        if f↑ = '''' then
          repeat write (f↑);
                 get (f)
          until f↑ = '''' else
        if f↑ = '{' then
          repeat write (f↑);
                 get (f)
```



```
          until f↑ = '}';
          write (f↑);
          get (f)
        end
      end ;
      writeln;
      get (f)
    end ;
    page (output);
    vytlačstrom (koreň);
end.
```

Tab. 4.4 obsahuje výsledky spracovania krátkeho programového textu programom 4.5.

Ukážka výstupu programu 4.5 *Tabuľka 4.4*

ARRAY	4							
A	4	8	18	18	18	18	20	20
	20	20	26					
BEGIN	8	14	16	18	25			
CONST	2							
DO	8	17	26					
ELSE	15							
END	10	21	22	23	28			
FOR	8	17	26					
IF	15							
INTEGER	3	4	7	12	13			
I	3	7	8	8	13	17	18	18
	20	20	26	26	26			
K	12	15	16	17	18	18	19	20
	20							
N	2	4	8	26	27			
OF	4							
OUTPUT	1							
PERMUTE	1							
PERM	12	16	19	27				
PRINT	6	15						
PROCEDURE	6	12						
PROGRAM	1							

THEN	15				
TO	8	17	26		
VAR	3	7	13		
WRITELN	9				
WRITE	8				
X	13	18	18	20	20

4.4.4 RUŠENIE VRCHOLOV V STROME

Venujme sa teraz problému, ktorý je inverzný k pridávaniu vrcholov (prvkov), t.j. k ich rušeniu. Našou úlohou bude definovať algoritmus rušenia vrcholov v strome, t.j. odstránenia vrcholu s kľúčom x v strome, ktorý má usporiadané kľúče. Žiaľ, proces rušenia prvkov nie je až taký jednoduchý ako pridávanie. Je priamočiary, ak prvok, ktorý chceme zrušiť, predstavuje buď koncový vrchol stromu (t.j. list), alebo má iba jediného nasledovníka. Ťažkosti vzniknú pri rušení prvkov, ktoré majú dvoch nasledovníkov, pretože nemôžeme jedným smerníkom nahradiť dva smerníky. V takýchto situáciách sa zrušený prvok nahradí buď najpravejším prvkom ľavého podstromu, alebo najľavejším prvkom pravého podstromu. Obidva majú najviac jedného nasledovníka. Detaily sú uvedené v rekurzívnej procedúre zruš (4.52), ktorá rozlišuje tri možné prípady:

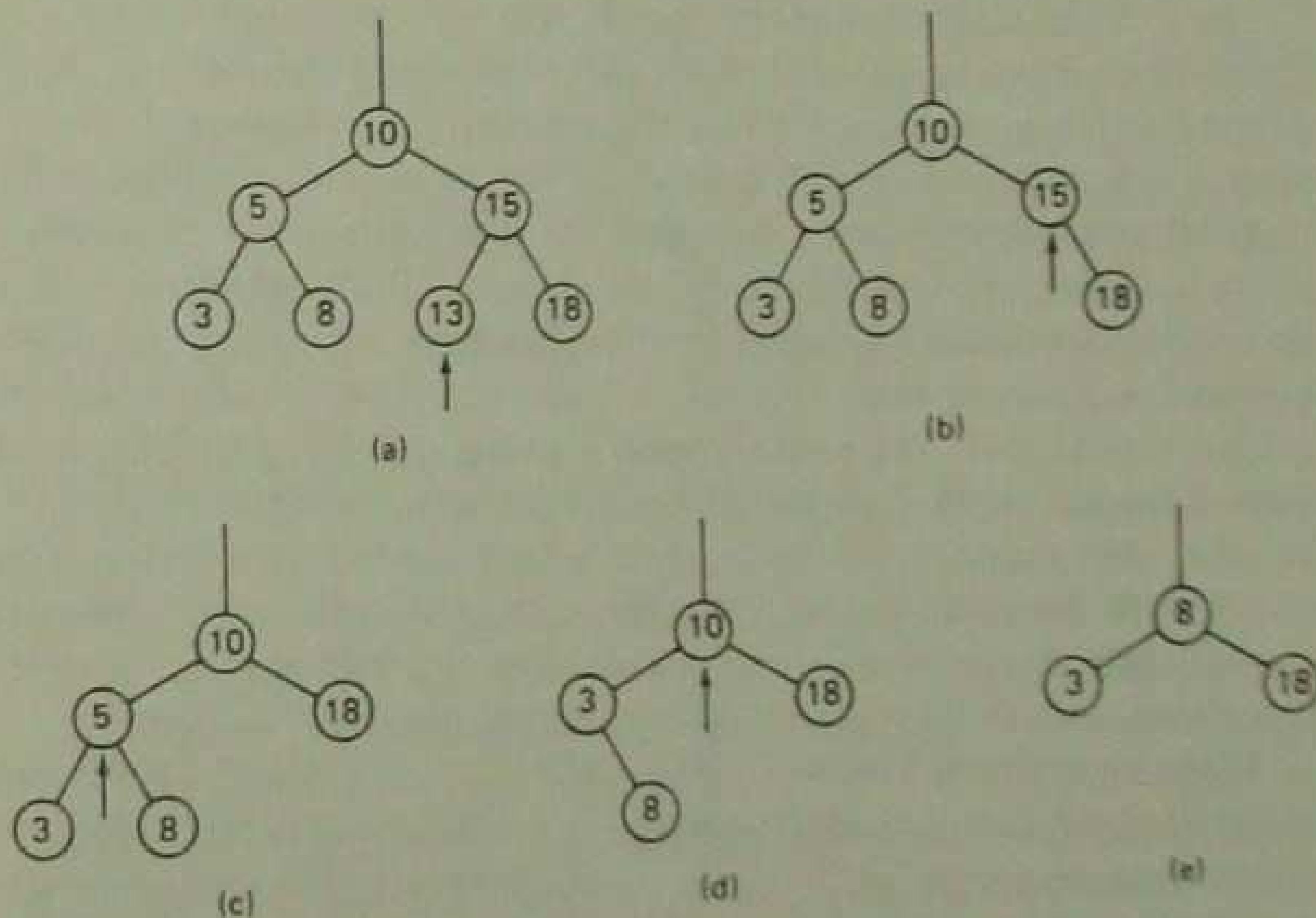
1. Prvok s kľúčom x sa v strome nenachádza.
2. Prvok s kľúčom x má najviac jedného nasledovníka.
3. Prvok s kľúčom x má dvoch nasledovníkov.

```
procedure zruš(x: integer; var p: ref);
  var q: ref;
  procedure del(var r: ref);
  begin if r↑.pravý ≠ nil then del(r↑.pravý) else
        begin q↑.kľúč := r↑.kľúč; q↑.počet := r↑.počet;
              q := r; r := r↑.ľavý
        end
  end;
begin {zruš}                                              (4.52)
  if p = nil then writeln('SLOVO NIE JE V STROME') else
  if x < p↑.kľúč then odober(x, p↑.ľavý) else
```



```
  if x > p↑.kľúč then odober(x, p↑.pravý) else
  begin {zruš p↑} q := p;
    if q↑.pravý = nil then p := q↑.ľavý else
    if q↑.ľavý = nil then p := q↑.pravý else del(q↑.ľavý);
    {dispose(q)}
  end
end {zruš}
```

Pomocná rekurzívna procedúra del sa vyvoláva iba v treťom prípade. Touto procedúrou zostupujme (prostredníctvom jej volaní) po najpravejšej vetve ľavého podstromu prvku $q\uparrow$, ktorý chceme odobrať. Pri návrate nahradí podstatné informácie (*kľúč* a *počet*) v prvku $q\uparrow$ zodpovedajúcimi hodnotami najpravejšieho prvku $r\uparrow$ v ľavom podstrome, na základe čoho možno prvok $r\uparrow$ odstrániť. Použitá nešpecifikovaná procedúra dispose(q) predstavuje inverznú operáciu k operácii new(q). Procedúra new(q), ako už vieme, pridelí potrebnú pamäť pre nový prvok. Procedúra dispose(q) potom oznámi počítačovému systému, že pamäť pridelená prvku $q\uparrow$ je už opäť k dispozícii na opätovné použitie.



Obr. 4.28. Rušenie vrcholov v strome

Aby sme si dokázali predstaviť činnosť procedúry (4.52), pozrime sa na *obr. 4.28*. Pôvodný strom je v ňom označený ako (*a*). Z tohto stromu postupne odoberieme vrcholy s kľúčmi 13, 15, 5, 10 a dostaneme výsledné stromy (pozri b) až e)).

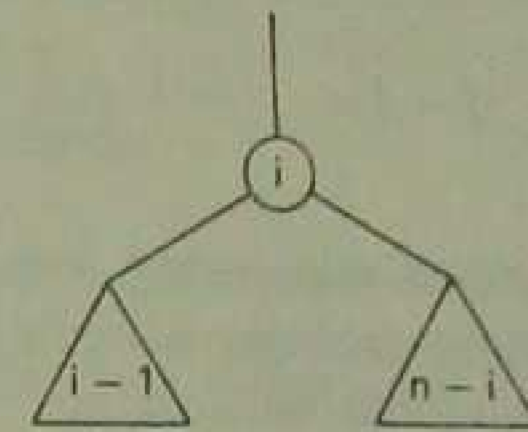
4.4.5 ANALÝZA VYHĽADÁVANIA A PRIDÁVANIA

Prirodzené a prospešné je, ak sa nám algoritmus vyhľadávania a pridávania v stromoch zdá podozrivý. Aspoň dovtedy je potrebné byť skeptickým, kým nezískame viac podrobnejších informácií o jeho správaní sa. Programátorom spôsobuje starosť predovšetkým skutočnosť, že vo všeobecnosti sa nevie, ako bude strom narastať; neexistuje v podstate žiadna predstava o jeho tvare. Jediné, čo možno predpokladať, je, že to pravdepodobne nebude dokonale vyvážený strom. Pretože priemerný počet porovnaní potrebných na lokalizáciu kľúča v dokonale vyváženom strome s n vrcholmi je približne $h = \log n$, bude počet porovnaní v strome, generovanom uvedeným algoritmom, väčší ako h. Ale o koľko väčší?

Nie je ťažké nájsť najhorší prípad. Predpokladajme, že všetky kľúče vstupného súboru sú už vzostupne (alebo zostupne) usporiadané. Každý kľúč je potom pripojený bezprostredne napravo (alebo naľavo) od svojho predchodcu a výsledný strom bude úplne degenerovaný, t.j. bude vyzerať ako lineárny zoznam. Vyhľadávanie bude v takomto prípade vyžadovať v priemere $n/2$ porovnaní. Tento najhorší prípad spôsobí zjavne malú výkonnosť vyhľadávacieho algoritmu, čím plne podporí našu nedôveru. Zostáva otázka, ako často sa bude takýto prípad vyskytovať. Presnejšie, chceme vedieť priemernú dĺžku cesty vyhľadávania a_n vzhľadom na všetkých n kľúčov a všetkých $n!$ stromov, ktoré možno vygenerovať z $n!$ permutácií pôvodných (rôznych) n kľúčov. Tento problém analýzy algoritmu sa zdá celkom priamočiary a uvádzame ho tu preto, že je jednak typickým príkladom analýzy algoritmu, jednak nám poskytuje významný praktický výsledok.

Majme n rôznych kľúčov s hodnotami 1, 2, ..., n. Predpokladajme, že kľúče nie sú usporiadané (vyskytujú sa v náhodnom poradí). Pravdepodobnosť, že prvý kľúč, ktorý sa, pochopiteľne, stane koreňom stromu, bude mať hodnotu i je $1/n$. Jeho ľavý podstrom bude obsahovať

$i - 1$ vrcholov, pravý podstrom $n - i$ vrcholov (*obr. 4.29*). Označme priemernú dĺžku cesty v ľavom podstrome symbolom a_{i-1}, v pravom podstrome symbolom a_{n-i}. Predpokladáme pritom, že všetky možné permutácie zvyšujúcich $n - 1$ kľúčov budú rovnako pravdepodobné.



Obr. 4.29. Rozdelenie váh v podstromoch

Priemerná dĺžka cesty v strome s n vrcholmi je daná súčtom súčinov úrovňového čísla každého vrcholu a pravdepodobnosti prístupu k jednotlivým vrcholom. Ak predpokladáme, že táto pravdepodobnosť bude rovnaká pre všetky vrcholy, tak platí

$$a_n = \frac{1}{n} \sum_{i=1}^{n} p_i \tag{4.53}$$

pričom p_i je dĺžka cesty pre i-tý vrchol stromu.

Vrcholy stromu, znázorneného na *obr. 4.29*, môžeme rozdeliť na tri triedy:

1. Priemerná dĺžka cesty pre $i - 1$ vrcholov v ľavom podstrome je $a_{i-1} + 1$.
2. Dĺžka cesty koreňa stromu je 1.
3. Priemerná dĺžka cesty pre $n - i$ vrcholov v pravom podstrome je $a_{n-i} + 1$.

Vzťah (4.53) potom môžeme vyjadriť ako sumu troch termov:

$$a_n^{(i)} = (a_{i-1} + 1)\frac{i-1}{n} + 1 \cdot \frac{1}{n} + (a_{n-i} + 1)\frac{n-i}{n} \tag{4.54}$$

Hľadanú priemernú dĺžku cesty a_n možno teraz odvodiť ako priemernú hodnotu $a_n^{(i)}$ pre všetky $i = 1, ..., n$, t.j. pre všetky stromy s kľúčmi 1, 2, ..., n v ich koreňoch.

$$a_n = \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\frac{i-1}{n}+\frac{1}{n}+(a_{n-i}+1)\frac{n-i}{n}\right] =$$

$$= 1 + \frac{1}{n^2}\sum_{i=1}^{n}[(i-1)a_{i-1}+(n-i)a_{n-i}] = \tag{4.55}$$

$$= 1 + \frac{2}{n^2}\sum_{i=1}^{n}(i-1)a_{i-1} = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i \, . \, a_i$$

Rovnica 4.55 predstavuje rekurentný vzťah pre a_n v tvare $a_n = f_1(a_1, a_2, \ldots, a_{n-1})$. Z neho môžeme odvodiť jednoduchší rekurentný vzťah $a_n = f_2(a_{n-1})$ nasledujúcim spôsobom;

Zo vzťahu (4.55) priamo vyplýva

1. $a_n = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i \, . \, a_i = 1 + \frac{2}{n^2}(n-1)a_{n-1} + \frac{2}{n^2}\sum_{i=1}^{n-2} i \, . \, a_i$

2. $a_{n-1} = 1 + \frac{2}{(n-1)^2}\sum_{i=1}^{n-2} i \, . \, a_i$

Vynásobením vzťahu 2 výrazom $((n-1)/2)^2$ dostávame

3. $\frac{2}{n^2}\sum_{i=1}^{n-2} i \, . \, a_i = \frac{(n-1)^2}{n^2}(a_{n-1} - 1)$

Ak dosadíme do rovnice 1 vzťah 3, dostávame

$$a_n = \frac{1}{n^2}((n^2-1)a_{n-1} + 2n - 1) \tag{4.56}$$

Zdá sa, že priemernú dĺžku cesty a_n možno vyjadriť v nerekurzívnom uzavretom tvare prostredníctvom harmonickej funkcie

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

$$a_n = 2\frac{n+1}{n} + H_n - 3 \tag{4.57}$$

(Čitateľ sa môže presvedčiť, že vzťah (4.57) vyhovuje rekurentnému vzťahu (4.56).)

Použitím Eulerovej formuly (a Eulerovej konštanty $\gamma \cong 0.577$)

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \ldots$$

dostávame pre veľké n vzťah

$$a_n \cong 2[\ln(n) + \gamma] - 3 = 2\ln(n) - c$$

Pretože priemerná dĺžka cesty v dokonale vyváženom strome je približne

$$a'_n = \log(n) - 1 \tag{4.58}$$

dostávame vzťah

$$\lim_{n\to\infty}\frac{a_n}{a'_n} = \frac{2\ln n}{\log n} = 2 \, . \ln 2 = 1.386 \tag{4.59}$$

Zanedbali sme v ňom konštantné výrazy, ktoré sú pre veľké n bezvýznamné.

Čo je pre nás podstatné z výsledku (4.59) tejto analýzy? Predovšetkým zistenie, že ak sa budeme snažiť za každú cenu skonštruovať dokonale vyvážený strom, namiesto „náhodného" generovaného programom 4.4, môžeme za predpokladu rovnakej pravdepodobnosti vyhľadania všetkých kľúčov očakávať priemerné zlepšenie dĺžky vyhľadávanej cesty najviac o 39 %. Zdôrazňujeme slovo priemerný, pretože zlepšenie môže byť, samozrejme, oveľa väčšie, a to konkrétne v prípade zdegenerovania vytváraného stromu na zoznam. Tento prípad sa však vyskytuje pomerne zriedkavo (ak sú všetky permutácie n pridávaných kľúčov rovnako pravdepodobné). V tejto súvislosti je pozoruhodné, že očakávaná priemerná dĺžka cesty v náhodnom strome rastie tiež logaritmicky s počtom jeho vrcholov, hoci v najhoršom prípade rastie dĺžka cesty lineárne.

Hodnota 39 % znamená hranicu prídavného úsilia, ktoré možno s výhodou vynaložiť na akýkoľvek druh reorganizácie stromovej štruktúry po pridaní kľúčov.

Prirodzene, naše rozhodnutie — investovať alebo neinvestovať do takejto činnosti — býva dosť ovplyvnené pomerom r medzi frekvenciami prístupu (výberu) do vrcholov (informácie) a pridávaním. Čím väčší je tento koeficient, tým viac sa oplatí investovať do reorganizačnej procedúry. Číslo 39 % je však pomerne nízke na to, aby sa vylepšenie priameho pridávania do stromov vo väčšine aplikácií vôbec vyplatilo, pokiaľ však nie je pomer počtu vrcholov a prístupov do nich, vzhľadom na pridávanie, dostatočne veľký (alebo ak sa obávame najhoršieho prípadu).

4.4.6 VYVÁŽENÉ STROMY

Z predchádzajúcej diskusie je jasné, že procedúra pridávania prvkov do stromu, ktorá vždy spôsobuje reštrukturalizáciu stromovej štruktúry za účelom dokonalého vyváženia, sa ťažko môže stať efektívnejšou, pretože obnova dokonalého vyváženia po náhodnom pridaní je dosť zložitá operácia. Možné zlepšenia spočívajú vo formulovaní menej prísnych definícií vyváženosti. Takéto nedokonalé kritériá vyváženosti by mali viesť k jednoduchším procedúram stromovej reorganizácie za cenu iba minimálneho zhoršenia priemernej výkonnosti vyhľadávania.

Jednu definíciu takejto vyváženosti sformulovali ADELSON-VELSKII a LANDIS [4-1]. Kritérium vyváženosti znie takto:

Strom je vyvážený vtedy a len vtedy, ak sa výšky dvoch podstromov každého vrcholu líšia najviac o 1.

Stromy, ktoré spĺňajú toto kritérium, sa často nazývajú *AVL-stromy* (podľa svojich vynálezcov). My ich budeme jednoducho nazývať *vyvážené stromy*, pretože kritérium vyváženosti je pre ne najcharakteristickejšie. (Poznamenávame, že všetky dokonale vyvážené stromy sú takisto AVL-vyváženými.)

Definícia je nielen jednoduchá, ale vedie aj k procedúre znovuvyváženia a k priemernej dĺžke cesty vyhľadávania, ktorá je prakticky identická s dĺžkou cesty dokonale vyváženého stromu.

Na vyvážených stromoch možno v čase $O(\log n)$ vykonávať nasledujúce operácie:

1. Vyhľadanie vrcholu s daným kľúčom.
2. Pridanie vrcholu s daným kľúčom.
3. Zrušenie vrcholu s daným kľúčom.

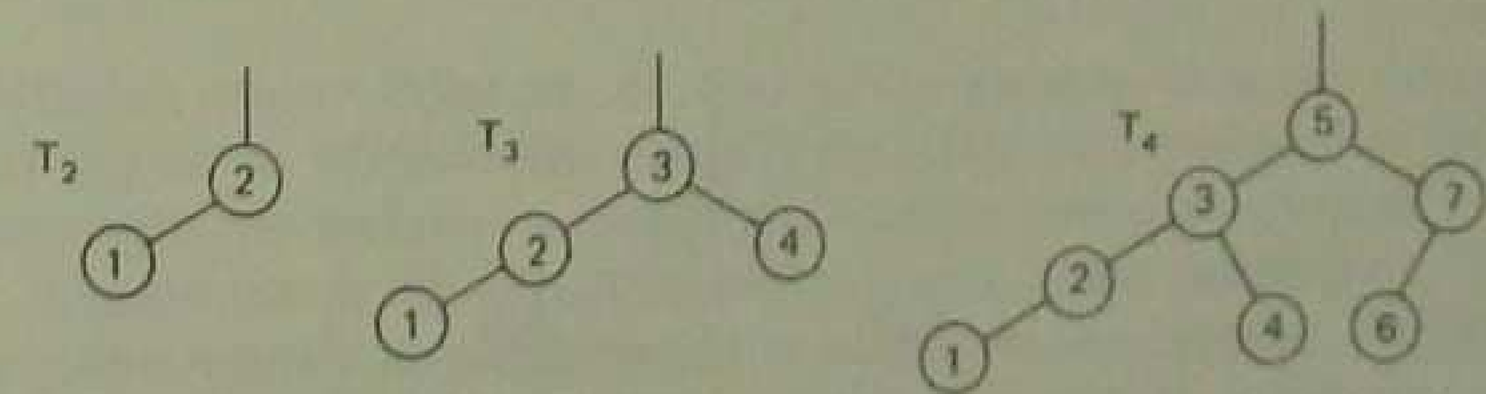
Tieto tvrdenia sú priame dôsledky vety, ktorú dokázali Adelson-Velskii a Landis a ktorá zaručuje, že vyvážený strom bude maximálne o 45 % (a nikdy nie viac) vyšší než jeho dokonale vyvážený „dvojník" bez ohľadu na počet vrcholov. Ak symbolom $h_b(n)$ označíme výšku vyváženého stromu s n vrcholmi, tak

$$\log(n+1) \leqq h_b(n) \leqq 1{,}4404 \,.\, \log(n+2) - 0{,}328 \qquad (4.60)$$

Optimálnu hodnotu získame, pochopiteľne, v prípade dokonale vyváženého stromu, keď $n = 2^k - 1$. Ale ako vyzerá štruktúra najhoršieho AVL-stromu?

Aby sme našli maximálnu výšku h pre všetky vyvážené stromy s n vrcholmi, uvažujme o pevnej výške h a pokúsme sa zostrojiť vyvážený strom s minimálnym počtom vrcholov. Je to odporúčaný spôsob, pretože podobne ako v prípade minimálnej výšky h aj v tomto prípade sa dá hodnota zistiť iba pre určité špecifické hodnoty n. Označme tento strom s výškou h symbolom T_h. Prirodzene, T_0 je potom prázdny strom, T_1 je strom s jediným vrcholom. Aby sme mohli zostrojiť strom T_h pre $h > 1$, pripojíme ku koreňu dva podstromy opäť s minimálnym počtom vrcholov. Teda tieto podstromy budú tiež T-stromy. Je jasné, že jeden podstrom musí mať výšku $h - 1$, pričom ten druhý ju môže mať o 1 menšiu, t. j. $h - 2$. Na *obr. 4.30* sú znázornené stromy s výškami 2, 3 a 4.

Pretože princíp kompozície týchto stromov sa veľmi podobá princípu definície Fibonacciho čísel, nazývame ich *Fibonacciho stromami.*



Obr. 4.30. Fibonacciho stromy výšky 2, 3 a 4

Definované sú takto:

1. Prázdny strom je Fibonacciho strom s výškou 0.
2. Jediný vrchol je Fibonacciho strom s výškou 1.
3. Ak T_{h-1} a T_{h-2} sú Fibonacciho stromy s výškou $h-1$ a $h-2$, tak $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$ je Fibonacciho strom s výškou h.
4. Žiadne iné stromy nie sú Fibonacciho stromy.

Počet vrcholov stromu T_h môžeme definovať pomocou jednoduchého rekurentného vzťahu:

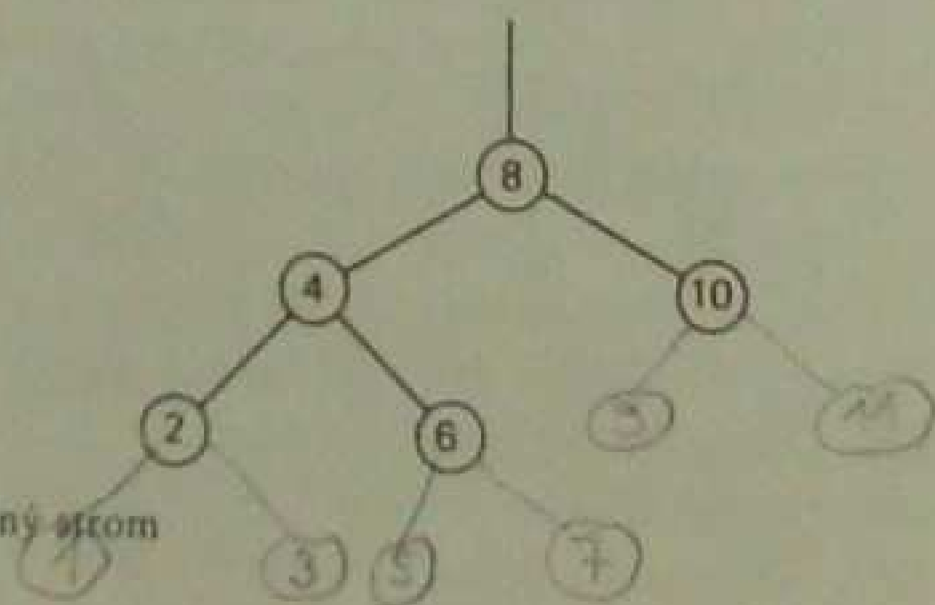
$$N_0 = 0, \quad N_1 = 1$$
$$N_h = N_{h-1} + 1 + N_{h-2} \tag{4.61}$$

Symbolmi N_i sú označené tie počty vrcholov, pre ktoré očakávame najhoršie prípady (t.j. horné limity) vzhľadom na vzťah (4.60).

4.4.7 PRIDÁVANIE DO VYVÁŽENÝCH STROMOV

Uvažujme teraz o tom, čo sa môže stať, ak pridáme nový vrchol do vyváženého stromu. Pre strom s koreňom r a dvoma podstromami, ľavým L a pravým R, môžu nastať tri prípady. Predpokladajme, že sa nový vrchol pridá do ľavého podstromu L, čím spôsobí zvýšenie jeho výšky o 1.

1. $h_L = h_R$: L a R budú mať rozdielne výšky, ale kritérium vyváženosti nie je porušené.
2. $h_L < h_R$: L a R budú mať rovnakú výšku, t.j. vyváženosť sa dokonca ešte zlepší.
3. $h_L > h_R$: kritérium vyváženosti sa poruší, strom treba rekonštruovať.

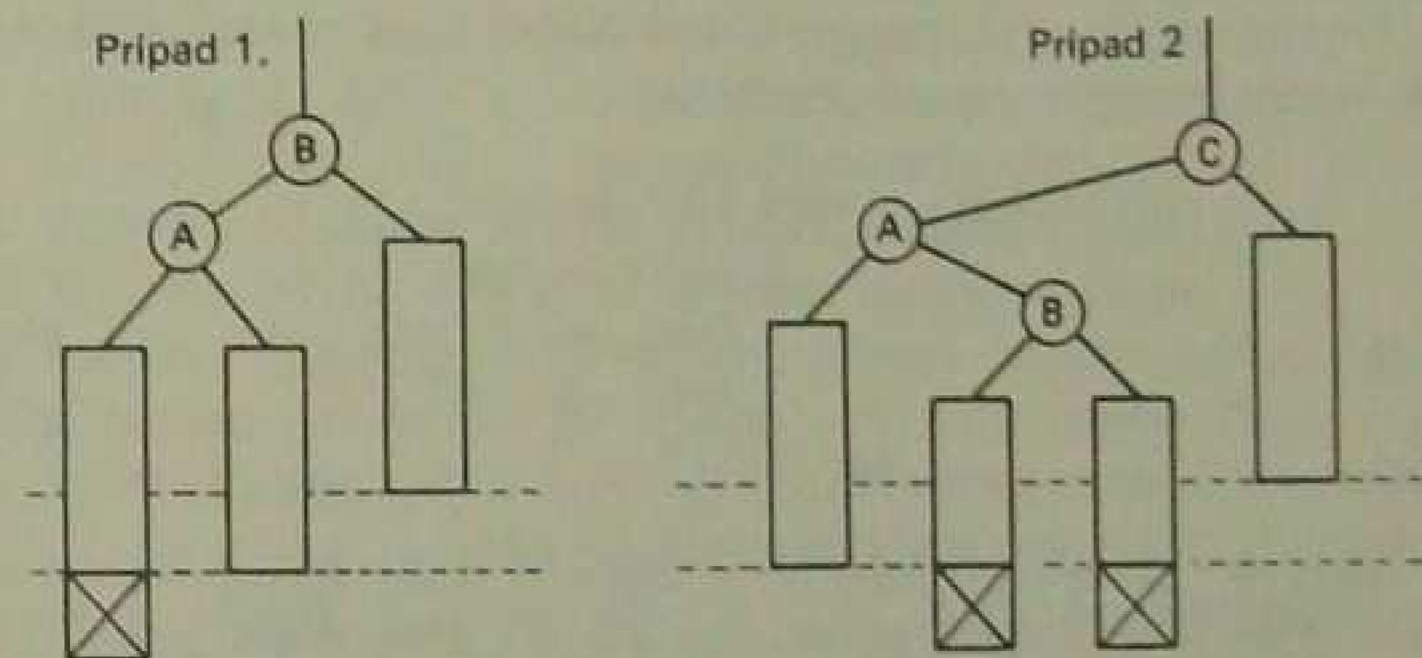


Obr. 4.31. Vyvážený strom

Uvažujme o strome na *obr. 4.31*. Vrcholy s kľúčmi 9 a 11 možno pridať bez nevyhnutnosti znovuvyváženia stromu; strom s koreňom 10 sa stane jednostranným (prípad 1); pri strome s koreňom 8 dôjde k zlepšeniu vyváženosti (prípad 2). Pridanie vrcholov s kľúčmi 1, 3, 5 alebo 7 znamená nevyhnutnosť dodatočného znovuvyváženia stromu.

Dôkladným preskúmaním situácie zistíme, že existujú v podstate iba dva rozdielne prípady vyžadujúce individuálne riešenie. Tretí prípad môžeme odvodiť na základe úvah o symetrii z predchádzajúcich dvoch prípadov. Prípad 1 charakterizuje pridanie vrcholov s kľúčmi 1 a 3, prípad 2 pridanie vrcholov 5 alebo 7 do stromu na *obr. 4.31*.

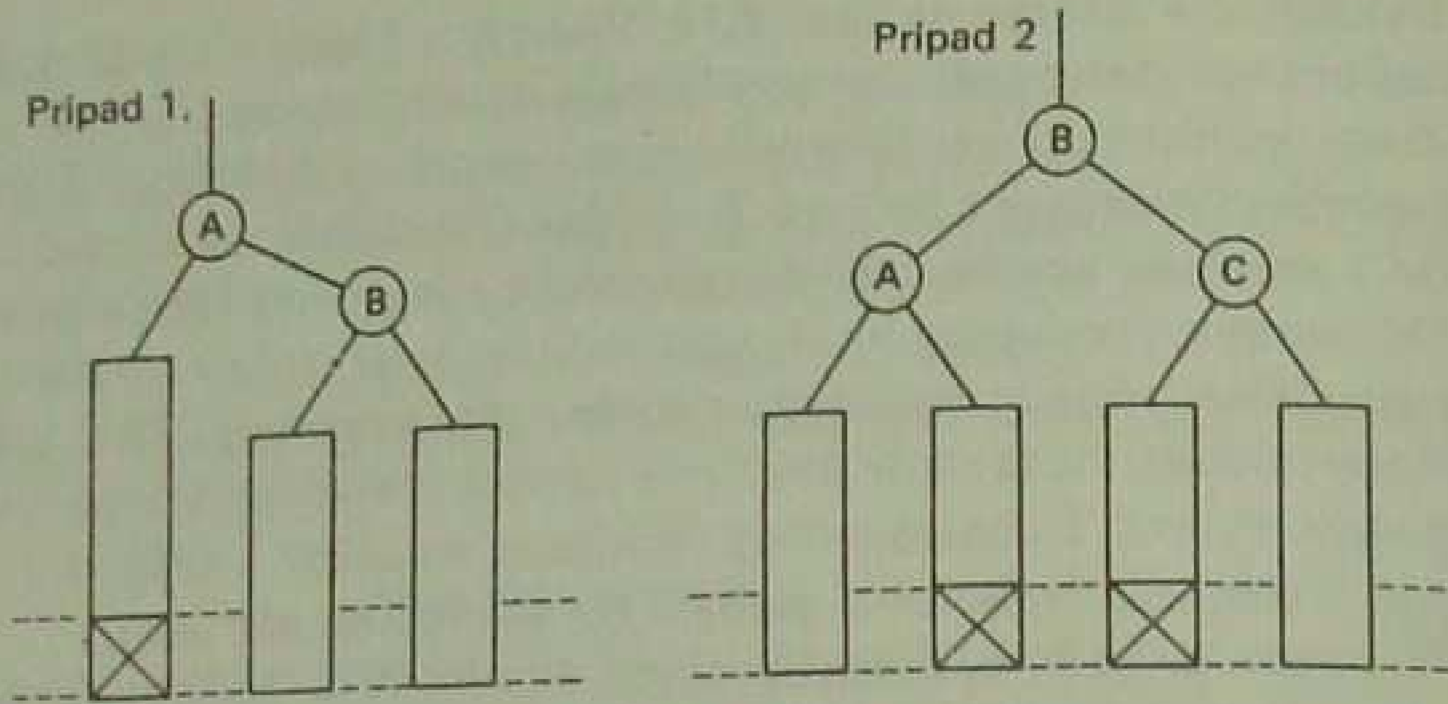
Tieto dva prípady sú zovšeobecnené na *obr. 4.32*. V ňom sú obdĺžnikmi označené jednotlivé podstromy a krížikom výška, o ktorú sa príslušný podstrom pridaním nového vrcholu zväčšil. Jednoduchými transformáciami oboch štruktúr obnovíme požadovanú vyváženosť.



Obr. 4.32. Nevyváženosť spôsobená pridaním nového vrcholu

Výsledok možno vidieť na *obr. 4.33*. Uvedomme si, že jediné dovolené pohyby sú vo vertikálnom smere, kým relatívne horizontálne pozície znázornených vrcholov a podstromov musia zostať nezmenené.

Algoritmus pridávania a znovuvyváženia závisí predovšetkým od spôsobu uchovávania informácie o stromovej vyváženosti. Extrémne riešenie spočíva v úplne implicitnom uchovávaní tejto informácie v samotnej štruktúre. V takomto prípade však treba po každom pridaní nového vrcholu znovu zisťovať príslušný vyvažovací faktor vrcholu, čo



Obr. 4.33. Obnovenie vyváženosti

nie je najlacnejšia operácia. Opačným extrémom je udržiavanie explicitného vyvažovacieho faktora pre každý vrchol stromu. Definíciu (4.48) typu vrchol potom možno rozšíriť na:

$$\textbf{type } \text{vrchol} = \textbf{record } \mathit{kľúč}\text{: integer; } \mathit{počet}\text{: integer; } \mathit{ľavý}, \mathit{pravý}\text{: ref; } \mathit{bal}: -1 \,..\, +1 \textbf{ end} \tag{4.62}$$

Vyvažovací faktor príslušného vrcholu budeme interpretovať ako rozdiel výšky jeho pravého podstromu a výšky jeho ľavého podstromu. Výsledný algoritmus bude založený na definícii (4.62) typu vrchol.

Proces pridania vrcholu sa v podstate skladá z týchto troch po sebe nasledujúcich bodov:

1. Prehľadanie stromu za účelom zistenia, či sa daný vrchol už v strome nenachádza.

2. Pridanie nového vrcholu a určenie výsledného vyvažovacieho faktora.

3. Skontrolovanie vyvažovacieho faktora každého vrcholu v opačnom smere cesty vyhľadávania (t. j. od vrcholu ku koreňu).

Aj keď táto metóda spôsobuje niektoré nadbytočné kontroly (keď sa raz dosiahne vyváženosť vrcholu, nie je potrebné ju overovať na jeho

predchodcoch), budeme sa spočiatku pridržiavať tejto evidentne správnej schémy, pretože ju možno implementovať jednoduchým rozšírením procedúry vyhľadávania a pridávania v programe 4.4. Táto procedúra opisuje operáciu vyhľadávania pre každý jeden vrchol a vzhľadom na jej rekurzívnu formuláciu ju možno jednoducho prispôsobiť tak, aby obsahovala prídavnú operáciu „na spiatočnej ceste vyhľadávacej trasy". V každom kroku je potrebné vrátiť informáciu o tom, či sa výška podstromu (v ktorom sa pridanie uskutočnilo) zväčšila alebo nie. Preto rozšírime zoznam parametrov procedúry o boolovský parameter h, ktorý bude oznamovať zväčšenie (resp. nezmenenie) výšky podstromu. Pochopiteľne, parameter h sa bude posielať referenciou, pretože prostredníctvom neho sa vracia výsledok.

Predpokladajme teraz, že proces pridania vrcholu sa vracia do vrcholu $p\uparrow$ z jeho ľavej vetvy (*obr. 4.32*) s informáciou, že výška podstromu sa zväčšila. Sme nútení rozlišovať tri situácie, týkajúce sa výšok podstromov pred pridaním nového vrcholu:

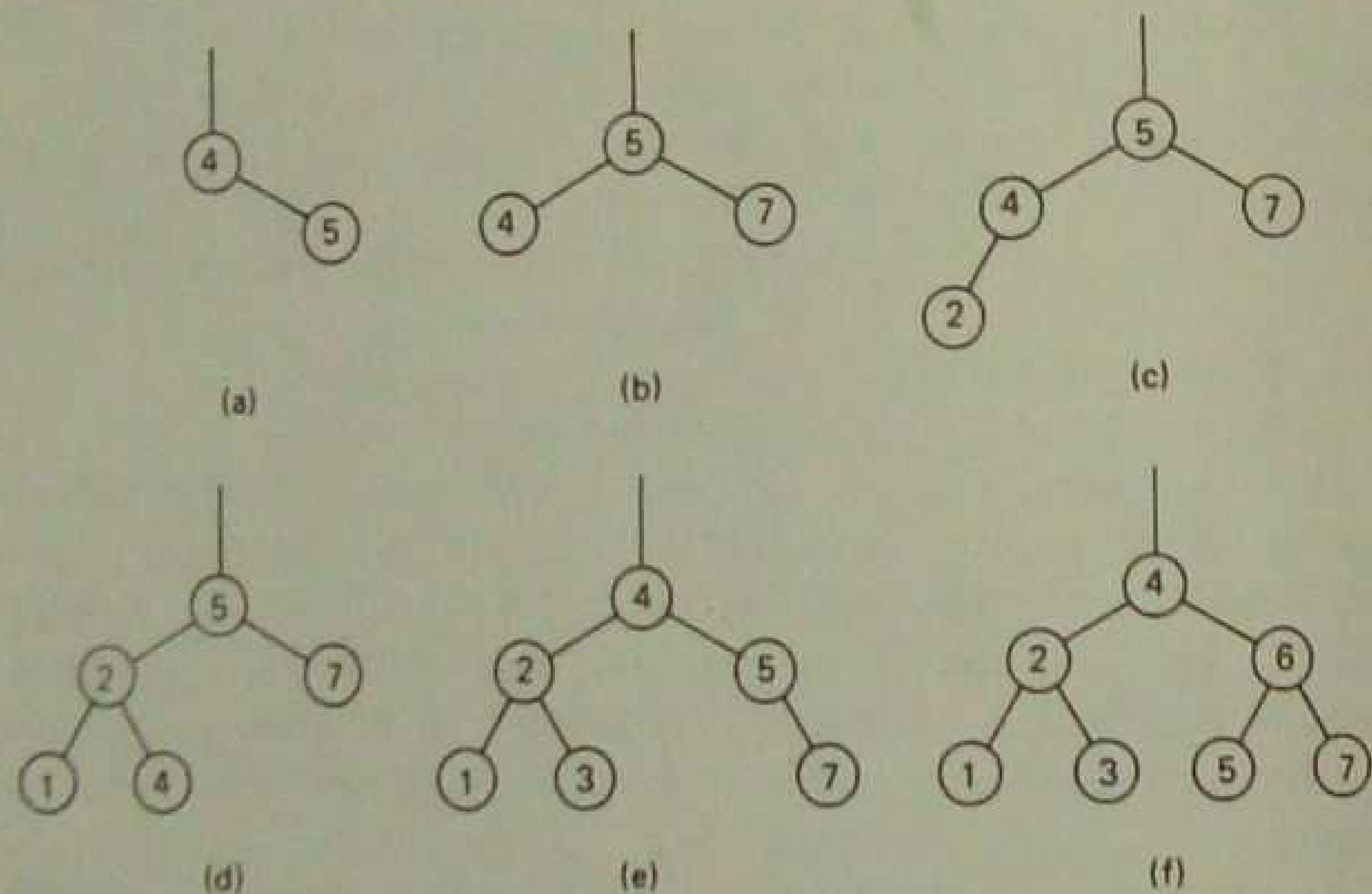
1. $h_L < h_R$, $p\uparrow.bal = +1$, predchádzajúca nevyváženosť vo vrchole p sa vyrovnala.
2. $h_L = h_R$, $p\uparrow.bal = 0$, váha sa teraz nakloní doľava.
3. $h_L > h_R$, $p\uparrow.bal = -1$, je potrebné znovuvyváženie.

V treťom prípade na základe preskúmania vyvažovacieho faktora koreňa ľavého podstromu (povedzme $p1\uparrow.bal$) zistíme, či ide o prípad 1 alebo 2 z *obr. 4.32*. Ak je výška ľavého podstromu tohto vrcholu väčšia, ako výška pravého podstromu, ide o prípad 1, v opačnom prípade o prípad 2. (Presvedčte sa sami, že v takomto prípade sa nemôže vyskytnúť ľavý podstrom s vyvažovacím faktorom 0 v jeho koreni.)

Potrebné operácie znovuvyváženia sú formulované ako postupnosti rôznych zámen smerníkov. Skutočne, smerníky sa cyklicky zamieňajú, v dôsledku čoho môže dôjsť buď k jednoduchej, alebo dvojitej rotácii dvoch alebo troch vrcholov. Okrem rotácie smerníkov treba primerane ponastavovať vyvažovacie faktory jednotlivých vrcholov. Podrobnosti si čitateľ môže všimnúť v procedúre vyhľadávania, pridávania a znovuvyvažovania, uvedených v schéme (4.63).

Princíp činnosti je na *obr. 4.34*. Uvažujme o binárnom strome (*a*), ktorý sa skladá iba z dvoch vrcholov. Pridanie kľúča 7 spôsobí nevyvá-

ženosť stromu (vznikne lineárny zoznam). Vyváženie takéhoto stromu vyžaduje jednoduchú *RR*-rotáciu, čím dostaneme dokonale vyvážený strom (*b*). Ďalším pridaním vrcholov 2 a 1 nastane nevyváženosť podstromu s koreňom 4. Na jeho vyváženie potrebujeme použiť jednoduchú *LL*-rotáciu (*d*). Nasledujúcim pridaním kľúča 3 sa naruší kritérium vyváženosti koreňa 5. Znovuvyváženie dosiahneme o niečo zložitejšou dvojitou *LR*-rotáciou; výsledkom je strom (*e*). Jediným kandidátom na stratu vyváženosti po ďalšom pridaní je vrchol 5. Skutočne, pridanie vrcholu 6 musí spôsobiť štvrtý prípad znovuvyváženia, zachyteného v algoritme (4.63) vo forme dvojitej *RL*-rotácie. Výsledným stromom je strom (*f*) na *obr. 4.34*.



Obr. 4.34. Pridávanie do vyváženého stromu

V súvislosti s výkonnosťou algoritmu pridávania do vyváženého stromu nás zaujímajú predovšetkým tieto dve otázky:

1. Aká bude očakávaná výška vytvoreného vyváženého stromu, ak sa všetkých $n!$ permutácií n kľúčov vyskytne s rovnakou pravdepodobnosťou?

2. Aká je pravdepodobnosť, že pridanie bude vyžadovať znovuvyváženie?

Matematická analýza tohto zložitého algoritmu je stále ešte otvorený problém. Empirické testy potvrdzujú domnienku, že očakávaná výška vyváženého stromu generovaného algoritmom (4.63) je $h = \log(n) + c$, pričom c je konštanta s malou hodnotou ($c \cong 0{,}25$). To znamená, že v praxi sa vyvážené AVL-stromy správajú rovnako dobre ako dokonale vyvážené stromy, navyše sa s nimi jednoduchšie manipuluje. Empirické pokusy nám tiež hovoria, že v priemere je potrebné jedno znovuvyváženie na približne každé dve pridania nových vrcholov. V tomto prípade sú jednoduché a dvojité rotácie rovnako pravdepodobné. Príklad na *obr. 4.34* sme starostlivo vybrali, aby sme mohli ukázať všetky možné rotácie, ktoré sa môžu vyskytnúť v rámci minimálneho počtu pridaní vrcholov.

```
procedure vyhľadaj (x: integer; var p: ref; var h: boolean);
  var p1, p2: ref;       {h = false}
begin
  if p = nil then
  begin {slovo nie je v strome; treba ho pridať do stromu}
    new(p); h: = true;
    with p↑ do
    begin
          kľúč: = x; počet: = 1;
          ľavý: = nil; pravý: = nil;
          bal: = 0
    end
  end else
  if x < p↑.kľúč then
  begin vyhľadaj (x, p↑.ľavý, h);
    if h then      {ľavá vetva sa zväčšila}
    case p↑.bal of                                          (4.63)
      1: begin p↑.bal: = 0; h: = false
         end;
      0: p↑.bal: = −1;
     −1: begin {znovuvyváženie}
          p1: = p↑.ľavý;
          if p1↑.bal = −1 then
```

```
            begin {jednoduchá LL-rotácia}
              p↑.ľavý: = p1↑.pravý; p1↑.pravý: = p;
              p↑.bal: = 0; p: = p1
            end else
            begin {dvojitá LR-rotácia}
              p2: = p1↑.pravý;
              p1↑.pravý: = p2↑.ľavý;
              p2↑.ľavý: = p1;
              p1↑.ľavý: = p2↑.pravý;
              p2↑.pravý: = p;
              if p2↑.bal = -1 then p↑.bal: = +1 else p↑.bal: = 0;
              if p2↑.bal = +1 then p1↑.bal: = -1 else p1↑.bal: = 0;
              p: = p2
            end;
            p↑.bal: = 0; h: = false
          end
        end
      end else
      if x > p↑.kľúč then
      begin vyhľadaj(x, p↑.pravý, h);
        if h then      {pravá vetva sa zväčšila}
        case p↑.bal of
        -1: begin p↑.bal: = 0; h: = false
            end;
         0: p↑.bal: = +1;
         1: begin {znovuvyváženie}
              p1: = p↑.pravý;
              if p1↑.bal = +1 then
              begin {jednoduchá RR-rotácia}
                p↑.pravý: = p1↑.ľavý;
                p1↑.ľavý: = p;
                p↑.bal: = 0;
                p: = p1
              end else
              begin {dvojitá RL-rotácia}
                p2: = p1↑.ľavý;
                p1↑.ľavý: = p2↑.pravý; p2↑.pravý: = p1;
                p↑.pravý: = p2↑.ľavý; p2↑.ľavý: = p;
                if p2↑.bal = +1 then p↑.bal: = -1 else p↑.bal: = 0;
                if p2↑.bal = -1 then p1↑.bal: = +1 else p1↑.bal: = 0;
                p: = p2
              end; p↑.bal: = 0; h: = false
            end
          end
        end
      else
      begin p↑.počet: = p↑.počet + 1;
            h: = false
      end
end {vyhľadaj}
```

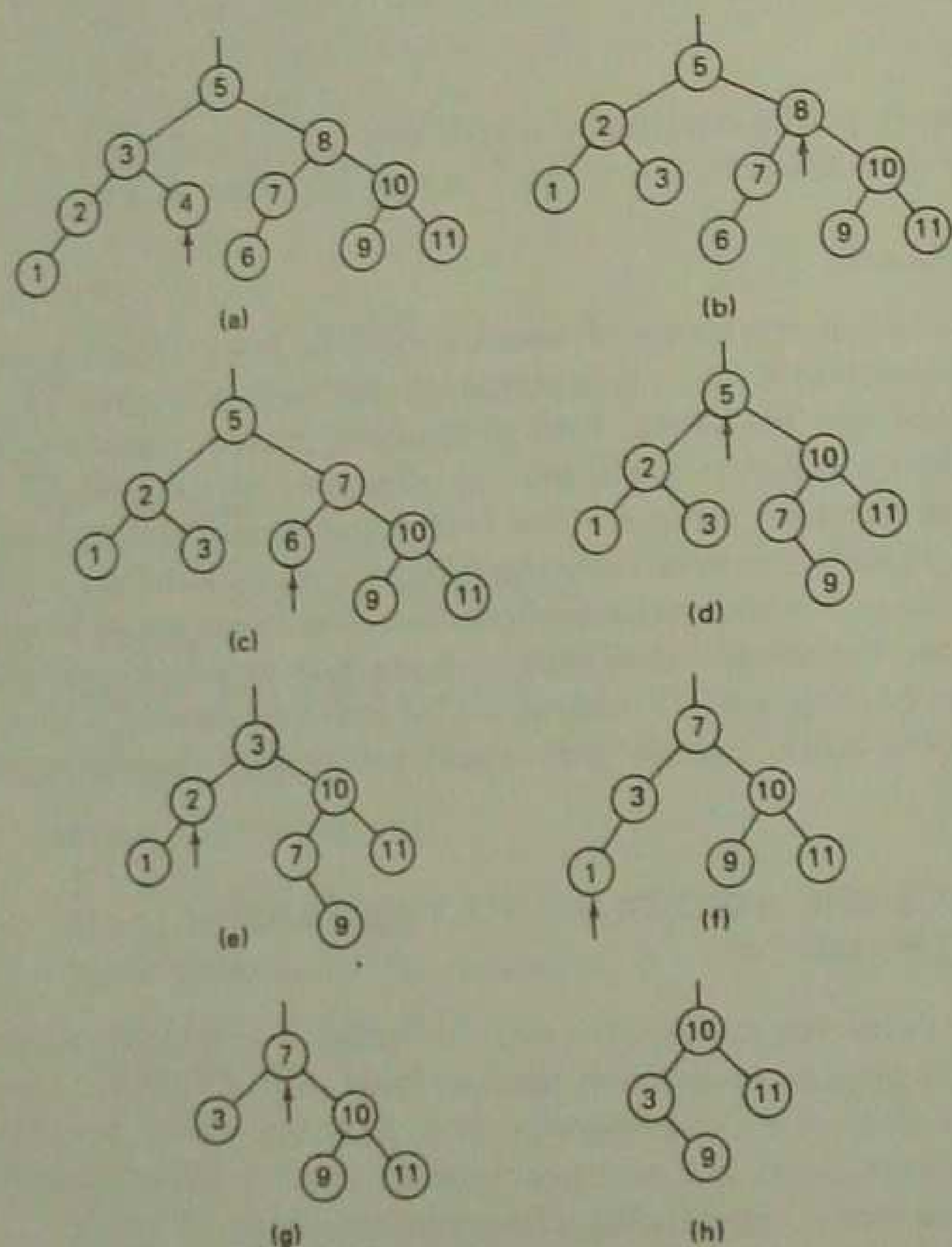

Zo zložitosti vyvažovacích operácií vyplýva, že vyvážené stromy by sa mali používať hlavne v tých prípadoch, keď bude zjavne viac výberov informácií ako pridávaní. Toto je skutočne pravda, pretože vrcholy takýchto vyhľadávacích stromov sú obyčajne implementované ako zhustené záznamy, predovšetkým kvôli šetreniu pamäti. Rýchlosť prístupu a aktualizácie vyvažovacieho faktora, vyžadujúceho iba dva bity, je preto často rozhodujúcim koeficientom efektívnosti operácie znovuvyváženia. Empirické vyhodnotenia ukazujú, že vyvážené stromy strácajú veľa zo svojej príťažlivosti, ak je silné zhustenie záznamov povinné. Je skutočne ťažké „poraziť" jednoduchý priamočiary algoritmus pridávania!

4.4.8 RUŠENIE VRCHOLOV VO VYVÁŽENÝCH STROMOCH

Naše doterajšie skúsenosti s rušením vrcholov zo stromov naznačujú, že aj v prípade vyvážených stromov bude rušenie vrcholov pravdepodobne zložitejšie ako pridávanie. Je to skutočne pravda, hoci operácia znovuvyváženia je v podstate rovnaká ako v prípade pridávania; pozostáva buď z jednoduchej, alebo dvojitej rotácie vrcholov.

Základnou schémou na realizáciu procedúry rušenia vrcholov z vy-

váženého stromu je algoritmus (4.52). Jednoduché sú opäť prípady koncových vrcholov a vrcholov, ktoré majú jediného nasledovníka. Ak má vrchol, ktorý chceme zrušiť, dva podstromy, tak ho opäť nahradíme najpravejším vrcholom jeho ľavého podstromu. Podobne ako v prípade pridávania (4.63) použijeme ďalší boolovský parameter h označujúci zmenšenie (alebo nezmenšenie) výšky podstromu. Iba v prípade, že h



Obr. 4.35. Rušenie vrcholov vo vyváženom strome

má hodnotu true, budeme uvažovať o znovuvyvážení. Parameter h nadobudne hodnotu true buď vtedy, keď nájdeme a zrušíme príslušný vrchol, alebo keď samotné znovuvyváženie spôsobí zmenšenie výšky podstromu. V algoritme (4.64) zavádzame dve (symetrické) vyvažovacie operácie v tvare procedúr, pretože sú volané z viacerých miest algoritmu rušenia vrcholov. Poznamenávame, že procedúra vyváž1 sa volá v prípade zmenšenia výšky ľavého podstromu, procedúra vyváž2 v opačnom prípade.

Operáciu rušenia vrcholov vo vyváženom strome znázorňuje *obr. 4.35*. Z pôvodného vyváženého stromu (*a*) sa postupne odoberajú vrcholy s kľúčmi 4, 8, 6, 5, 2, 1 a 7; výsledkom sú stromy (*b*), ..., (*h*).

Zrušenie vrcholu s kľúčom 4 je jednoduché, pretože tento vrchol je listom. Zrušením tohto vrcholu sa však naruší vyváženosť vrcholu 3. Operácia znovuvyváženia vrcholu vyžaduje jednoduchú *LL*-rotáciu.

Znovuvyváženie bude opäť potrebné pri zrušení vrcholu 6. V tomto prípade treba vyvážiť pravý podstrom koreňa 7, a to pomocou jednoduchej *RR*-rotácie. Zrušenie vrcholu 2 je síce opäť priamočiare, pretože tento vrchol má iba jedného nasledovníka, spôsobí však zložitú dvojitú *RL*-rotáciu. Predtým ako zrušíme vrchol 7, je potrebné nahradiť ho najpravejším vrcholom jeho ľavého podstromu, t.j. vrcholom s kľúčom 3. Nasledujúca dvojitá *LR*-rotácia spôsobí znovuvyváženie stromu a jeho záverečnú podobu (*h*).

```
procedure zruš (x: integer; var p: ref; var h: boolean);
  var q: ref;       {h = false}
  procedure vyváž1 (var p: ref; var h: boolean);
    var p1, p2: ref; b1, b2: -1 .. +1;
  begin {h = true, ľavá vetva sa zmenšila}
    case p↑.bal of
    -1: p↑.bal: = 0;
     0: begin p↑.bal: = +1; h: = false
        end;
     1: begin {znovuvyváženie} p1: = p↑.pravý; b1: = p1↑.bal
        if b1 ≧ 0 then
        begin {jednoduchá RR-rotácia}
          p↑.pravý: = p1↑.ľavý; p1↑.ľavý: = p;
```

```
        if b1 = 0 then
        begin p↑.bal: = +1; p1↑.bal: = −1; h: = false
        end else
        begin p↑.bal: = 0; p1↑.bal: = 0
        end;
        p: = p1
      end else
      begin {dvojitá RL-rotácia}
        p2: = p1↑.ľavý; b2: = p2↑.bal;
        p1.ľavý: = p2↑.pravý; p2↑.pravý: = p1;
        p.pravý: = p2↑.ľavý; p2↑.ľavý: = p;
        if b2 = +1 then p↑.bal: = −1 else p↑.bal: = 0;
        if b2 = −1 then p1↑.bal: = +1 else p1↑.bal: = 0;
        p: = p2; p2↑.bal: = 0
      end
    end
  end
end {vyvážl}                                                      (4.64)
procedure vyváž2 (var p: ref; var h: boolean);
  var p1, p2: ref; b1, b2: −1 .. +1;
begin {h = true, pravá vetva sa zmenšila}
  case p↑.bal of
    1: p↑.bal: = 0;
    0: begin p↑.bal: = −1; h: = false
       end;
   −1: begin {znovuvyváženie}
      p1: = p↑.ľavý; b1: = p1↑.bal;
      if b1 = 0 then
      begin {jednoduchá LL-rotácia}
        p↑.ľavý: = p1↑.pravý; p1↑.pravý: = p;
        if b1 = 0 then
        begin p↑.bal: = −1; p1↑.bal: = +1; h: = false
        end else
        begin p↑.bal: = 0; p1↑.bal: = 0
        end;
        p: = p1
```



```
      end else
      begin {dvojitá LR-rotácia}
        p2: = p1↑.pravý; b2: = p2↑.bal;
        p1↑.pravý: = p2↑.ľavý; p2↑.ľavý: = p1;
        p↑.ľavý: = p2↑.pravý; p2↑.pravý: = p;
        if b2 = −1 then p↑.bal: = +1 else p↑.bal: = 0;
        if b2 = +1 then p1↑.bal: = −1 else p1↑.bal = 0;
        p: = p2; p2↑.bal: = 0
      end
    end
  end
end {vyváž2};
procedure del (var r: ref; var h: boolean);
begin {h = false}
  if r↑.pravý ≠ nil then
  begin del (r↑.pravý, h); if h then vyváž2 (r, h)
  end else
  begin q↑.kľúč: = r↑.kľúč; q↑.počet: = r↑.počet;
        r: = r↑.ľavý; h: = true
  end
end;
begin {zruš}                                                      (4.64)
  if p = nil then
    begin writeln ('KĽÚČ NIE JE V STROME'); h: = false
    end else
  if x < p↑.kľúč then
    begin zruš (x, p↑.ľavý, h); if h then vyvážl (p, h)
    end else
  if x > p↑.kľúč then
    begin zruš (x, p↑.pravý, h); if h then vyváž2 (p, h)
    end else
  begin {zruš p↑} q: = p;
    if q↑.pravý = nil then
      begin p: = q↑.ľavý; h: = true
      end else
    if q↑.ľavý = nil then
```

```
      begin p: = q↑.pravý; h: = true
      end else
    begin del(q↑.ľavý, h); if h then vyvážl (p, h)
    end;
    {dispose(q)}
  end
end {zruš}
```

Je jasné, že zrušenie prvku vo vyváženom strome možno vykonať — v najhoršom prípade — prostredníctvom $O(\log n)$ operácií. Neslobodno však prehliadnuť podstatný rozdiel medzi správaním sa procedúry pridávania a procedúry rušenia. Zatiaľ čo pridanie jednoduchého kľúča môže spôsobiť nanajvýš jednu rotáciu (dvoch alebo troch vrcholov), zrušenie môže vyžadovať rotáciu všetkých vrcholov absolvovanej cesty. Uvažujme napr. o zrušení najpravejšieho vrcholu Fibonacciho stromu. V tomto prípade zrušenie ľubovoľného vrcholu spôsobí zmenšenie výšky stromu; navyše, zrušenie najpravejšieho vrcholu vyžaduje maximálny počet rotácií. Tento prípad predstavuje najhorší výber vrcholu v najhoršom prípade vyváženosti stromu, teda najnešťastnejšiu kombináciu možností!

S akou pravdepodobnosťou sa potom vyskytujú potreby rotácie vo všeobecnosti?

Prekvapujúcim výsledkom empirických testov je, že zatiaľ čo približne pre každé druhé pridanie treba jednu rotáciu, až každé piate zrušenie vyžaduje jednu rotáciu. Preto možno rušenie vrcholov vo vyvážených stromoch pokladať za také jednoduché (alebo zložité) ako pridávanie.

4.4.9 OPTIMÁLNE VYHĽADÁVACIE STROMY

Doteraz boli naše úvahy o organizácii vyhľadávacích stromov založené na predpoklade, že frekvencia prístupu bude rovnaká pre všetky vrcholy, t. j. všetky kľúče sa vyskytnú s rovnakou pravdepodobnosťou (ako parameter vyhľadávania). Je to zrejme najlepší predpoklad, pokiaľ nemáme predstavu o distribúcii prístupov. Vyskytujú sa však prípady (i keď sú skôr výnimkou ako pravidlom), keď sú známe informácie o pravdepodobnosti prístupu k jednotlivým kľúčom. Tieto prípady sú

charakteristické tým, že kľúče vo vrcholoch zostávajú nezmenené, t. j. na danom vyhľadávacom strome sa neaplikuje ani operácia pridávania, ani operácia rušenia vrcholov, teda strom si udržuje konštantnú štruktúru. Typickým príkladom je syntaktický analyzátor kompilátora, ktorý pre každé slovo (identifikátor) zistí, či je alebo nie je kľúčovým (rezervovaným) slovom. Pomocou štatistických vyhodnocovacích metód potom môžeme pre stovky kompilovaných programov presne určiť relatívne frekvencie výskytov jednotlivých identifikátorov, a tým aj frekvencie prístupov k jednotlivým kľúčom.

Predpokladajme, že symbolom p_i označujeme pravdepodobnosť prístupu k i-tému vrcholu vyhľadávacieho stromu.

$$P_r\{x = k_i\} = p_i, \quad \sum_{i=1}^{n} p_i = 1 \tag{4.65}$$

Našou snahou teraz bude organizovať vyhľadávací strom takým spôsobom, aby bol celkový počet vyhľadávacích krokov minimálny. Preto pozmeníme definíciu dĺžky cesty (4.34) tak, aby zohľadňovala i určitú váhu pridelenú každému vrcholu. Vrcholy, ktoré sú pomerne často navštevované, sa stanú ťažkými vrcholmi; na druhej strane vrcholy navštevované zriedkavo sa považujú za ľahké. Váhovaná (vnútorná) dĺžka cesty je potom daná súčtom všetkých ciest od koreňa do každého vrcholu, váhovaného pravdepodobnosťou prístupu k tomuto vrcholu.

$$P_I = \sum_{i=1}^{n} p_i h_i \tag{4.66}$$

Symbol h_i určuje úroveň i-tého vrcholu (alebo aj jeho vzdialenosť od koreňa $+1$). Našim cieľom bude minimalizácia váhovanej dĺžky cesty pre dané rozloženie pravdepodobnosti.

Ako príklad uvažujme o množine kľúčov 1, 2, 3 s pravdepodobnosťami prístupov $p_1 = 1/7$, $p_2 = 2/7$ a $p_3 = 4/7$. Tieto tri kľúče môžu byť usporiadané vo forme vyhľadávacích stromov piatimi rôznymi spôsobmi (*obr. 4.36*). Váhované dĺžky ciest budú potom pre jednotlivé stromy, na základe vzťahu (4.66), tieto:

$$P_I^{(a)} = \frac{1}{7}(1.3 + 2.2 + 4.1) = \frac{11}{7}$$

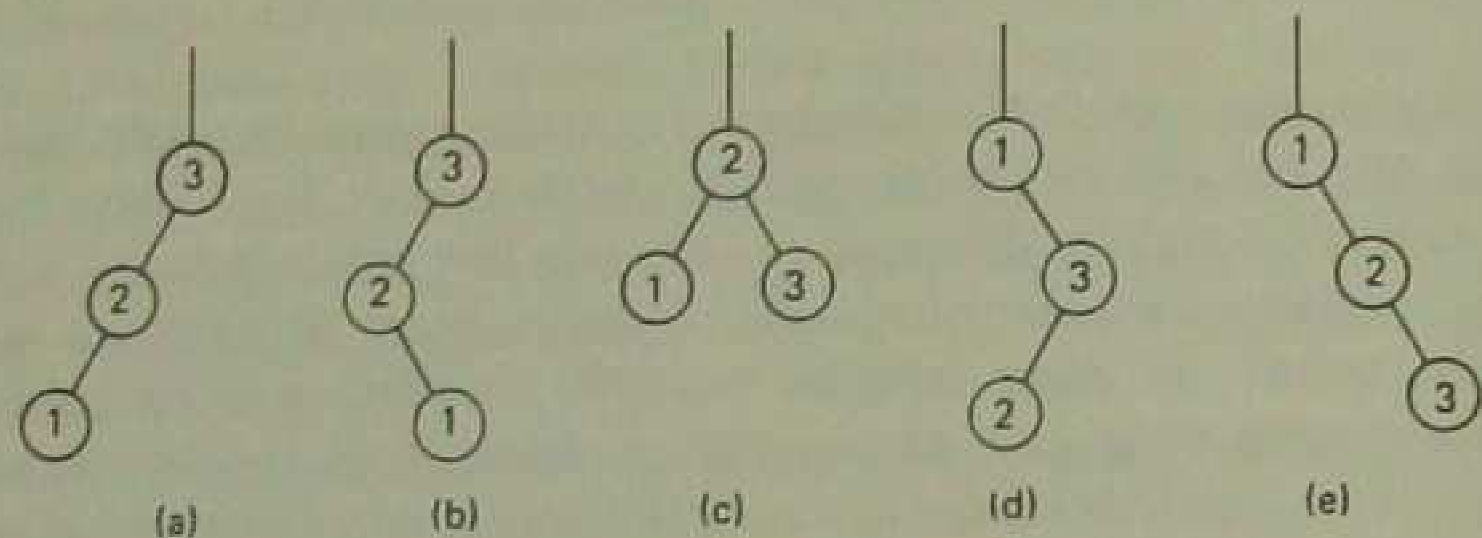
$$P_I^{(b)} = \frac{1}{7}(1.2 + 2.3 + 4.1) = \frac{12}{7}$$

$$P_I^{(c)} = \frac{1}{7}(1.2 + 2.1 + 4.2) = \frac{12}{7}$$

$$P_I^{(d)} = \frac{1}{7}(1.1 + 2.3 + 4.2) = \frac{15}{7}$$

$$P_I^{(e)} = \frac{1}{7}(1.1 + 2.2 + 4.3) = \frac{17}{7}$$

Vidíme, že v tomto príklade je optimálnym usporiadaním degenerovaný strom (*a*), a nie dokonale vyvážený strom (*c*).



Obr. 4.36. Vyhľadávacie stromy s tromi vrcholmi

Príklad syntaktického analyzátora kompilátora nás podnecuje k tomu, aby sme sa týmto problémom zaoberali v súvislosti s trochu všeobecnejšími podmienkami: slová, vyskytujúce sa v zdrojovom texte, nie sú vždy kľúčovými, skôr naopak, výskyt takýchto slov býva väčšinou výnimkou. Zistenie, že slovo k nie je vo vyhľadávacom strome kľúčom, môžeme chápať ako prístup k hypotetickému „špeciálnemu vrcholu", vloženému medzi najbližší nižší a najbližší vyšší kľúč (*obr. 4.19*) s patričnou dĺžkou vonkajšej cesty.

Ak poznáme pravdepodobnosť q_j výskytu vyhľadávaného argumentu x medzi dvoma kľúčmi k_j a k_{j+1}, tak na základe tejto informácie možno značne zmeniť štruktúru optimálneho vyhľadávacieho stromu. Teda zovšeobecníme problém tým, že budeme uvažovať aj o neúspešnom vyhľadávaní.

Celkovú priemernú váhovanú dĺžku cesty môžeme vyjadriť vzťahom

$$P = \sum_{i=1}^{n} p_i h_i + \sum_{j=0}^{m} q_j h'_j$$

pričom (4.67)

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{m} q_j = 1$$

Symbolom h_i označujeme úroveň i-tého (vnútorného) vrcholu symbolom h'_j úroveň j-tého vonkajšieho vrcholu. Priemernú váhovanú dĺžku cesty možno nazývať „cenou" vyhľadávacieho stromu, pretože reprezentuje mieru očakávaného množstva námahy vynaloženej na vyhľadávanie. Vyhľadávací strom, ktorého štruktúra prináša minimálnu cenu spomedzi všetkých stromov s danou množinou kľúčov k_i a pravdepodobnosťami p_i a q_j, sa nazýva *optimálny strom*. Na nájdenie optimálneho stromu nepotrebujeme požiadavku, aby sa súčet p a q rovnal jednotke. V skutočnosti sa tieto pravdepodobnosti bežne určujú na základe experimentov, pri ktorých sa počítajú prístupy k jednotlivým vrcholom. Teda namiesto pravdepodobností p_i a q_j budeme ďalej používať takéto počty frekvencií a označíme ich symbolmi a_i, b_j, pričom

— a_i určuje, koľkokrát sa vyhľadávaný argument x rovnal kľúču k_i,
— b_j určuje, koľkokrát sa vyhľadávaný argument x nachádzal medzi kľúčmi k_j a k_{j+1}.

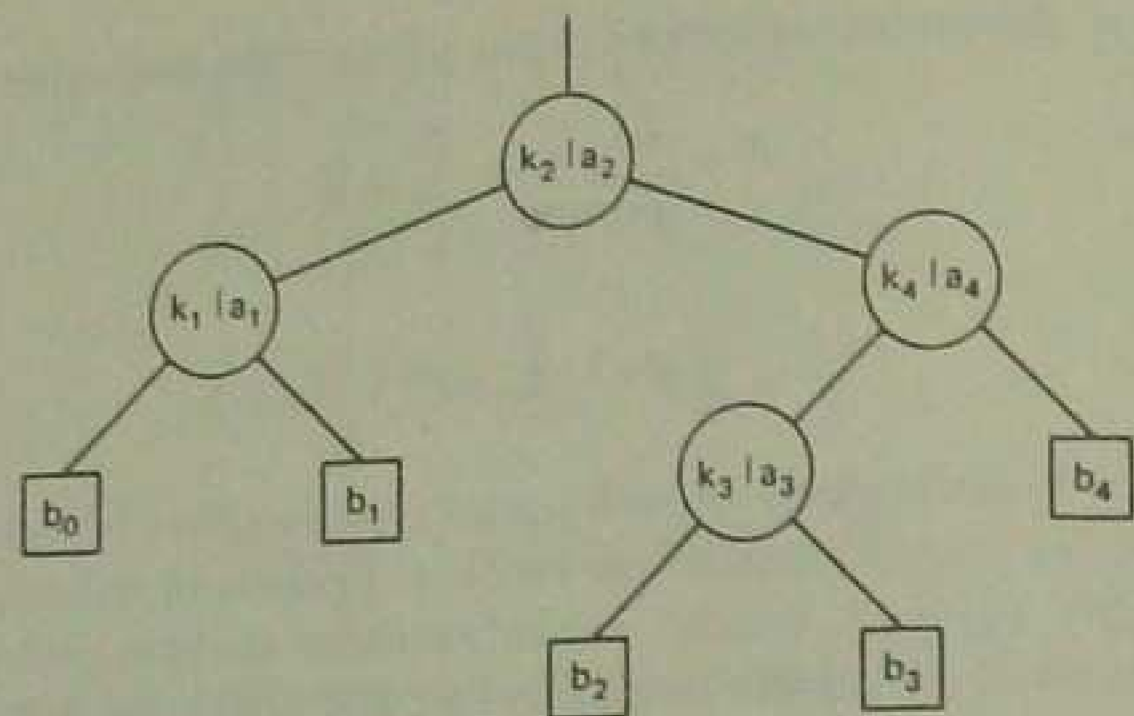
Býva zvykom, že symbolom b_0 sa označuje, koľkokrát je x menšie ako k_1, a symbolom b_n, koľkokrát je x väčšie ako k_n (*obr. 4.37*).

Symbolom P budeme ďalej vyjadrovať kumulovanú váhovanú dĺžku cesty namiesto priemernej dĺžky cesty:

$$P = \sum_{i=1}^{n} a_i h_i + \sum_{j=0}^{n} b_j h'_j \tag{4.68}$$

Takto, okrem toho, že sa vyhneme potrebe vypočítavania pravdepodobnosti z nameraných frekvenčných údajov, získame pri našom hľadaní optimálneho stromu ďalšiu výhodu, a to možnosť používania iba celých čísel.

Vzhľadom na to, že počet možných konfigurácií n vrcholov rastie exponenciálne s ich počtom n, zdá sa, že úloha hľadania optima bude



Obr. 4.37. Vyhľadávací strom spolu s frekvenciami prístupu

pre veľké n dosť beznádejná. Optimálne stromy však majú jednu významnú vlastnosť, ktorá je výhodná pri vyhľadávaní: všetky ich podstromy sú tiež optimálne. Napríklad ak je strom na *obr. 4.37* pre dané a, b optimálny, aj podstrom s kľúčmi k_3 a k_4 je optimálny. Táto vlastnosť vedie k návrhu algoritmu, ktorý systematicky hľadá väčšie a väčšie stromy, začínajúc s jednotlivými vrcholmi ako s najmenšími možnými podstromami. Strom takto rastie „od listov ku koreňu", čo predstavuje vzhľadom na naše kreslenie stromov obrátene smer „zdola nahor" [4-6].

Jadrom tohto algoritmu je rovnica (4.69). Nech P je váhovaná dĺžka cesty stromu a P_L, resp. P_R dĺžky ľavého, resp. pravého podstromu koreňa. Potom P sa rovná súčtu P_L, P_R a počtu, koľkokrát sa uskutočnilo vyhľadávanie jednou vetvou až ku koreňu, čo vlastne predstavuje celkový počet W vyhľadávacích pokusov.

$$P = P_L + W + P_R \tag{4.69}$$

$$W = \sum_{i=1}^{n} a_i + \sum_{j=0}^{n} b_j \tag{4.70}$$

Symbolom W sme označili váhu stromu. Jeho priemerná dĺžka cesty je potom P/W.

Z týchto úvah vyplýva potreba označiť váhy a dĺžky ciest každého

podstromu skladajúceho sa z počtu pripojených kľúčov. Označme preto symbolom w_{ij} váhu a symbolom p_{ij} dĺžku cesty optimálneho podstromu T_{ij}, skladajúceho sa z vrcholov s kľúčmi $k_{i+1}, k_{i+2}, \ldots, k_j$. Tieto veličiny sú definované pomocou rekurentných vzťahov (4.71) a (4.72):

$$\begin{aligned} w_{ii} &= b_i & (0 \leq i \leq n) \\ w_{ij} &= w_{i,j-1} + a_j + b_j & (0 \leq i < j \leq n) \end{aligned} \tag{4.71}$$

$$\begin{aligned} p_{ii} &= w_{ii} & (0 \leq i \leq n) \\ p_{ij} &= w_{ij} + \min_{i<k \leq j} (p_{i,k-1} + p_{kj}) & (0 \leq i < j \leq n) \end{aligned} \tag{4.72}$$

Posledný vzťah vyplýva bezprostredne z rovnice (4.69) a z definície optimálnosti.

Pretože existuje približne $(1/2)\,n^2$ hodnôt p_{ij} a zo vzťahu (4.72) vyplýva nevyhnutnosť voľby medzi $0 < j - i \leq n$ prípadmi, bude operácia minimalizácie vyžadovať približne $(1/6)\,n^3$ operácií. KNUTH zistil, že faktor n možno ušetriť na základe nasledujúcej úvahy, ktorej dôsledkom je navyše aj praktická použiteľnosť tohto algoritmu.

Nech r_{ij} je hodnota k znamenajúceho minimum vo vzťahu (4.72). Vyhľadávanie pre r_{ij} potom možno obmedziť na oveľa menší interval, t.j. môžeme zmenšiť počet $j - i$ vyhodnocovacích krokov. Kľúčovým momentom je zistenie, že ak raz nájdeme koreň r_{ij} optimálneho podstromu T_{ij}, tak už ani rozšírenie stromu pridaním vrcholu do jeho pravého podstromu, ani odstránenie jeho najľavejšieho vrcholu nemôže spôsobiť posunutie tohto koreňa smerom doľava. Túto skutočnosť možno vyjadriť vzťahom

$$r_{i,j-1} \leq r_{ij} \leq r_{i+1,j} \tag{4.73}$$

z ktorého vyplýva, že rozsah možných riešení hľadaného r_{ij} sa redukuje na interval $r_{i,j-1} \ldots r_{i+1,j}$, čo môže znamenať až $O(n^2)$ základných krokov. Teraz už môžeme začať s podrobnou konštrukciou optimalizačného algoritmu. Budeme sa pritom odvolávať na nasledujúce definície, týkajúce sa optimálnych stromov T_{ij}, ktorých vrcholy obsahujú kľúče $k_{i+1}, \ldots, k_j$.

1. a_i: frekvencia vyhľadávania pre kľúč k_i.
2. b_j: frekvencia vyhľadávania argumentu x medzi vrcholmi s kľúčmi k_j a k_{j+1}.

3. w_{ij}: váha stromu T_{ij}.
4. p_{ij}: váhovaná dĺžka cesty stromu T_{ij}.
5. r_{ij}: index koreňa stromu T_{ij}.

Prostredníctvom typu index, definovaného ako

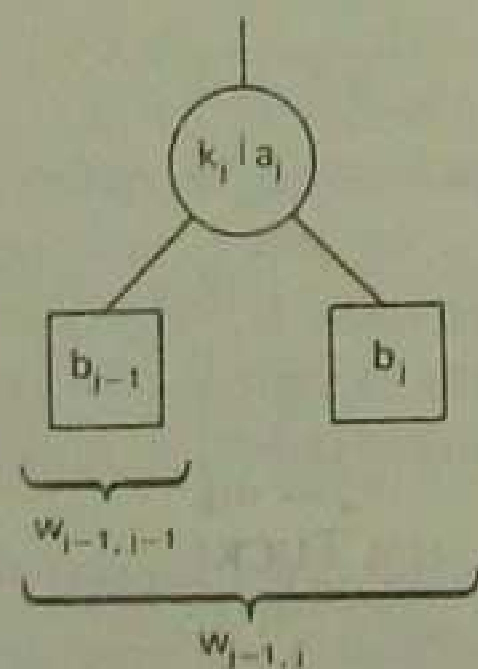
type index = 0 . . n

môžeme definovať nasledujúce polia:

$$
\begin{aligned}
a&: \textbf{array } [1\,..\,n] \textbf{ of } \text{integer};\\
b&: \textbf{array } [\text{index}] \textbf{ of } \text{integer};\\
p, w&: \textbf{array}[\text{index, index}] \textbf{ of } \text{integer};\\
r&: \textbf{array } [\text{index, index}] \textbf{ of } \text{index}
\end{aligned}
\tag{4.74}
$$

Predpokladajme, že sa váha w_{ij} vypočítala priamočiaro, z príslušných hodnôt a, b [pozri (4.71)]. Uvažujme ďalej, že w bude parametrom procedúry, podľa ktorej postupujeme, a r bude jej výsledkom, pretože r úplne charakterizuje danú štruktúru. Nakoniec predpokladajme, že p bude obsahovať priebežné výsledky. Náš postup začneme pri najmenších možných podstromoch, t.j. takých, ktoré neobsahujú žiadne vrcholy, a potom budeme postupovať stále k väčším a väčším stromom. Označme symbolom h šírku $j - i$ podstromu T_{ij}. Potom môžeme, na základe vzťahu (4.72), jednoducho odvodiť hodnoty p_{ii} pre všetky stromy so šírkou $h = 0$.

for i: = 0 **to** n **do** $p[i, i]$: = $w[i, i]$ (4.75)



Obr. 4.38. Optimálny strom s jedným vrcholom

V prípade, že šírka $h = 1$, pôjde o stromy pozostávajúce z jedného vrcholu, ktorý je, pochopiteľne, i koreňom (*obr. 4.38*).

```
for i: = 0 to n − 1 do
   begin j: = i + 1; p[i, j]: = p[i, i] + p[j, j]; r[i, j]: = j        (4.76)
   end
```

Uvedomme si, že premennou i je určený rozsah ľavého a premennou j pravého indexu uvažovaného stromu T_{ij}. V ostatných prípadoch, t.j. pre $h > 1$, použijeme príkaz cyklu, ktorého riadiaca premenná h bude nadobúdať hodnoty v rozsahu od 2 do n. V prípade $h = n$ budeme mať do činenia s úplne „rozvetveným" stromom $T_{0,N}$. V každom z uvedených prípadov možno minimálnu dĺžku cesty p_{ij} a príslušný index koreňa r_{ij} stanoviť pomocou jednoduchého príkazu cyklu, ktorého riadiaca premenná — index k — bude nadobúdať hodnoty z intervalu daného vzťahom (4.73).

```
for h: = 2 to n do
for i: = 0 to n − h do
begin j: = i + h;
   „nájdenie hodnôt m a min = minimum (p[i, m − 1] + p[m, j])
   pre všetky m tak, aby platilo                                   (4.77)
      r[i, j − 1] ≦ m ≦ r[i + 1, j]";
   p[i, j]: = min + w[i, j]; r[i, j]: = m
end
```

Podrobnosti týkajúce sa zjemnenia príkazu, uvedeného v úvodzovkách, možno vidieť v programe 4.6. Priemerná dĺžka cesty stromu $T_{0,n}$ je potom daná podielom $p_{0,n}/w_{0,n}$ a koreňom tohto stromu je vrchol určený indexom $r_{0,n}$.

Z algoritmu (4.77) je jasné, že úsilie potrebné na stanovenie optimálnej štruktúry je rádovo $O(n^2)$; veľkosť potrebnej operačnej pamäti je tiež $O(n^2)$. Tieto ukazovatele sú pre veľké n, prirodzene, neprijateľné, preto sa žiadajú oveľa efektívnejšie algoritmy. Jedným z nich je algoritmus, ktorého autormi sú HU a TUCKER [4-5], ktorý vyžaduje iba $O(n)$ pamäti a $O(n \cdot \log n)$ výpočtových operácií. Tento algoritmus však funguje iba pre prípad nulových frekvencií kľúčov ($a_i = 0$), t.j. vtedy, keď sa registrujú iba neúspešné vyhľadávacie pokusy.

Iný algoritmus, ktorý tiež vyžaduje $O(n)$ pamäti a $O(n \cdot \log n)$ výpočtových operácií, navrhli WALKER a GOTLIEB [4-11]. Namiesto snahy nájsť optimum poskytuje tento algoritmus takmer optimálny strom. Preto môže byť založený na heuristických princípoch. Základnú myšlienku môžeme zhrnúť takto:

Predstavme si, že by sme vrcholy (pravé i špeciálne) rozmiestnili na lineárnej stupnici, pričom by boli váhované svojimi frekvenciami (alebo pravdepodobnosťami) prístupu. Nájdime vrchol, ktorý je najbližšie k „stredu váhy". Jeho vrchol nazývame ťažisko a jeho index je

$$\frac{1}{w}\left(\sum_{i=1}^{n} i \cdot a_i + \sum_{j=0}^{n} j \cdot b_j\right) \tag{4.78}$$

zaokrúhlený na najbližšie celé číslo. Ak by mali všetky vrcholy rovnakú váhu, bol by koreň hľadaného optimálneho stromu zjavne totožný s ťažiskom. Vo väčšine prípadov sa však tento koreň bude nachádzať v tesnej blízkosti ťažiska. Isté obmedzené vyhľadávanie sa potom používa na zistenie lokálneho optima; v zápätí sa táto procedúra aplikuje na obidva vzniknuté podstromy. Pravdepodobnosť výskytu koreňa v tesnej blízkosti ťažiska sa zväčšuje s veľkosťou stromu n. Len čo podstromy dosiahnu istú „ovládateľnú" veľkosť, možno určiť ich optimum pomocou uvedeného presného algoritmu.

4.4.10 ZOBRAZENIE STROMOVEJ ŠTRUKTÚRY

Obráťme teraz našu pozornosť na programovací problém, ktorý úzko súvisí so stromovou problematikou. Pôjde nám o vygenerovanie dostatočne prehľadnej a zrozumiteľnej grafickej podoby stromovej štruktúry na výstupnom zariadení počítača, ktorým nech je normálna tlačiareň. To znamená, že chceme načrtnúť obraz stromu, vytlačiť jeho kľúče ako vrcholy a pospájať ich primeranými vodorovnými a zvislými znakmi (tak, aby vetvy stromu boli zobrazené neprerušovanými čiarami).

Pri manipulácii s riadkovou tlačiarňou, ktorej údaje reprezentujeme pomocou textového súboru, t. j. postupnosťou znakov, môžeme postupovať iba v prísne určenom smere — zľava doprava a zhora nadol.

Preto sa zdá byť rozumnejšie, ak najprv vybudujeme reprezentáciu stromu, ktorá veľmi presne odzrkadľuje jeho topologickú štruktúru. Druhý krok potom predstavuje zobrazenie tohto obrazu v primeranej podobe na stránku tlačiarne a vypočítanie presných súradníc vrcholov a hrán.

Pri uskutočňovaní prvej úlohy môžeme využiť naše skúsenosti s algoritmami generujúcimi stromové štruktúry a bez váhania prispôsobiť rekurzívne riešenie rekurzívne definovanému problému. Podobne ako v programe 4.3 sformulujeme funkciu strom. Parametre i, j budú predstavovať medzné indexy vrcholov patriacich k stromu. Koreň stromu bude predstavovať vrchol s indexom r_{ij}. Ale predtým, ako budeme pokračovať, musíme definovať typ premenných reprezentujúcich vrcholy stromu. Tieto musia obsahovať dva smerníky na svoje podstromy a kľúč vrcholu. Vzhľadom na isté zámery, o ktorých budeme hovoriť v rámci druhého kroku, musí každá premenná typu vrchol obsahovať ešte dve zložky, ktoré nazývame *poz* a *link*. Navrhnutú definíciu možno vidieť v schéme (4.79) a výslednú funkciu v programe 4.6.

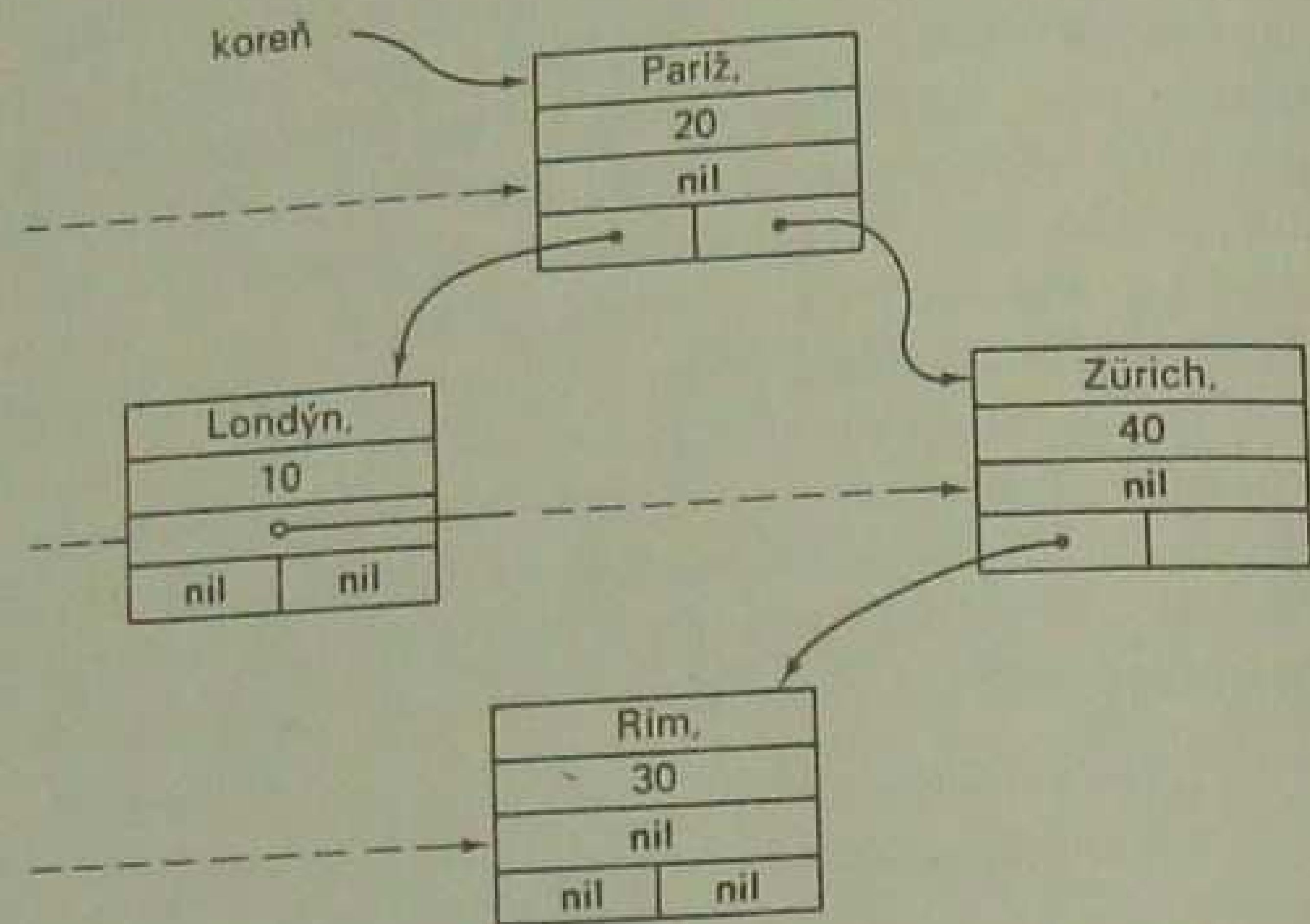
```
type ref = ↑vrchol;
  vrchol = record kľúč: alfa;
                  poz: riadkovápozícia;          (4.79)
                  ľavý, pravý, link: ref
           end
```

Uvedomme si, že táto procedúra počíta množstvo vygenerovaných vrcholov pomocou globálnej premennej k. k-tému vrcholu sa priradí k-tý kľúč a pretože kľúče sú abecedne usporiadané, súčin premennej k a konštantnej mierky určuje vodorovnú súradnicu každého kľúča, ktorá sa uchováva spolu s ďalšími informáciami. Všimnime si tiež, že sme upustili od doterajšej zvyklosti používať výlučne celé čísla ako kľúče, a predpokladáme, že sú typu alfa, reprezentovaného poľom znakov s danou (maximálnou) dĺžkou. Táto dĺžka súčasne určuje počet platných znakov kľúča, nad ktorými je definované abecedné usporiadanie.

Aby sme si vytvorili obraz o tom, čo sme až doteraz prebrali, pozrime sa na *obr. 4.39*. Pre danú množinu n kľúčov a vypočítanú maticu r_{ij} spôsobia príkazy

$k := 0$; *koreň* := strom (0, n)

vygenerovanie predbežne pospájanej stromovej štruktúry so znázornenými vodorovnými pozíciami vrcholov a zvislými pozíciami určenými implicitne na základe ich úrovne v strome.



Obr. 4.39. Strom generovaný programom 4.6

Teraz môžeme pokračovať druhým krokom — zobrazením stromu na papier. V tomto prípade budeme musieť postupovať výlučne od koreňovej úrovne nadol, pričom v každom kroku spracujeme jeden riadok vrcholov. Ale ako sa dostaneme k vrcholom, ktoré sa majú nachádzať v jednom rade? Za týmto účelom, t.j. nájsť a pospájať vrcholy jedného riadku, sme zaviedli spomenutú záznamovú zložku *link*. Pomocou nej sa realizuje zreťazenie vrcholov, ktoré je na *obr. 4.39* znázornené prerušovanou čiarou. V každom kroku teda predpokladáme prítomnosť zreťazenia vrcholov, ktoré sa majú vytlačiť. Toto zreťazenie označíme ako momentálne na rozdiel od druhého zreťazenia, ktorým sú pospájaní nasledovníci spracovávaného vrcholu a ktoré nazveme nasledujúcim. Pri zostúpení na nižšiu úroveň sa najprv nasledujúce zreťazenie stane momentálnym a potom sa označí ako prázdne.

Podrobnosti algoritmu možno vidieť v programe 4.6. Nasledujúce poznámky uvádzame na objasnenie niektorých možných sporných miest tohto algoritmu:

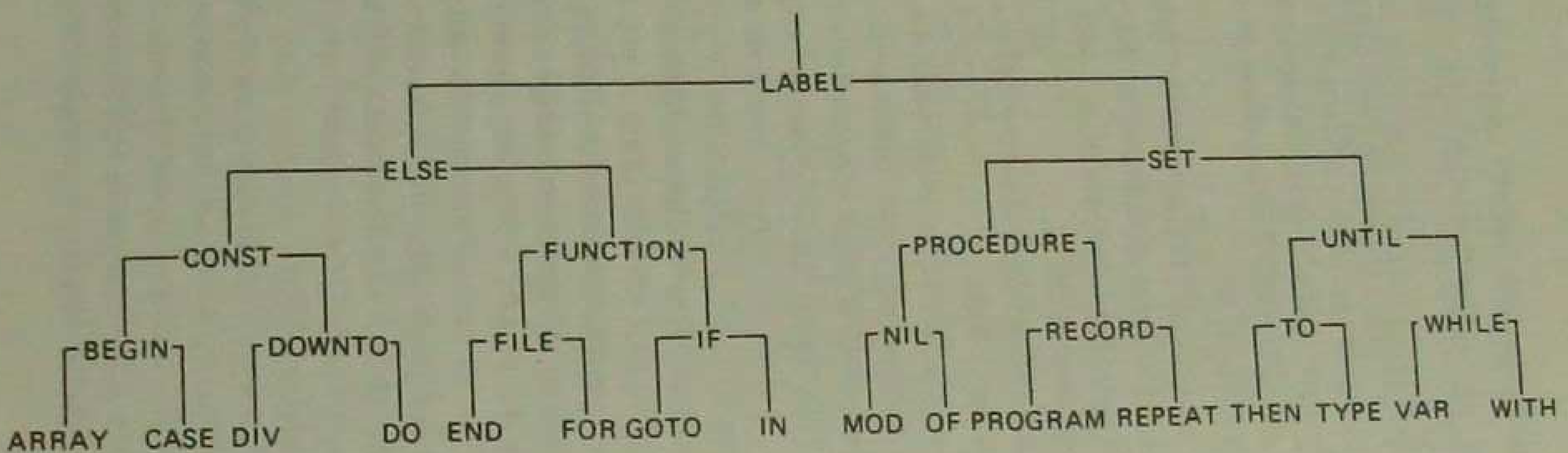
1. Zreťazené vrcholy jedného riadku sa generujú zľava doprava, čo znamená, že vrchol, ktorý sa nachádza celkom vľavo, bude vygenerovaný ako posledný. Pretože vrcholy sa navštevujú v tom istom poradí, musí byť ich zoznam invertovaný. Táto inverzia sa uskutoční súčasne so zmenou uvedených zreťazení.

2. Tlačený riadok zobrazujúci kľúče vrcholov, nazývaný tiež hlavný alebo matričný riadok, obsahuje aj vodorovné čiary (*obr. 4.40*). Premenné $u1$, $u2$, $u3$, $u4$ označujú začiatočné a koncové pozície ľavých a pravých vodorovných čiar vrcholu.

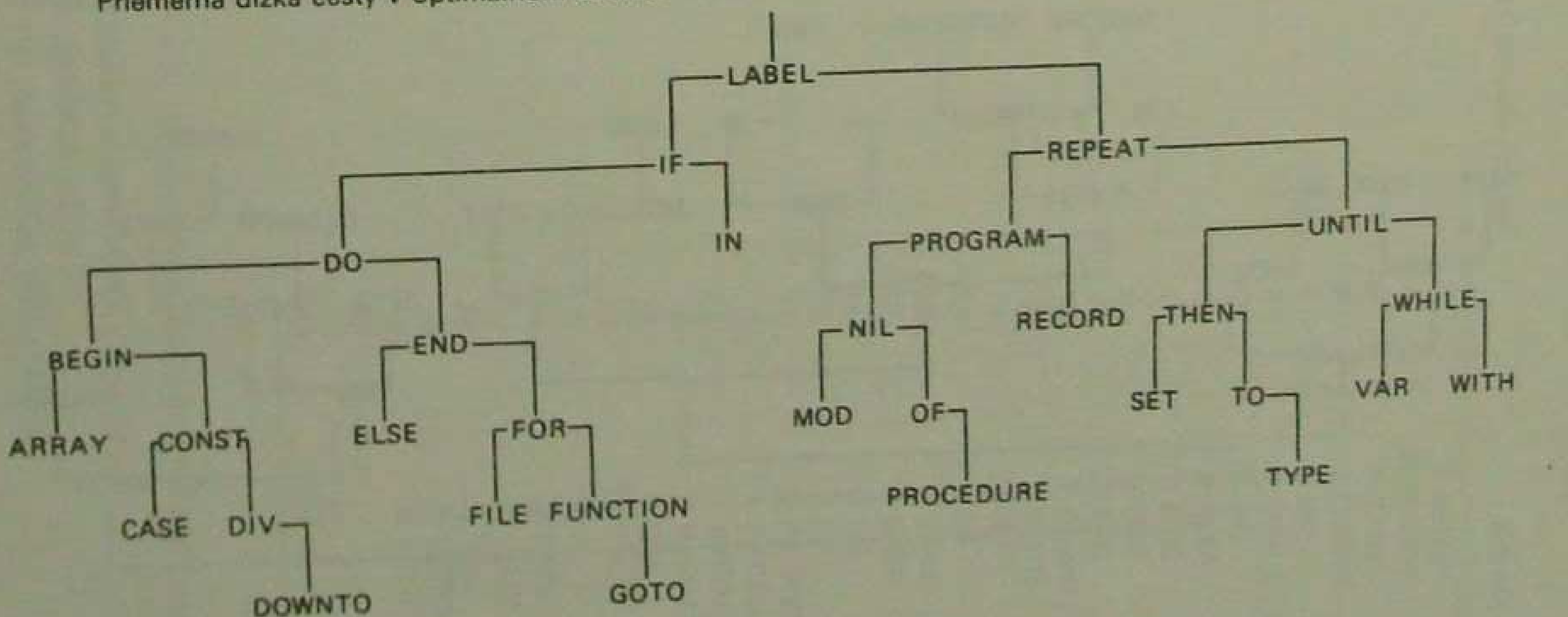
3. Pred zobrazením každého matričného riadku sa vytlačia tri riadky znázorňujúce zvislé časti hrán.

Pristúpme teraz k opisu štruktúry programu 4.6. Jeho dve hlavné časti predstavujú procedúru nájdenia optimálneho vyhľadávacieho stromu pre danú váhovú distribúciu w a procedúru zobrazenia stromu pre dané indexy r. Celý program je prispôsobený na spracúvanie programových textov, najmä však tých, ktoré sú napísané v jazyku pascal. Jednotlivé fázy programu pracujú takto: V prvej fáze sa číta text programu, rozpoznávajú sa identifikátory a kľúčové slová, na základe ktorých dostaneme frekvenciu vyhľadávania a_i pre kľúč k_i a frekvenciu vyhľadávania b_j pre identifikátory medzi kľúčmi k_j a k_{j+1}. Len čo sa vytlačí frekvenčná štatistika, program pokračuje ďalšou fázou, v ktorej sa vypočíta dĺžka cesty dokonale vyváženého stromu a určia sa korene jeho podstromov. Potom sa zobrazí priemerná váhovaná dĺžka cesty a celý strom. V tretej fáze sa aktivuje procedúra optstrom, ktorá vytvorí optimálny vyhľadávací strom. Vzápätí sa tento i vytlačí. Nakoniec, pri použití tých istých procedúr, sa vytvorí a zobrazí optimálny strom, ale iba vzhľadom na frekvencie kľúčov.

Tab. 4.5 a *obr. 4.40* až *obr. 4.42* zobrazujú výsledky programu 4.6, aplikovaného na svoj vlastný zdrojový text. Rozdiely v týchto troch obrázkoch potvrdzujú, že vyvážený strom ešte nemožno pokladať ani len za takmer optimálny a že frekvencie nekľúčových identifikátorov výrazne ovplyvňujú voľbu optimálnej štruktúry.

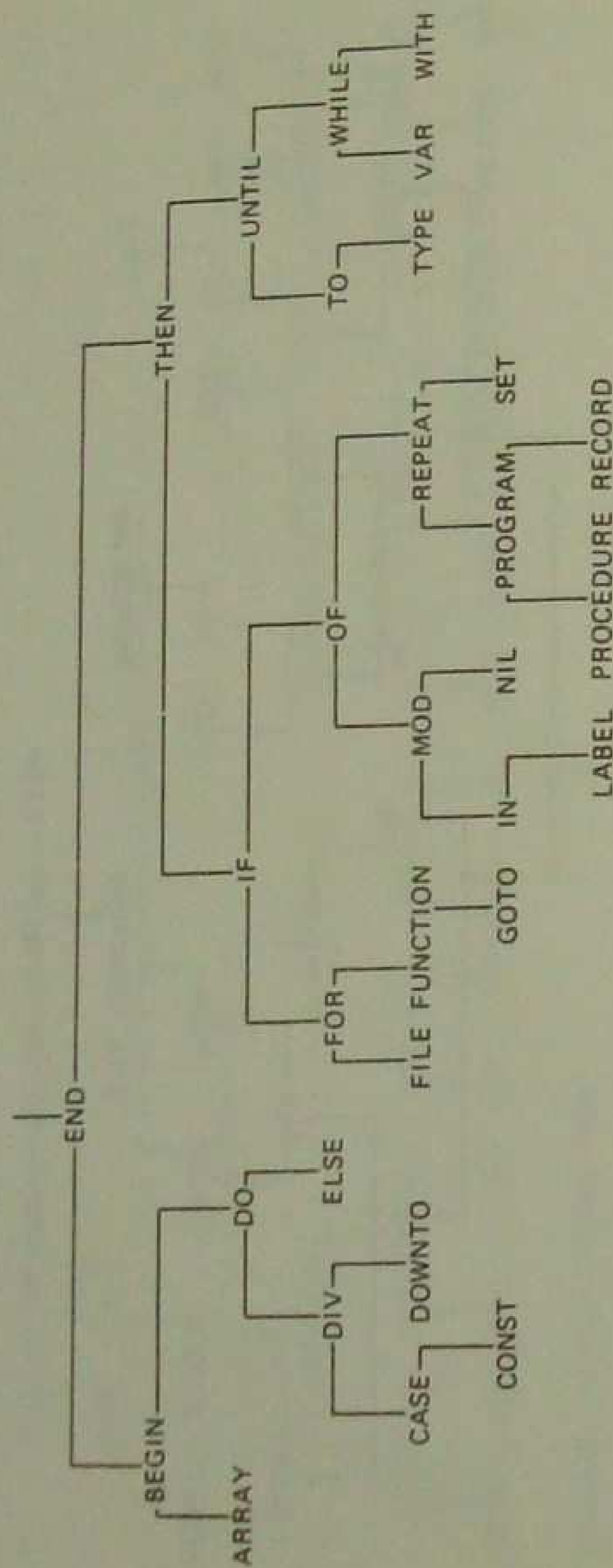


Obr. 4.40. Dokonale vyvážený strom



Obr. 4.41. Optimálny vyhľadávací strom

Optimálny strom zohľadňujúci iba kľúče



Obr. 4.42. Optimálny strom zohľadňujúci iba kľúčové slová

Kľúče a frekvencie výskytu *Tabuľka 4.5*

4	7	ARRAY
14	27	BEGIN
19	0	CASE
15	2	CONST
8	5	DIV
0	0	DOWNTO
0	20	DO
0	8	ELSE
0	28	END
1	0	FILE
0	12	FOR
0	2	FUNCTION
0	0	GOTO
9	13	IF
23	2	IN
208	0	LABEL
22	0	MOD
17	10	NIL
24	7	OF
17	2	PROCEDURE
0	1	PROGRAM
53	1	RECORD
6	8	REPEAT
16	0	SET
10	13	THEN
0	12	TO
6	2	TYPE
1	8	UNTIL
39	5	VAR
0	8	WHILE
0	0	WITH
37		
549	203	

PROGRAM 4.6. *Nájdenie optimálneho vyhľadávacieho stromu*

program Optimálnystrom (input, output);
const $n = 31$; {počet kľúčov}
    $k1n = 10$; {maximálna dĺžka kľúča}

```
type index = 0..n;
     alfa = packed array [1..kln] of char;
var ch: char;
    k1, k2: integer;
    id: alfa; {identifikátor alebo kľúč}
    buf: array [1..kln] of char; {vyrovnávacia znaková pamäť}
    kľúč: array [1..n] of alfa;
    i, j, k: integer;
    a: array [1..n] of integer;
    b: array [index] of integer;
    p, w: array [index, index] of integer;
    r: array [index, index] of index;
    suma, sumb: integer;
function balstrom(i, j: index): integer;
  var k: integer;
begin k: = (i + j + 1) div 2; r[i, j]: = k;
  if i ≧ j then balstrom: = b[k] else
    balstrom: = balstrom(i, k − 1) + balstrom(k, j) + w[i, j]
end {balstrom};
procedure optstrom;
  var x, min: integer;
      i, j, k, h, m: index;
begin {parameter: w, výsledok: p, r}
  for i: = 0 to n do p[i, i]: = w[i, i];  {šírka stromu h = 0}
  for i: = 0 to n − 1 do                 {šírka stromu h = 1}
  begin j: = i + 1;
    p[i, j]: = p[i, i] + p[j, j]; r[i, j]: = j
  end;
  for h: = 2 to n do        {h = šírka uvažovaného stromu}
  for i: = 0 to n − h do    {i = ľavý index uvažovaného stromu}
  begin j: = i + h; {j = pravý index uvažovaného stromu}
    m: = r[i, j − 1]; min: = p[i, m − 1] + p[m, j];
    for k: = m + 1 to r[i + 1, j] do
    begin x: = p[i, k − 1] + p[k, j];
      if x < min then
```



```
        begin m: = k; min: = x
        end
    end;
    p[i, j]: = min + w[i, j]; r[i, j]: = m
  end
end {optstrom};
procedure vytlačstrom;
  const lw = 120;  {šírka riadku tlačiarne}
  type ref = ↑vrchol;
       riadkovápozícia = 0..lw;
       vrchol = record kľúč: alfa;
                       pos: riadkovápozícia;
                       ľavý, pravý, link: ref
                end;
  var koreň, momentálny, nasledujúci: ref;
      q, q1, q2: ref;
      i, k: integer;
      u, u1, u2, u3, u4: riadkovápozícia;
  function strom(i, j: index): ref;
    var p: ref;
  begin if i = j then p: = nil else
    begin new(p);
      p↑.ľavý: = strom(i, r[i, j] − 1);
      p↑.pos: = trunc((lw − kln) * k/(n − 1)) + (kln div 2);
                                        k: = k + 1;
      p↑.kľúč: = kľúč[r[i, j]];
      p↑.pravý: = strom(r[i, j], j)
    end;
    strom: = p
  end;
  begin k: = 0; koreň: = strom(0, n);
    momentálny: = koreň; koreň↑.link: = nil;
    nasledujúci: = nil;
    while momentálny ≠ nil do
    begin {postup smerom nadol; najprv sa zobrazia zvislé riadky}
```

```
      for i: = 1 to 3 do
      begin u: = 0; q: = momentálny;
        repeat u1: = q↑.pos;
          repeat write(' '); u: = u + 1
          until u = u1;
          write('|'); u: = u + 1; q: = q↑.link
        until q = nil;
        writeln
      end;
      {teraz sa zobrazí matričný riadok; zostúpi sa o 1 úroveň a vytvo-
      rí sa ďalší zoznam — z nasledovníkov vrcholov momentálneho
      zoznamu}
    q: = momentálny; u: = 0;
    repeat unpack(q↑.kľúč, buf, 1);
      {centralizácia kľúča vzhľadom na pozíciu pos}
      i: = kln;
      while buf[i] = ' ' do i: = i − 1;
      u2: = q↑.pos − ((i − 1) div 2); u3: = u2 + i;
      q1: = q↑.ľavý; p2: = q↑.pravý;
      if q1 = nil then u1: = u2 else
        begin u1 = q1↑.pos; q1↑.link: = nasledujúci;
                        nasledujúci: = q1
        end;
      if q2 = nil then u4: = u3 else
        begin u4: = q2↑.pos + 1; q2↑.link: = nasledujúci;
                              nasledujúci: = q2
        end;
      i: = 0;
      while u < u1 do begin write(' '); u: = u + 1 end;
      while u < u2 do begin write('−'); u: = u + 1 end;
      while u < u3 do begin i: = i + 1; write(buf[i]); u: = u + 1
                      end;
      while u < u4 do begin write('−'); u: = u + 1
                      end;
      q: = q↑.link
    until q = nil;
```



```
        writeln;
        {inverzia ďalšieho zoznamu, ktorý sa stane momentálnym}
        momentálny: = nil;
        while nasledujúci ≠ nil do
          begin q: = nasledujúci; nasledujúci: = q↑.link;
                q↑.link: = momentálny; momentálny: = q
          end
        end
      end {vytlačstrom};
    begin {inicializácia tabuľky kľúčov a počítadiel}
      kľúč  [1]: = 'ARRAY';          kľúč  [2]: = 'BEGIN';
      kľúč  [3]: = 'CASE';           kľúč  [4]: = 'CONST';
      kľúč  [5]: = 'DIV';            kľúč  [6]: = 'DOWNTO';
      kľúč  [7]: = 'DO';             kľúč  [8]: = 'ELSE';
      kľúč  [9]: = 'END';            kľúč[10]: = 'FILE';
      kľúč[11]: = 'FOR';             kľúč[12]: = 'FUNCTION';
      kľúč[13]: = 'GOTO';            kľúč[14]: = 'IF';
      kľúč[15]: = 'IN';              kľúč[16]: = 'LABEL';
      kľúč[17]: = 'MOD';             kľúč[18]: = 'NIL';
      kľúč[19]: = 'OF';              kľúč[20]: = 'PROCEDURE';
      kľúč[21]: = 'PROGRAM';         kľúč[22]: = 'RECORD';
      kľúč[23]: = 'REPEAT';          kľúč[24]: = 'SET';
      kľúč[25]: = 'THEN';            kľúč[26]: = 'TO';
      kľúč[27]: = 'TYPE';            kľúč[28]: = 'UNTIL';
      kľúč[29]: = 'VAR';             kľúč[30]: = 'WHILE';
      kľúč[31]: = 'WITH';
      for i: = 1 to n do
        begin a[i]: = 0; b[i]: = 0 end;
      b[0]: = 0; k2: = kln;
      {čítanie vstupného textu a určenie hodnôt a, b}
      while ¬ eof(input) do
      begin read(ch);
        if ch in ['A' .. 'Z'] then
        begin {identifikátor alebo kľúč} k1: = 0;
          repeat if k1 < kln then
                begin k1: = k1 + 1; buf[k1]: = ch end;
```

```
        read (ch)
        until ¬ ch in ['A' .. 'Z', '0' .. '9'];
        if k1 ≧ k2 then k2 := k1 else
        repeat buf[k2] := ' '; k2 := k2 − 1
        until k2 = k1;
        pack (buf, 1, id);
        i := 1; j := n;
        repeat k := (i + j) div 2;
          if kľúč[k] ≦ id then i := k + 1;
          if kľúč[k] ≧ id then j := k − 1;
        until i > j;
        if kľúč[k] ≧ id then a[k] := a[k] + 1 else
          begin k := (i + j) div 2; b[k] := b[k] + 1 end
    end else
      if ch = '''' then repeat read (ch) until ch = '''' else
      if ch = '{' then repeat read (ch) until ch = '}'
  end;
  writeln ('KĽÚČE A FREKVENCIE VÝSKYTU : ');
  suma := 0; sumb := b[0];
  for i := 1 to n do
  begin suma := suma + a[i]; sumb := sumb + b[i];
    writeln (b[i − 1], a[i], ' ', kľúč[i])
  end;
  writeln (b[n]);
  writeln ('   ------   ------');
  writeln (sumb, suma);
  {výpočet w pomocou a, b}
  for i := 0 to n do
  begin w[i, i] := b[i];
    for j := i + 1 to n do w[i, j] := w[i, j − 1] + a[j] + b[j]
  end;
  write ('PRIEMERNÁ DĹŽKA CESTY VO VYVÁŽENOM
        STROME');
  writeln (balstrom (0, n)/w[0, n] : 6 : 3); vytlačstrom; optstrom;
  write ('PRIEMERNÁ DĹŽKA CESTY V OPTIMÁLNOM
        STROME');
```



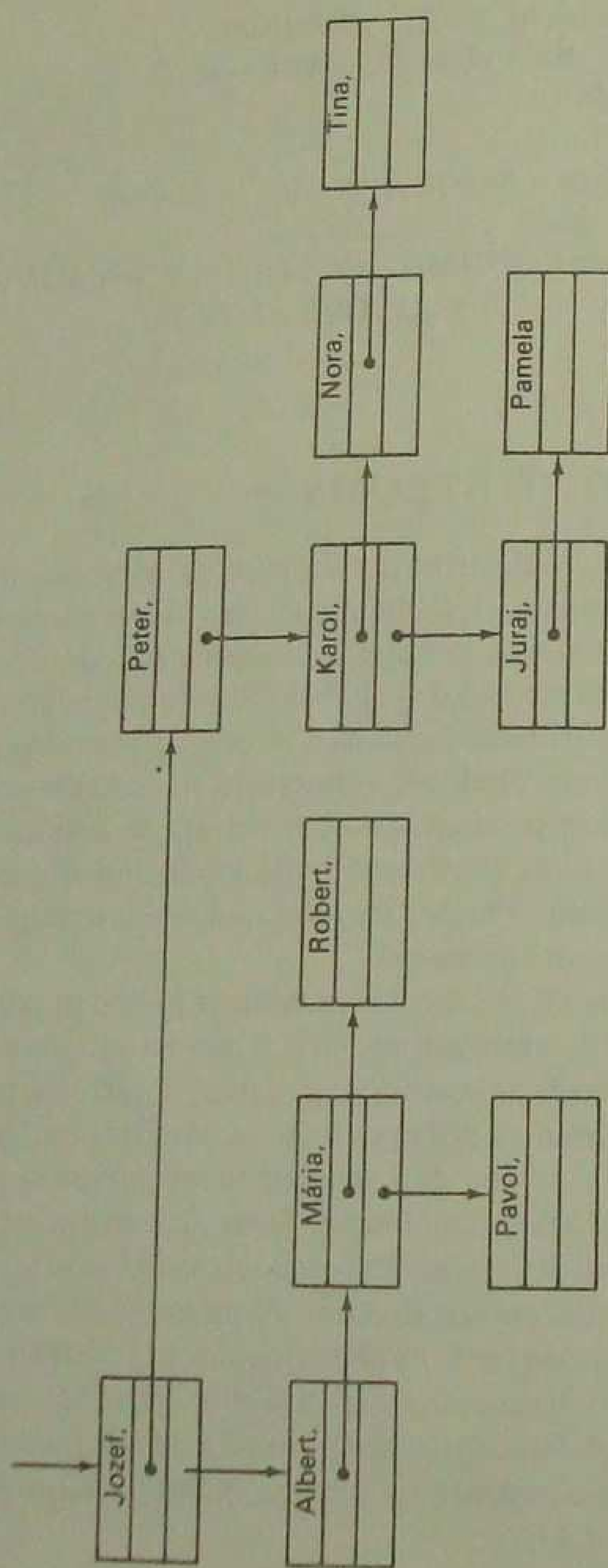
```
  writeln (p[0, n]/w[0, n] : 6 : 3); vytlačstrom;
  {teraz uvažujme iba o kľúčoch, teda b = 0}
  for i := 0 to n do
  begin w[i, i] := 0;
    for j := i + 1 to n do w[i, j] := w[i, j − 1] + a[j]
  end;
  optstrom; writeln ('OPTIMÁLNY STROM ZOHĽADŇUJÚCI
                     IBA KĽÚČOVÉ SLOVÁ');
  vytlačstrom
end.
```

4.5 VIACCESTNÉ STROMY

Doteraz sme skúmali stromy, v ktorých každý vrchol mal maximálne dvoch nasledovníkov, t. j. binárne stromy. Tieto nám plne vyhovujú v takých prípadoch, ako je napr. zobrazenie rodinných vzťahov prostredníctvom rodokmeňa, kde je každý človek spojený so svojimi rodičmi. Nakoniec, nikto nemá viac ako dvoch rodičov! Ale čo urobíme v prípade, keď dáme prednosť zobrazeniu na základe potomstva? Je jasné, že budeme musieť uvážiť skutočnosť, že niektorí ľudia majú viac ako dve deti, na základe čoho budú ich stromy obsahovať vrcholy s viacerými vetvami. Pretože nepoznáme lepší termín, budeme ich nazývať *viaccestnými stromami.*

Samozrejme, na týchto štruktúrach nie je nič mimoriadne a my sme sa už oboznámili so všetkými prostriedkami na programovanie a definovanie údajov, takže sme schopní zvládnuť aj takéto štruktúry. Ak sa napr. určí horná hranica počtu detí (čo je, samozrejme, značne futuristický predpoklad), možno deti reprezentovať pomocou poľa, predstavujúceho jednu zložku záznamovej štruktúry reprezentujúcej osobu. Ak by však počet detí jednotlivých osôb veľmi kolísal, došlo by pri takto zvolenej reprezentácii k veľmi úbohému využitiu pamäti.

V takomto prípade bude výhodnejšie, ak nasledovníkov usporiadame do lineárneho zoznamu so smerníkom (priradenom rodičovi) na najmladšieho (alebo najstaršieho) potomka. Možnú definíciu typu osoba pre tento prípad znázorňuje schéma (4.80) a možnú štruktúru údajov ukazuje *obr. 4.43.*



Obr. 4.43. Viaccestný strom

type osoba = **record** *meno*: alfa;
nevlastný potomok: ↑osoba;
vlastný potomok: ↑osoba (4.80)
end

(V uvedenej definícii rozlišujeme dva typy potomkov — vlastného, ktorý má spoločných oboch rodičov, a nevlastného, ktorý má jedného vlastného a druhého nevlastného rodiča; z anglických termínov offspring a sibling — pozn. prekl.)

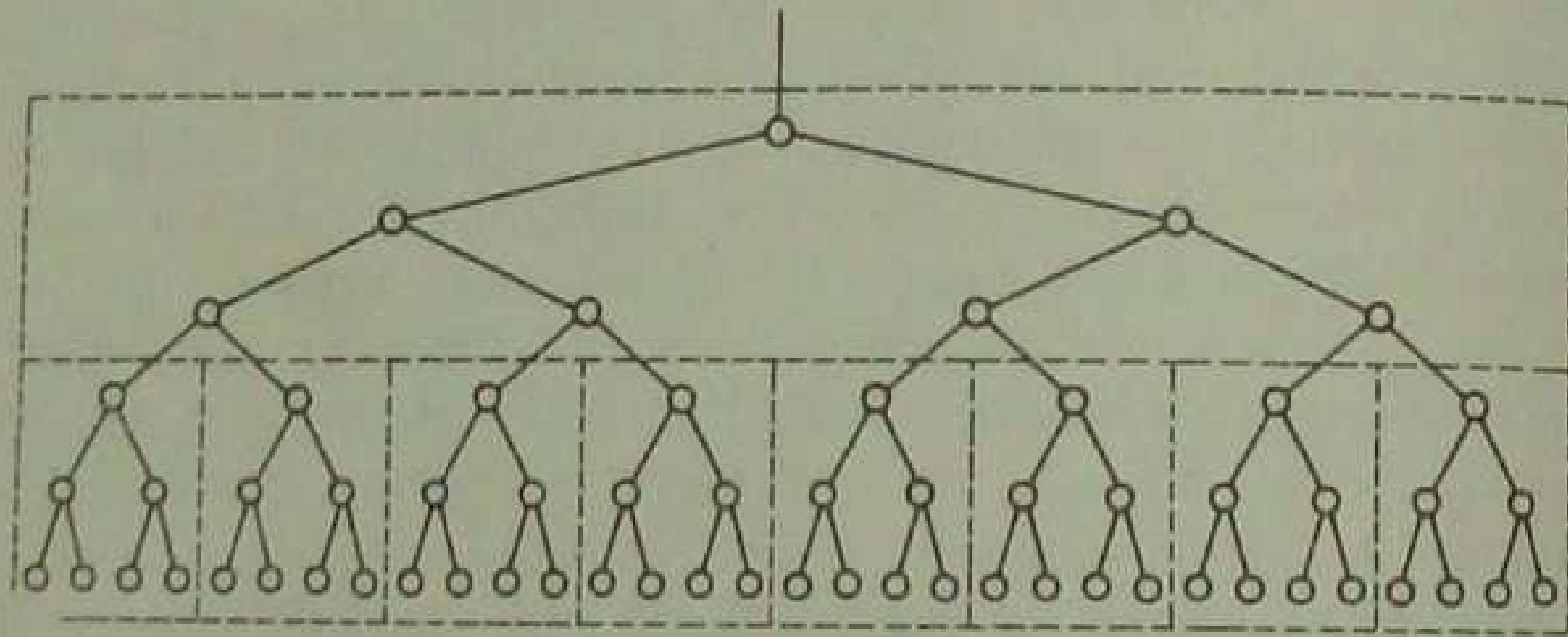
Predstavme si, že by sme tento obrázok otočili o 45°. Výsledkom by bol dokonalý binárny strom, ktorý by však nezodpovedal skutočnosti, pretože dve referencie (uvedené v definícii 4.80) majú funkčne úplne odlišné významy.

Človek obyčajne nezaobchádza rovnako so svojim nevlastným potomkom a so svojím vlastným potomkom, a preto by tak nemal robiť ani pri tvorbe definícií údajov. Uvedený príklad by sa dal jednoducho rozšíriť na zložitejšiu štruktúru, ak by sme pridali ďalšie zložky do každého záznamu osoba, čím by bolo možné vyjadriť ďalšie rodinné vzťahy. Možnými kandidátmi, ktorých nemožno z uvedenej definície odvodiť, by boli manžel a manželka alebo otec a matka. Takáto štruktúra potom rýchlo narastá do zložitej „relačnej banky údajov", v ktorej možno zobraziť i niekoľko stromov. Algoritmy, operujúce na takýchto štruktúrach, bývajú silne späté so svojimi definíciami údajov, a preto nemá význam venovať sa špecifikácii nejakých všeobecných pravidiel alebo široko použiteľných techník.

Existuje však veľmi zaujímavá praktická oblasť, v ktorej by takéto stromy našli patričné uplatnenie. Ide o konštrukciu a údržbu veľkých vyhľadávacích stromov, pri ktorých sa predpokladajú operácie pridávania a odoberania prvkov, ale ktoré nemožno umiestniť celé do operačnej pamäti počítača (buď pre spomenutú veľkosť, alebo preto, že dlhodobejšie obsadenie pamäti je príliš drahé).

Predpokladajme teda, že vrcholy stromu sú uložené na sekundárnych pamäťových médiách, akými sú napr. diskové pamäti. Potom môžeme s výhodou použiť dynamické štruktúry údajov, zavedené v tejto kapitole, a ich jednoduchým prispôsobením zabezpečiť aj prístup k sekundárnym pamätiam. Zásadnou inováciou je iba to, že smerniky

budú reprezentované adresami diskovej pamäti namiesto operačnej pamäti. Použitie binárneho stromu na organizáciu uloženia množiny údajov pozostávajúcej, povedzme, z milióna položiek, vyžaduje v priemere približne $\log_2 10^6 \cong 20$ vyhľadávacích krokov. Pretože každý krok vyžaduje jeden prístup na disk (s patričnou čakacou dobou), bola by žiadúca taká organizácia pamäti, ktorá nevyžaduje toľko prístupov. Viaccestné stromy sú najvhodnejším riešením tohto problému. Ak je sprístupnená určitá položka sekundárnej pamäti, je prístupná aj celá množina ďalších položiek (bez ďalších nákladov). To však predpokladá, že strom bude rozdelený na podstromy reprezentované ako pamäťové jednotky (bloky), ku ktorým môžeme pristupovať naraz. Tieto podstromy budeme nazývať *stránky*. Na *obr. 4.44* je znázornený binárny strom rozdelený na stránky, pričom každá stránka obsahuje 7 vrcholov.



Obr. 4.44. Binárny strom rozdelený na stránky

Úspory v počte prístupov na disk môžu byť značné (každý prístup k stránke vyžaduje teraz jeden prístup na disk). Predpokladajme, že na jednu stránku umiestnime 100 vrcholov (zdá sa to ako výhodné číslo); vyhľadávanie v strome s milión vrcholmi bude potom vyžadovať iba $\log_{100} 10^6 = 3$ prístupy k stránke namiesto dvadsiatich! Pravda, ak strom ponecháme rásť „náhodne", tak najhorší prípad môže byť ešte stále 10^4! Je jasné, že určitá schéma na usmerňovanie rastu bude v prípade viaccestných stromov zvlášť potrebná.

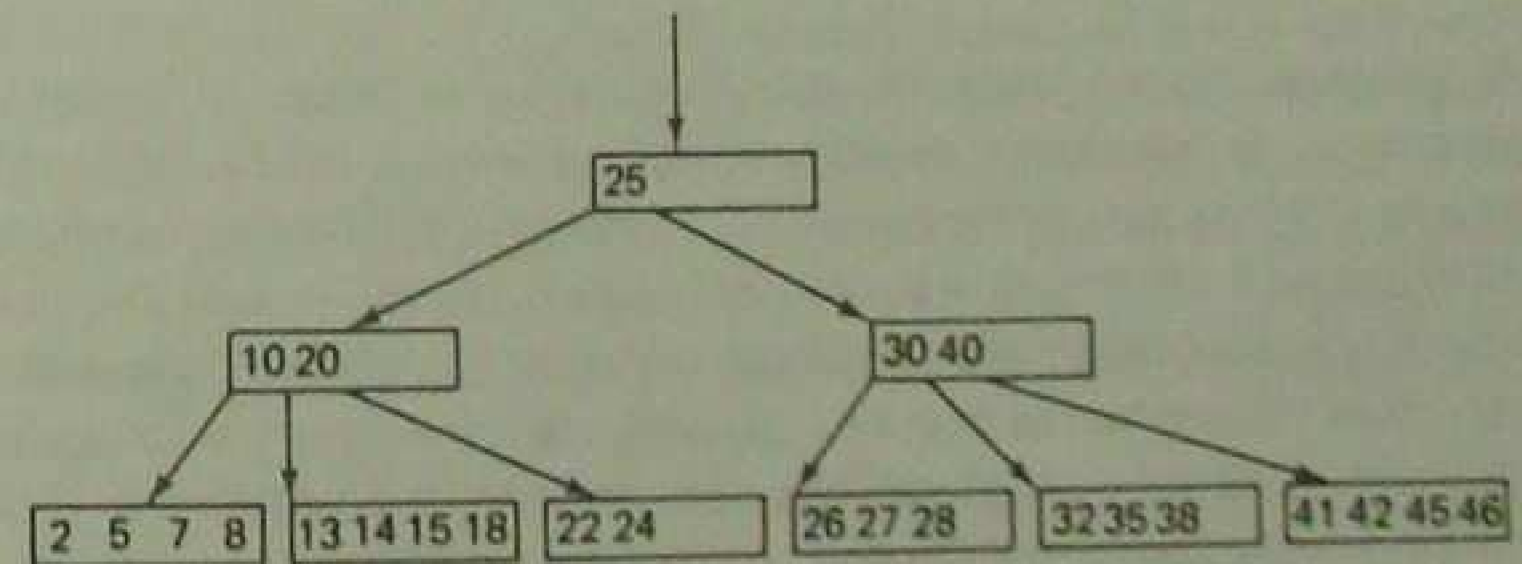
4.5.1 B-STROMY

Ak hľadáme kritériá na usmerňovanie rastu stromu, tak rýchlo vylúčime dokonalé vyváženie, pretože vyžaduje príliš mnoho vyvažovacích operácií. Pravidlá musia byť teda o niečo miernejšie. Veľmi múdre kritérium navrhol R. BAYER [4.2] v roku 1970: Každá stránka (okrem jednej) musí obsahovať n až $2n$ vrcholov pre danú konštantu n. Potom pre strom s N položkami a maximálnou veľkosťou stránky $2n$ vrcholov bude najhorší prípad vyžadovať $\log_n N$ prístupov k stránkam. Vidíme, že prístupy k stránkam v celom vyhľadávaní jasne dominujú. Navyše dôležitý faktor využitia pamäti je aspoň 50 %, pretože stránky sú vždy aspoň do polovice zaplnené. Je potešiteľné, že pri všetkých týchto výhodách budú algoritmy vyhľadávania, pridávania a odoberania pomerne jednoduché. Podrobnejšie sa im budeme venovať neskôr.

Štruktúra, ktorá spĺňa uvedené kritériá, sa nazýva *B-strom* a jej charakteristika vyzerá takto (symbolom n vyjadrujeme *rád* B-stromu):

1. Každá stránka obsahuje najviac $2n$ položiek (kľúčov).
2. Každá stránka, okrem stránky koreňa, obsahuje aspoň n položiek.
3. Každá stránka je buď listovou stránkou, t. j. nemá nasledovníkov, alebo má $m + 1$ nasledovníkov, pričom m je počet jej kľúčov.
4. Všetky listové stránky sú na jednej úrovni.

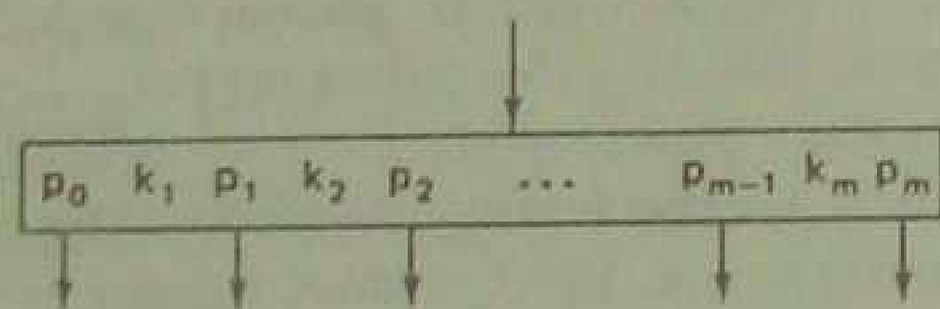
Obr. 4.45 znázorňuje trojúrovňový B-strom druhého rádu. Všetky stránky obsahujú 2, 3 alebo 4 prvky; výnimkou je koreň, ktorý môže obsahovať iba jediný prvok. Všetky listové stránky sú na tretej úrovni.



Obr. 4.45. B-strom druhého rádu

Ak by sme vložením nasledovníka medzi kľúče v stránkach ich predchodcov tento B-strom „stlačili" na jednu úroveň, budú kľúče usporiadané vzostupne, zľava doprava. Takéto usporiadanie predstavuje prirodzené rozšírenie organizácie binárnych vyhľadávacích stromov a súčasne určuje metódu vyhľadávania prvku s daným kľúčom. Majme stránku v tvare znázornenom na *obr. 4.46* a nech má hľadaný prvok kľúč x. Za predpokladu, že stránka bola presunutá do operačnej pamäti, môžeme použiť normálne vyhľadávacie metódy na nájdenie kľúča medzi kľúčmi $k_1, k_2, ..., k_m$. Pre dostatočne veľké m možno použiť i binárne vyhľadávanie; ak je m malé, stačí bežné sekvenčné vyhľadávanie. (Poznamenávame, že čas potrebný na vyhľadávanie v operačnej pamäti je pravdepodobne zanedbateľný v porovnaní s časom, ktorý vyžaduje presun stránky zo sekundárnej do operačnej pamäti.) V prípade neúspešného vyhľadávania sa ocitneme v jednej z troch možných situácií:

1. $k_i < x < k_{i+1}$ pre $1 \leq i < m$. Vyhľadávanie pokračuje na stránke $p_i\uparrow$.
2. $k_m < x$. Vyhľadávanie pokračuje na stránke $p_m\uparrow$.
3. $x < k_1$. Vyhľadávanie pokračuje na stránke $p_0\uparrow$.



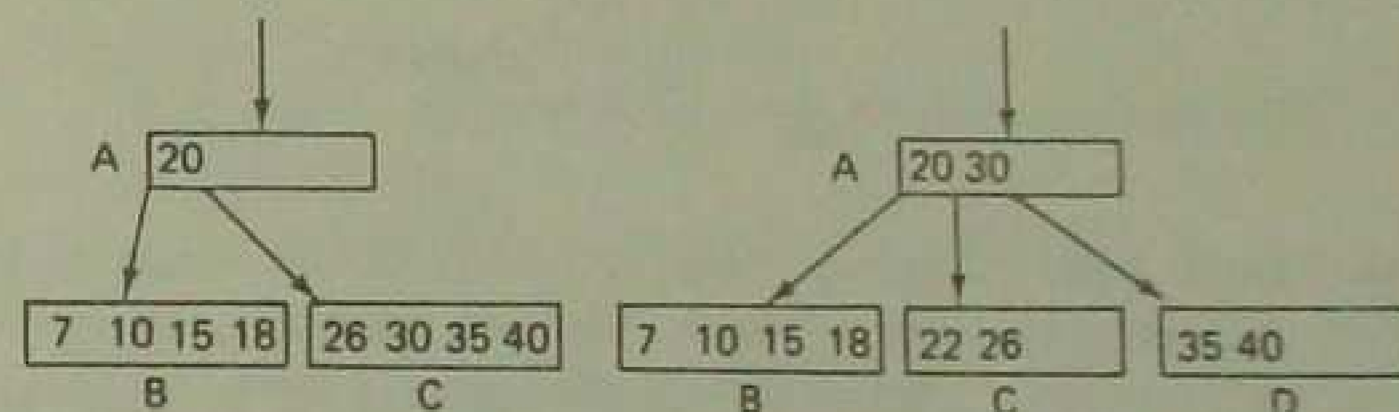
Obr. 4.46. Stránka B-stromu s m kľúčmi

V prípade, že má smerník, určujúci ďalšiu stránku, hodnotu **nil**, znamená to, že neexistuje stránka nasledovníkov, a teda prvok s kľúčom x sa v celom strome nenachádza. Vyhľadávanie je preto ukončené.

Pridávanie do B-stromov je tiež prekvapujúco jednoduché. Ak sa má pridať prvok do stránky obsahujúcej $m < 2n$ prvkov (t. j. do nie celkom plnej stránky), sústredí sa proces pridávania na túto stránku. Iba prípad pridávania prvku do plnej stránky spôsobí určité komplikácie, ktoré ústia do zmeny štruktúry stromu, resp. k vytvoreniu ďalších stránok.

Aby sme lepšie pochopili situáciu, ktorá nastane, pozrime sa na *obr. 4.47*, ktorý ilustruje pridávanie kľúča 22 do B-stromu druhého rádu. Proces pridávania prebieha v týchto krokoch:

1. Zistilo sa, že kľúč 22 chýba; jeho pridávanie do stránky C je nemožné, pretože je plná.
2. Stránka C sa rozdelí na dve stránky (t. j. vytvorí sa nová stránka D).
3. Všetkých $m + 1$ kľúčov sa rovnomerne rozdelí do stránok C a D, pričom prostredný kľúč sa presunie o jednu úroveň vyššie, do stránky predchodcov A.



Obr. 4.47. Pridávanie kľúča 22 do B-stromu

Táto veľmi elegantná schéma zachováva všetky charakteristické vlastnosti B-stromov. Predovšetkým tú, že rozdelené stránky obsahujú presne n prvkov. Pochopiteľne, pridanie prvku do stránky predchodcov môže spôsobiť jej preplnenie, čím sa proces delenia na stránky ďalej zväčšuje. V extrémnom prípade môže delenie pokračovať až ku koreňovej stránke. Toto je súčasne jediný spôsob, ako môže B-strom zväčšovať výšku. B-strom má teda zvláštny spôsob rastu: od listov ku koreňu.

Na základe uvedených stručných opisov pristúpime teraz k detailnej tvorbe programu. Je už zrejmé, že najvhodnejším bude rekurzívny tvar algoritmu, najmä v súvislosti so spätným (vzhľadom na smer vyhľadávania) procesom rozdeľovania. Všeobecná štruktúra programu bude preto podobná algoritmu pridávania do vyvážených stromov, pochopiteľne, okrem niektorých detailov.

Najprv musíme definovať štruktúru stránka. Rozhodneme sa pre reprezentáciu v tvare záznam.

```
type stránka = record m: index;
                      p0: ref;                              (4.81)
                      e: array [1..nn] of prvok
               end
```

pričom

```
const nn = 2*n;
type ref = ↑stránka;
     index = 0..nn
                                                            (4.82)
```

a

```
type prvok = record kľúč: integer;
                    p: ref;
                    počet: integer
             end
```

Zložka *počet* opäť zastupuje ľubovoľné ďalšie informácie, ktoré môžu byť spojené s každým prvkom. Nemajú však žiadny vplyv na skutočné vyhľadávanie. Uvedomme si, že každá stránka predstavuje priestor pre $2n$ prvkov. Zložka m určuje skutočný počet prvkov v stránke. Keď platí $m \geqq n$ (okrem koreňovej stránky), máme zaručené aspoň 50 %-né využitie pamäti.

Algoritmus vyhľadávania a pridávania v B-stromoch je časťou programu 4.7 a je formulovaný ako procedúra vyhľadaj. Jeho hlavná štruktúra je priamočiara, pripomínajúca jednoduché vyhľadávanie v binárnom strome okrem rozhodovania o vetvení, ktoré nie je založené na binárnom výbere. Na druhej strane je však vyhľadávanie v rámci stránky realizované formou binárneho vyhľadávania v poli e.

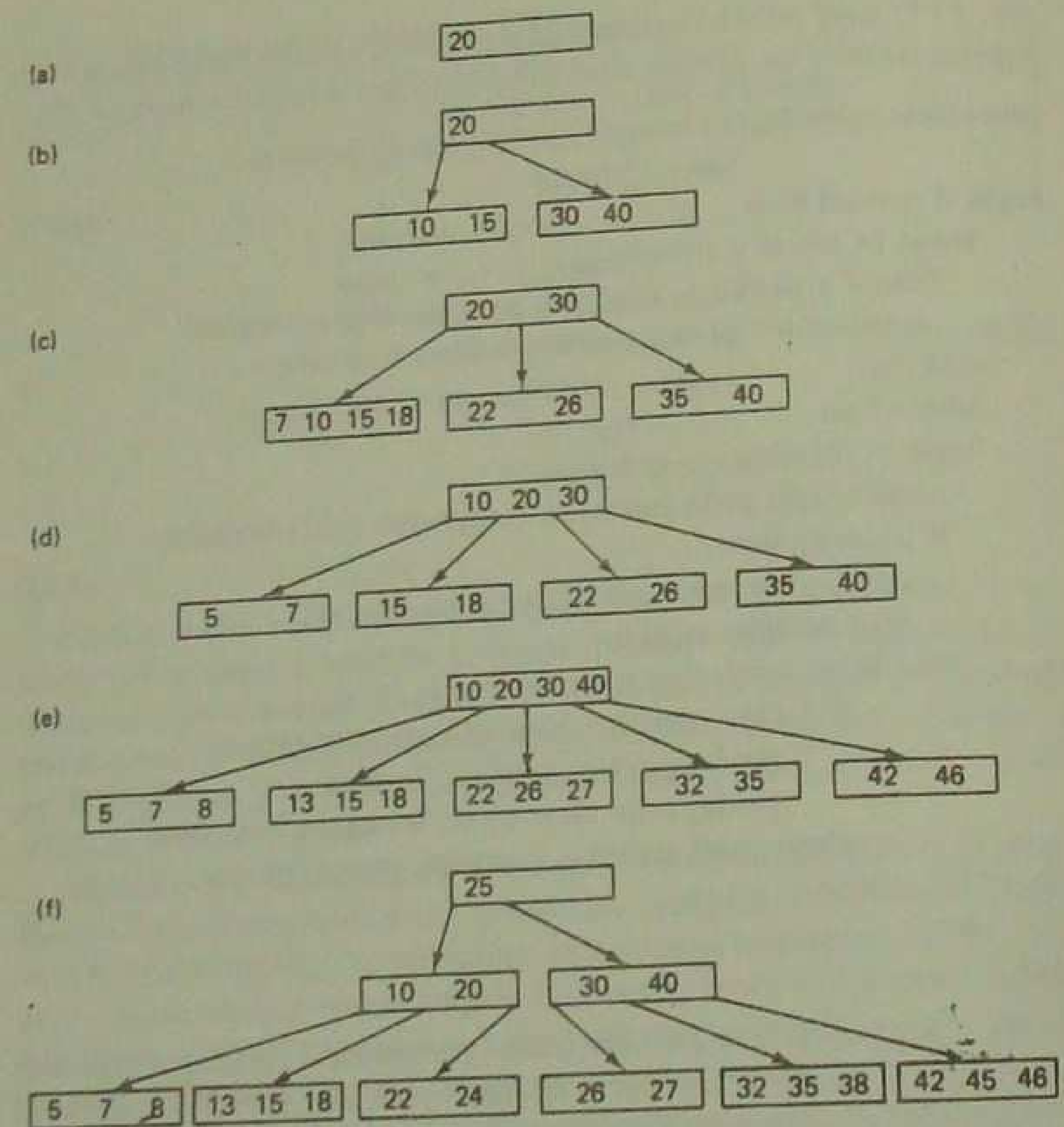
Algoritmus pridávania je kvôli prehľadnosti a zrozumiteľnosti formulovaný ako samostatná procedúra, ktorá sa aktivuje v okamihu, keď procedúra vyhľadaj oznámi „pohyb" prvku smerom nahor, t. j. ku koreňu. Táto indikácia sa realizuje prostredníctvom boolovského parametra h, ktorý má podobnú funkciu ako v prípade algoritmu pridávania do vyváženého stromu, kde oznamoval narastanie podstromu. Ak má parameter h hodnotu true, druhý parameter u reprezentuje prvok „putujúci" nahor. Poznamenávame, že proces pridávania začína v hypotetických stránkach, t. j. v „špeciálnych vrcholoch" znázornených na

obr. 4.19; nový prvok sa pomocou parametra u okamžite premiestni do listovej stránky za účelom skutočného pridania (pozri schému 4.83).

```
procedure vyhľadaj (x: integer; a: ref; var h: boolean;
                    var u: prvok);
begin if a = nil then
   begin {x nie je v strome}
      Priraď x prvku u, nastav h na true; čím sa oznámi
      „putovanie" prvku u smerom hore v strome
   end else
   with a↑ do
   begin {vyhľadaj x v stránke a↑}
      prehľadanie poľa metódou binárneho vyhľadávania;
      if nájdený then                                         (4.83)
         zväčši premennú počet určujúcu výskyt
         relevantného prvku
      else begin vyhľadaj (x, nasledovník, h, u);
            if h then {prvok u postúpil smerom nahor}
               if (počet prvkov v a↑) < 2n then
                  pridaj u do stránky a↑ a nastav h na false
            else rozdeľ stránku a presuň stredný prvok
         end
   end
end
```

Ak má parameter h po aktivovaní procedúry vyhľadaj v hlavnom programe hodnotu true, znamená to, že sa delí koreňová stránka. Pretože táto stránka má význačné postavenie, treba proces jej rozdelenia osobitne naprogramovať. Tento proces bude pozostávať iba z vytvorenia novej (koreňovej) stránky a z vloženia jedného prvku, daného parametrom u, do nej. V dôsledku toho bude nová koreňová stránka obsahovať iba tento jediný prvok. Detaily možno nájsť v programe 4.7. *Obr. 4.48* dokumentuje výsledok programu 4.7, použitého na konštrukciu B-stromu, s nasledujúcou postupnosťou pridávaných kľúčov:

20; 40 10 30 15; 35 7 26 18 22; 5;
42 13 46 27 8 32; 38 24 45 25



Obr. 4.48. Rast B-stromu druhého rádu

Bodkočiarky znamenajú situačné okamihy po každom vzniku novej stránky. Pridanie posledného kľúča spôsobí dve rozdelenia stránok a vznik troch nových stránok.

Všimnime si zvláštny význam príkazu **with** v tomto programe. Jasný je už zo schémy (4.83). V prvom rade určuje, že identifikátory položiek typu stránka sa vnútri príkazu **with** automaticky vzťahujú na stránku $a\uparrow$. Ak by sa stránky v skutočnosti nachádzali v sekundárnej pamäti,

čo je nepochybne potrebné v prípade systémov s veľkými bankami údajov, tak príkaz **with** možno interpretovať aj v zmysle „znamenajúci" presun určenej stránky zo sekundárnej do primárnej pamäti. Pretože každé volanie procedúry vyhľadaj znamená vytvorenie jednej stránky v operačnej pamäti, bude potrebné nanajvýš $k = \log_n N$ jej rekurzívnych volaní. Preto ak bude strom obsahovať N prvkov, musíme vedieť umiestniť k stránok v operačnej pamäti. Toto je jeden obmedzujúci faktor, týkajúci sa veľkosti stránky $2n$. V skutočnosti musíme byť schopní umiestniť v operačnej pamäti viac ako k stránok, pretože proces pridania môže spôsobiť delenie, a tým aj vznik nových stránok. Prirodzeným dôsledkom je trvalé umiestnenie koreňovej stránky v primárnej pamäti, pretože každý dotaz je zahájený bezpodmienečne v koreňovej stránke.

Ďalšou pozitívnou vlastnosťou B-stromovej organizácie je jej vhodnosť a hospodárnosť v prípade výhradne sekvenčnej aktualizácie celej banky údajov. Každá stránka sa presunie iba jediný raz do primárnej pamäti.

Proces odoberania prvkov v B-strome je v podstate priamočiary, ale zložitý v detailoch. Musíme rozlišovať dve rôzne okolnosti:

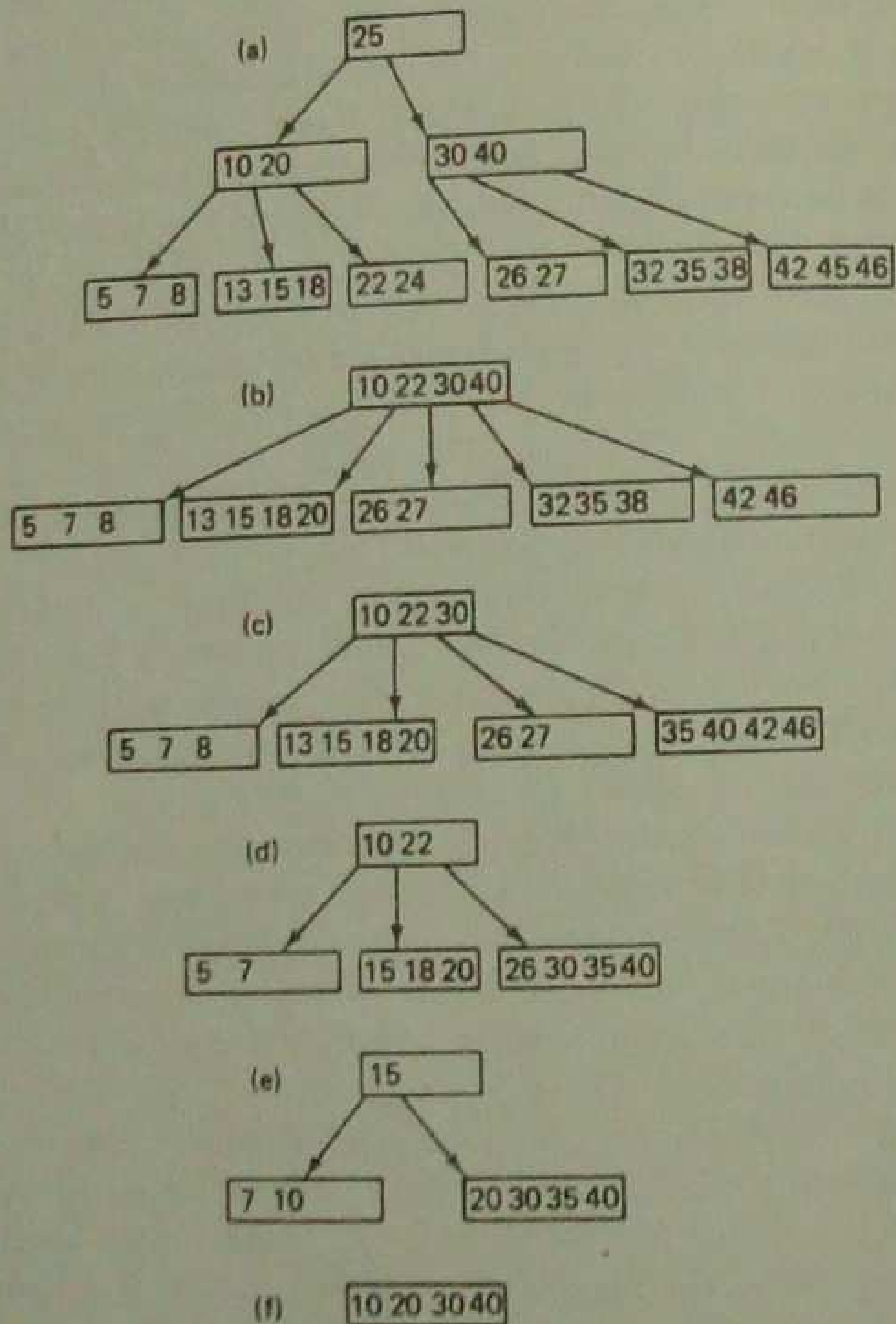
1. Prvok, ktorý chceme odobrať, sa nachádza v listovej stránke; v tomto prípade je algoritmus jeho odobratia jasný a jednoduchý.

2. Prvok nie je v listovej stránke; musí sa nahradiť jedným z dvoch lexikograficky susedných prvkov, ktorý možno jednoducho odobrať, ak sa náhodou nachádza v listovej stránke.

Spôsob hľadania susedného kľúča, spomenutý v druhom prípade, je analogický s metódou použitou v procese odoberania prvkov z binárneho stromu. Preto budeme postupovať takto: zostúpime v smere najpravejších smerníkov až do listovej stránky P, nahradíme prvok, ktorý sa má odobrať, najpravejším prvkom stránky P a potom znížime veľkosť P o jednotku.

V každom prípade musí za znížením veľkosti stránky nasledovať kontrola počtu prvkov m tejto stránky. Pretože ak by nastal prípad $m < n$ (t. j. nezaplnenie stránky), bola by porušená základná charakteristika B-stromov. Takáto situácia sa jednak oznámi prostredníctvom boolovského parametra h a jednak treba uskutočniť istú prídavnú operáciu. Jediným východiskom z núdze je vypožičať si alebo pripojiť

prvok z jednej zo susedných stránok. Pretože táto operácia vyžaduje umiestnenie stránky Q do operačnej pamäti, čo je v podstate drahá operácia, snažíme sa využiť túto neželateľnú situáciu a pripojiť súčasne viac ako jediný prvok. Zaužívanou stratégiou je rozdeľovať prvky rovnomerne do oboch stránok P a Q. Tento spôsob nazývame *vyvažovanie*.



Obr. 4.49. Zánik B-stromu druhého rádu

Prirodzene, môže sa stať, že nebudeme môcť pripojiť žiadny prvok, pretože stránka Q už dosiahla svoju minimálnu veľkosť n. V takomto prípade sa celkový počet prvkov na stránkach P a Q rovná hodnote $2n - 1$; v dôsledku toho môžeme zlúčiť tieto dve stránky do jednej, pričom do nej pridáme prostredný prvok stránky predchodcov (stránok P a Q) a nakoniec sa zbavíme stránky Q. Toto je postup, ktorý je presne opačný ako delenie stránok. Celý proces si môžeme názorne predstaviť, ak uvážime odobratie kľúča 22 v strome na *obr. 4.47*.

Zdôrazňujeme ešte raz, že odstránenie prostredného kľúča zo stránky predchodcov môže spôsobiť zmenšenie veľkosti stránky pod pripustnú hranicu n, čo má za následok ďalšiu špeciálnu operáciu (vyváženie alebo zlučovanie) na nasledujúcej úrovni. V extrémnom prípade sa môže zlučovanie stránok rozšíriť až po koreň. Ak sa veľkosť koreňovej stránky zmenší na nulu, tak sa odstráni, čím dôjde k zmenšeniu výšky B-stromu. Toto je v podstate jediný spôsob zmenšenia výšky B-stromu.

Postupný zánik B-stromu z *obr. 4.48* v dôsledku sekvenčného odoberania kľúčov zobrazuje *obr. 4.49*:

25 45 24; 38 32; 8 27 46 13 42;
5 22 18 26; 7 35 15;

Bodkočiarkami sú opäť naznačené momenty redukcie počtu stránok. Algoritmus rušenia prvkov je obsiahnutý formou procedúry v programe 4.7. Pozoruhodná je najmä podobnosť jeho štruktúry so štruktúrou algoritmu rušenia prvkov vo vyváženom strome.

PROGRAM 4.7. *Vyhľadávanie, pridávanie a rušenie v B-strome*

```
program Bstrom (input, output);
{vyhľadávanie, pridávanie a rušenie v B-strome}
const n = 2; nn = 4;   {veľkosť stránky}
type ref = ↑stránka;
     prvok = record kľúč: integer;
                    p: ref;
                    počet: integer
             end;
     stránka = record m: 0..nn; {počet prvkov}
                      p0: ref;
```

```
                e: array [1..nn] of prvok
           end;
var koreň, q: ref; x: integer;
   h: boolean; u: prvok;
procedure vyhľadaj(x: integer; a: ref; var h: boolean;
                   var v: prvok);
{Vyhľadá kľúč x v B-strome s koreňom a; ak ho nájde, zvýši hodnotu
počítadla, v opačnom prípade pridá prvok s kľúčom x do stromu
a hodnotu počítadla nastaví na 1. Ak treba presunúť prvok na nižšiu
úroveň, priradí ho parametru v; h: = „strom sa zväčšil"}
  var k, y, r: integer; q: ref; u: prvok;
  procedure pridaj;
    var i: integer; b: ref;
  begin {pridaj u do pravej vetvy a↑.e[r]}
    with a↑ do
    begin if m < nn then
      begin m: = m + 1; h: = false;
        for i: = m downto r + 2 do e[i]: = e[i - 1];
        e[r + 1]: = u
      end else
      begin {stránka a↑ je plná; rozdeľ ju a prostredný
            prvok priraď parametru v}
        new(b);
        if r ≦ n then
        begin if r = n then v: = u else
            begin v: = e[n];
              for i: = n downto r + 2 do e[i]: = e[i - 1];
              e[r + 1]: = u
            end;
          for i: = 1 to n do b↑.e[i]: = a↑.e[i + n]
        end else
        begin {pridaj u do pravej stránky} r: = r - n;
            v: = e[n + 1];
          for i: = 1 to r - 1 do b↑.e[i]: = a↑.e[i + n + 1];
          b↑.e[r]: = u;
          for i: = r + 1 to n do b↑.e[i]: = a↑.e[i + n]
```


```
        end;
        m: = n; b↑.m: = n; b↑.p0: = v.p; v.p: = b
      end
    end {with}
  end {pridaj}
begin {vyhľadaj kľúč x v stránke a↑; h = false}
  if a = nil then
  begin {prvok s kľúčom x nie je v strome} h: = true;
    with v do
    begin kľúč: = x; počet: = 1; p: = nil
    end
  end else
  with a↑ do
  begin y: = 1; r: = m; {binárne prehľadanie poľa}
    repeat k: = (y + r) div 2;
      if x ≦ e[k].kľúč then r: = k - 1;
    if x ≧ e[k].kľúč then y: = k + 1;
    until r < y;
    if y - r > 1 then
    begin {nájdený} e[k].počet: = e[k].počet + 1; h: = false
    end else
    begin {prvok nie je v tejto stránke}
      if r = 0 then q: = p0 else q: = e[r].p;
      vyhľadaj(x, q, h, u); if h then pridaj
    end
  end
end {vyhľadaj};
procedure zruš(x: integer; a: ref; var h: boolean);
{vyhľadaj a zruš kľúč x v B-strome a; ak dôjde k nezaplneniu stránky,
vyváži ju so susednou stránkou, ak je to možné, inak ju zlúči so
stránkou b; h: = „stránka a je poddimenzovaná"}
  var i, k, y, r: integer; q: ref;
  procedure nezaplnenie(c, a: ref; s: integer;
                        var h: boolean);
    {a = nezaplnená stránka, c = stránka predchodcov}
    var b: ref; i, k, mb, mc: integer;
```

```
begin mc: = c↑.m; {h = true, a↑.m = n − 1}
  if s < mc then
  begin {b: = stránka vpravo od a} s: = s + 1;
    b: = c↑.e[s].p; mb: = b↑.m; k: = (mb − n + 1) div 2;
    {k = počet prvkov v susednej stránke b}
    a↑.e[n]: = c↑.e[s]; a↑.e[n].p: = b↑.p0;
    if k > 0 then
    begin {presuň k prvkov z b do a}
      for i: = 1 to k − 1 do a↑.e[i + n]: = b↑.e[i];
      c↑.e[s]: = b↑.e[k]; c↑.e[s].p: = b;
      b↑.p0: = b↑.e[k].p; mb: = mb − k;
      for i: = 1 to mb do b↑.e[i]: = b↑.e[i + k];
      b↑.m: = mb; a↑.m: = n − 1 + k; h: = false
    end else
    begin {zlúč stránky a, b}
      for i: = 1 to n do a↑.e[i + n]: = b↑.e[i];
      for i: = s to mc − 1 do c↑.e[i]: = c↑.e[i + 1];
      a↑.m: = nn; c↑.m: = mc − 1; {dispose (b)}
      h: = c↑.m < n
    end
  end else
  begin {b: = stránka vľavo od a}
    if s = 1 then b: = c↑.p0 else b: = c↑.e[s − 1].p;
    mb: = b↑.m + 1; k: = (mb − n) div 2;
    if k > 0 then
    begin {presuň k prvok zo stránky b do a}
      for i: = n − 1 downto 1 do a↑.e[i + k]: = a↑.e[i];
      a↑.e[k]: = c↑.e[s]; a↑.e[k].p: = a↑.p0; mb: = mb − k;
      for i: = k − 1 downto 1 do a↑.e[i]: = b↑.e[i + mb];
      a↑.p0: = b↑.e[mb].p;
      c↑.e[s]: = b↑.e[mb]; c↑.e[s].p: = a;
      b↑.m: = mb − 1; a↑.m: = n − 1 + k; h: = false
    end else
    begin {zlúč stránky a, b}
      b↑.e[mb]: = c↑.e[s]; b↑.e[mb].p: = a↑.p0;
      for i: = 1 to n − 1 do b↑.e[i + mb]: = a↑.e[i];
```



```
      b↑.m: = nn; c↑.m: = mc − 1; {dispose (a)}
      h: = c↑.m < n
    end
  end
end {nezaplnenie};
procedure del (p: ref; var h: boolean);
  var q: ref; {a, k sú globálne}
  begin
    with p↑ do
    begin q: = e[m].p;
      if q ≠ nil then
      begin del (q, h); if h then nezaplnenie (p, q, m, h)
      end else
      begin p↑.e[m].p: = a↑.e[k].p; a↑.e[k]: = p↑.e[m];
        m: = m − 1; h: = m < n
      end
    end
  end {del};
begin {zruš}
  if a = nil then
  begin writeln ('KĽÚČ NIE JE V STROME'); h: = false
  end else
  with a↑ do
  begin y: = 1; r: = m; {binárne prehľadanie poľa}
    repeat k: = (y + r) div 2;
      if x ≦ e[k].kľúč then r: = k − 1;
      if x ≧ e[k].kľúč then y: = k + 1;
    until y > r;
    if r = 0 then q: = p0 else q: = e[r].p;
    if y − r > 1 then
    begin {nájdený, zruš e[k]}
      if q = nil then
      begin {a je koncová stránka} m: = m − 1; h: = m < n;
        for i: = k to m do e[i]: = e[i + 1];
      end else
      begin del (q, h); if h then nezaplnenie (a, q, r, h)
```

```
          end
        end else
        begin zruš (x, q, h); if h then nezaplnenie (a, q, r, h)
        end
      end
    end {zruš};
    procedure vytlačstrom (p: ref; y: integer);
      var i: integer;
    begin if p ≠ nil then
      with p↑ do
      begin for i: = 1 to y do write (' ');
        for i: = 1 to m do write (e[i].kľúč: 4); writeln;
        vytlačstrom (p0, y + 1);
        for i: = 1 to m do vytlačstrom (e[i].p, y + 1)
      end
    end;
    begin koreň: = nil; read (x);
      while x ≠ 0 do
      begin writeln ('HĽADANÝ KĽÚČ', x); vyhľadaj (x, koreň, h, u);
        if h then
        begin {vytvor a pripoj novú stránku} q: = koreň;
                                             new (koreň);
          with koreň↑ do begin m: = 1; p0: = q; e[1]: = u end
        end; vytlačstrom (koreň, 1); read (x)
      end;
      read (x);
      while x ≠ 0 do
      begin writeln ('ODOBERANÝ KĽÚČ', x);
        zruš (x, koreň, h);
        if h then
        begin {veľkosť základnej stránky sa zmenšila}
          if koreň↑.m = 0 then
          begin q: = koreň; koreň: = q↑.p0; {dispose (q)}
          end
        end;
        vytlačstrom (koreň, 1); read (x)
      end
    end.
```

Rozsiahlu analýzu účinnosti B-stromov uskutočnili R. BAYER a E. MC CREIGHT a nachádza sa v článkoch [4-2] až [4-4]. Autori sa venujú predovšetkým otázke optimálnej veľkosti stránky n, ktorá veľmi závisí od charakteristík dostupnej pamäti a počítačového systému.

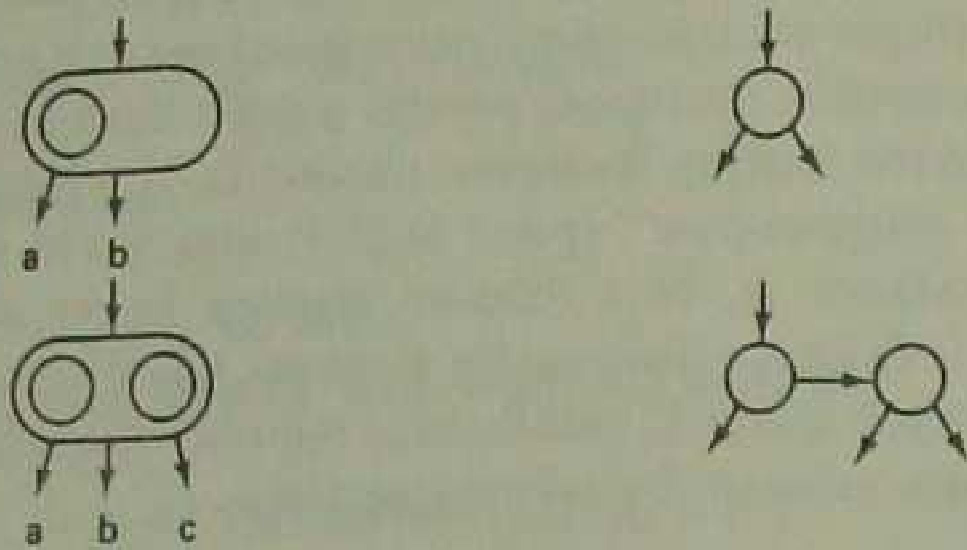
Niekoľko variácií schémy B-stromu podáva D. E. KNUTH vo svojej knihe „Umenie programovať" (pozri aj [2-7], strany 476 až 479). Významným poznatkom je, že s delením stránok by sa malo počkať podobne, ako sa čaká so zlučovaním stránok, t.j. kým sa nevyskúša vyváženie susedných stránok. Odhliadnuc od toho ukazuje sa, že navrhované vylepšenia prinesú iba minimálne zisky.

4.5.2 BINÁRNE B-STROMY

Najmenej zaujímavými druhmi B-stromov sa zdajú byť B-stromy prvého rádu ($n = 1$). Niekedy sa im však oplatí venovať pozornosť. Je pochopiteľné, že B-stromy prvého rádu nie sú vhodné na reprezentáciu veľkých, usporiadaných, indexovaných množín údajov, ktoré vyžadujú sekundárne pamäti; približne 50 % všetkých stránok by obsahovalo iba jediný prvok. Zabudnime preto na sekundárne pamäti a venujme sa problémom vyhľadávacích stromov opäť v súvislosti s jednoúrovňovou pamäťou.

Binárny B-strom (BB-strom) sa skladá z vrcholov (stránok) s jedným alebo dvoma prvkami. Preto stránka obsahuje buď dva alebo tri smerníky na nasledovníkov; z tejto skutočnosti vznikol termín 2—3 strom. V súlade s definíciou B-stromov budú všetky listové stránky na tej istej úrovni a všetky nelistové stránky BB-stromov budú mať dvoch alebo troch nasledovníkov (vrátane koreňa). Pretože teraz berieme do úvahy iba primárnu pamäť, bude veľmi dôležité optimálne využitie pamäťového priestoru. Preto je jasné, že reprezentácia jednotlivých prvkov vrcholu pomocou poľa nebude vyhovujúca. Vhodnejšou bude druhá alternatíva, dynamické prideľovanie pamäti, t.j. každý vrchol bude

obsahovať zoznam prvkov dĺžky 1 alebo 2. Pretože každý vrchol má maximálne troch nasledovníkov, a teda musí byť schopný uchovávať až tri smerníky, pokúsime sa skombinovať smerníky na nasledovníkov so smerníkmi realizujúcimi zoznamy prvkov, ako vidno na *obr. 4.50*.



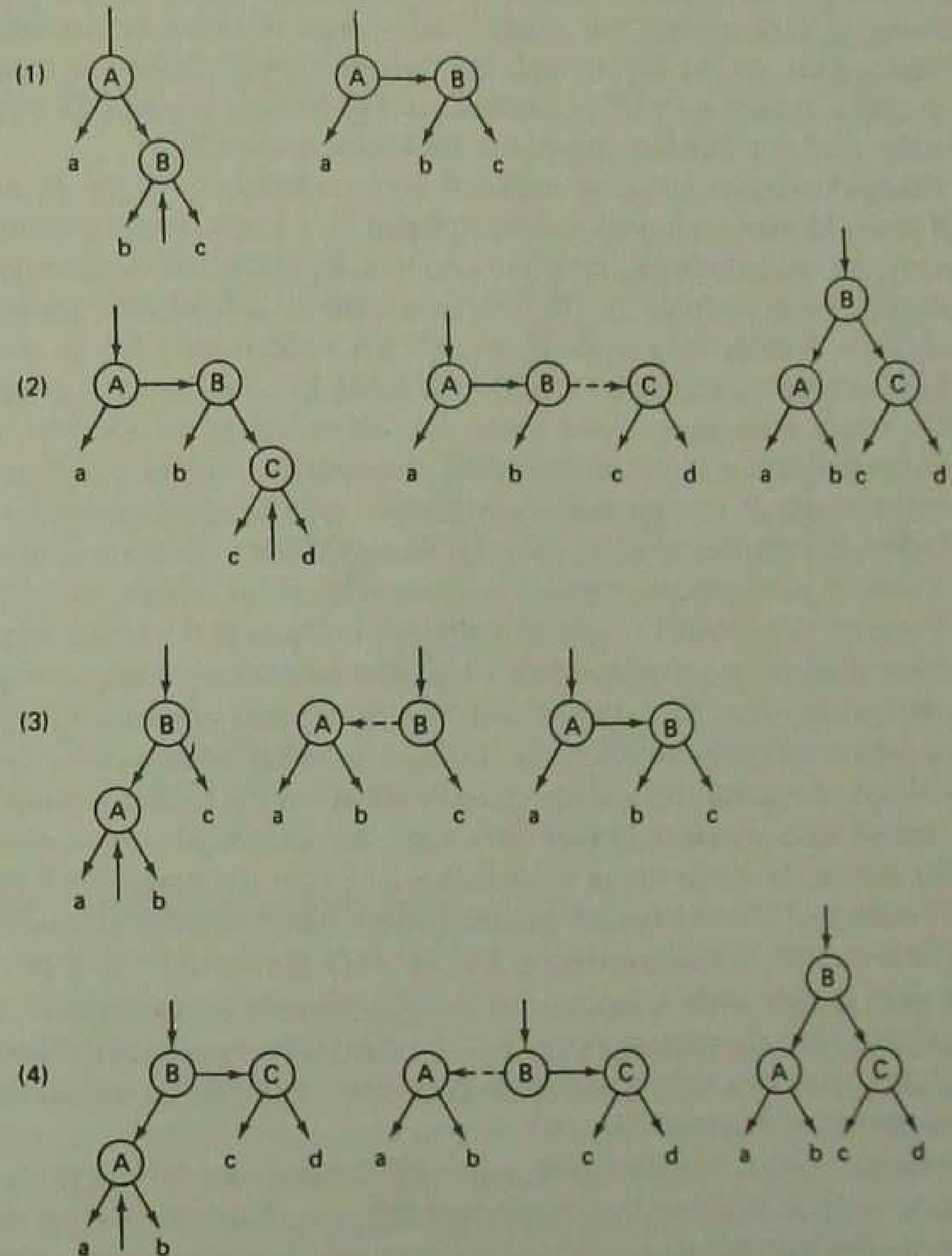
Obr. 4.50. Reprezentácia vrcholov BB-stromu

Vrchol B-stromu týmto stráca svoju skutočnú identitu a prvky preberajú úlohu vrcholov normálneho binárneho stromu. Treba však rozlišovať smerníky na priamych nasledovníkov (t. j. zvislých nasledovníkov) od smerníkov na nevlastných nasledovníkov (t. j. horizontálnych, čiže vyskytujúcich sa na tej istej stránke). Vzhľadom na to, že iba smerníky doprava môžu byť horizontálne, stačí na zaznamenanie tohto rozdielu jediný bit. Zavedieme preto boolovskú zložku h s významom „horizontálny". Definícia typu vrchol, založená na takejto reprezentácii, je vyjadrená schémou (4.84). Navrhol a preskúmal ju R. BAYER [4-3] v roku 1971. Táto definícia reprezentuje organizáciu stromového vyhľadávania, ktorá zaručuje maximálnu dĺžku cesty $p = 2 . \lceil \log N \rceil$.

```
type vrchol = record kľúč: integer;
                     ................
                     ľavý, pravý: ref;          (4.84)
                     h: boolean;
              end
```

Keď sa teraz bližšie pozrieme na problém pridávania kľúčov, zistíme, že vzhľadom na možný rast ľavého a pravého podstromu môžu nastať štyri rôzne situácie, ktoré sú znázornené na *obr. 4.51*. Uvedomme si, že

charakteristickou vlastnosťou B-stromov je ich rast od listov ku koreňu. Navyše treba zachovávať i ďalšiu vlastnosť, ktorou je rovnaká úroveň všetkých listov stromu.



Obr. 4.51. Pridanie vrcholov do BB-stromu

(4.63). Pochopiteľne, schéma zarovnaného stromu predstavuje alternatívu kritéria vyváženosti AVL-stromov. Preto by bolo prospešné a zvládnuteľné porovnanie výkonnosti oboch algoritmov.

My sa však zdržíme matematických analýz a sústredíme našu pozornosť iba na niektoré zásadné rozdiely. Dá sa dokázať, že vyvážené AVL-stromy predstavujú podmnožinu zarovnaných stromov. Preto je trieda zarovnaných stromov väčšia. Z toho vyplýva, že ich dĺžka cesty je v priemere väčšia ako pri AVL-stromoch. V tejto súvislosti si všimnime najhorší prípad, t. j. strom (4) na *obr. 4.53*. Na druhej strane preusporiadanie vrcholov nebude ani zďaleka tak časté. Vyvážený strom budeme preto uprednostňovať v tých prípadoch, ktoré vyžadujú oveľa viac výberov kľúčov ako pridávaní (alebo rušení); v prípade, že by tento pomer bol 1 : 1, uprednostňujeme zarovnaný strom.

Ťažko povedať, kde je hranica. Pri jej určovaní však musíme brať do úvahy nielen pomer medzi frekvenciami výberu a zmenou štruktúry, ale i charakteristiky implementácie. Ide predovšetkým o prípad veľmi zhustenej záznamovej štruktúry reprezentujúcej typ vrchol, v dôsledku čoho prístup k prvkom vrcholu vyžaduje náročné manipulácie s časťami strojových slov. Boolovské zložky (*lh* a *rh* v prípade zarovnaných stromov) sa dajú implementovať oveľa efektívnejšie ako zložky, ktoré nadobúdajú tri hodnoty (ako v prípade zložky *bal* pri vyvážených stromoch).

4.6 TRANSFORMÁCIE KĽÚČA

Hlavný problém, s ktorým sa budeme zaoberať v poslednej časti tejto kapitoly a ktorý použijeme pri hľadaní riešení, spojených s dynamickým prideľovaním pamäti, je takýto:

Máme danú množinu S prvkov charakterizovaných hodnotami kľúčov, medzi ktorými je definovaná relácia usporiadania. Úlohou je nájsť takú organizáciu množiny S, aby výber prvku s daným kľúčom vyžadoval najmenšiu možnú námahu.

Prirodzene, každý prvok v pamäti počítača je prístupný prostredníctvom svojej adresy a. Teda problém je nájsť vhodné zobrazenie H kľúčov (K) do pamäťových adries (A):

$$H : K \rightarrow A$$

V článku 4.5 bolo toto zobrazenie implementované rôznymi vyhľadávacími algoritmami v štruktúrach zoznam a strom, založených na rôznych organizáciách údajov. V tomto článku uvedieme ešte jeden spôsob, ktorý je v podstate veľmi jednoduchý a v mnohých prípadoch veľmi efektívny. To, že má i svoje nevýhody, rozoberieme neskôr.

Organizáciu údajov, ktorá sa používa pri tejto technike, predstavuje štruktúra pole. H je potom zobrazenie, ktorým sa kľúče transformujú do indexov poľa, z čoho vznikol pre túto techniku i pojem *transformácia kľúča*. Poznamenávame, že vzhľadom na statickú povahu štruktúry pole nebudeme potrebovať procedúry dynamického prideľovania pamäti. Z toho, samozrejme, vyplýva, že táto problematika by nemala byť zaradená do tejto kapitoly, zaoberajúcej sa problémami dynamických informačných štruktúr. Napriek tomu si myslíme, že vzhľadom na problémové oblasti, v ktorých je použiteľnosť transformácie kľúča dôstojným súperom dynamických stromových štruktúr, je táto kapitola vhodným miestom na jej prezentáciu.

Základným nedostatkom používania techniky transformácie kľúča je, že množina možných hodnôt kľúčov je oveľa väčšia ako množina dostupných adries pamäti (indexov poľa). Typickým príkladom je použitie slov (ktoré pozostávajú až z desiatich písmen) ako kľúčov na identifikovanie jednotlivcov v množine, ktorá obsahuje, povedzme, do tisíc osôb. Jednoduchým výpočtom zistíme, že existuje 26^{10} možných kľúčov, ktoré treba zobraziť do 10^3 možných indexov. Funkcia H je preto zvyčajne funkciou, zobrazujúcou n objektov do jedného. Ak máme daný kľúč k, tak prvým krokom v operácii výberu (vyhľadávania) je výpočet príslušného indexu $h = H(k)$. Druhým a, prirodzene, dôležitým krokom, je overiť, či prvok s kľúčom k je skutočne identifikovaný indexom h v poli (tabuľke) T, t. j. či $T[H(k)].kľúč = k$. Prirodzeným dôsledkom tohto sú dve otázky:

1. Aký druh funkcie H máme použiť?
2. Čo robiť v prípade, ak funkcia H vypočíta index, ktorý neurčí správnu lokalitu požadovaného prvku?

Odpoveď na druhú otázku znie: treba použiť nejakú metódu, ktorá určí alternatívnu lokalitu prvku, povedzme index h', a ak ani toto ešte nie je správne miesto požadovaného prvku, tretí index h'' atď. Ak sa v identifikovanom mieste pamäti nachádza iný kľúč ako požadovaný,

dôjde k situácii, ktorú nazývame *kolízia*; úloha generovania alternatívnych indexov sa nazýva *ošetrovanie kolízií*. V nasledujúcich odsekoch sa budeme zaoberať výberom transformačnej funkcie a metódami ošetrovania kolízií.

4.6.1 VOĽBA TRANSFORMAČNEJ FUNKCIE

Základnou požiadavkou dobrej transformačnej funkcie je, aby rozmiestňovala kľúče, pokiaľ možno čo najrovnomernejšie, v rámci rozsahu hodnôt indexov. Nehľadiac na splnenie tejto požiadavky, distribúcia nemá byť viazaná na nijakú schému a je skutočne žiadúce, aby sme mali dojem, že je úplne náhodná. Na základe tejto vlastnosti sa dospelo k trochu nevedeckému pomenovaniu tejto metódy — „*hašovanie*", t. j. „posekanie" alebo „pomletie na kašu" — a transformačná funkcia získala názov „*hašovacia funkcia*"[1]. Pochopiteľne, mala by byť efektívne vypočítateľná, t. j. realizovaná prostredníctvom minimálneho počtu základných aritmetických operácií.

Predpokladajme, že máme k dispozícii funkciu ord (k), ktorá určuje poradové číslo kľúča k v množine všetkých možných kľúčov. Predpokladajme ďalej, že i je index poľa a môže nadobúdať celočíselné hodnoty v rozsahu $0 \ldots N-1$, pričom N určuje veľkosť poľa. Potom prirodzenou voľbou transformačnej funkcie je

$$H(k) = \text{ord}(k) \ \mathbf{mod} \ N \tag{4.88}$$

Vlastnosťou tejto funkcie je, že hodnoty kľúčov rozdeľuje rovnomerne v rámci rozsahu indexov, a je preto základom väčšiny transformácií kľúčov. V prípade, že by N bolo mocninou dvojky, je i mimoriadne efektívne vypočítateľná. Toto je však práve prípad, ktorému sa musíme vyhnúť, ak sú kľúče tvorené postupnosťami písmen. Predpoklad, že

---

[1] Slovenskí programátori vedú vleklé spory so slovenskými jazykovedcami, ako najvhodnejšie prekladať anglický termín *hashing* a všetky termíny z neho odvodené. Pretože však v slovenčine slovo s podobným základom a významom už existuje, a to hašé (a znamená sekané alebo mleté mäso), nevidíme dôvod, prečo by sme ho nemohli slobodne používať aj my, keď kuchári môžu.... Navyše v angličtine vzniklo toto slovo podobnou asociáciou (pozri KNUTH [2-7] a WIEDERMANN [4-12]) — pozn. prekl.

všetky kľúče sú rovnako pravdepodobné, je v tomto prípade úplne nesprávny. V skutočnosti sa slová, ktoré sa odlišujú iba minimálnym počtom znakov, zobrazujú s veľkou pravdepodobnosťou do identických indexov, čím dochádza k veľmi nerovnomernej distribúcii. Odporúča sa preto, aby N vo vzťahu (4.88) bolo prvočíslo [4-7]. To má za následok potrebu úplnej operácie delenia, namiesto maskovania binárnych čísel, čo však nevadí, pretože väčšina moderných počítačov má vo svojom inštrukčnom repertoári zabudovanú aj inštrukciu delenia.

Často sa transformačné funkcie, ktoré využívajú vlastnosti logických operácií, ako napr. operácie vzájomného vylúčenia (XOR), používajú pre tie časti kľúčov, ktoré sú reprezentované postupnosťou binárnych číslic. Tieto operácie zvyknú byť na niektorých počítačoch rýchlejšie ako delenie, ale niekedy zlyhávajú pri rovnomernej distribúcii kľúčov v rámci rozsahu indexov. Preto tieto metódy ďalej nebudeme podrobnejšie rozoberať.

4.6.2 OŠETROVANIE KOLÍZIÍ

V prípade, ak prvok tabuľky zodpovedajúci danému kľúču nie je hľadaným prvkom, dochádza ku kolízii, t.j. kľúče dvoch rôznych prvkov sú zobrazené do toho istého indexu. V takomto prípade potrebujeme ďalší pokus, skúmajúci index, ktorý získame deterministickým spôsobom z daného kľúča. Existuje viacero spôsobov vytvorenia sekundárnych indexov. Zvyčajným a efektívnym spôsobom je pospájanie všetkých prvkov s rovnakými kľúčmi $H(k)$ do zoznamov. Tejto metóde hovoríme *priame zreťazenie*. Prvky tohto zoznamu môžu byť buď v primárnej pamäti, alebo v oblasti preplnenia. Táto metóda je pomerne efektívna, aj keď má nevýhodu v tom, že treba udržiavať sekundárny zoznam, čo pre každý prvok znamená prídavné miesto v pamäti pre smerník (alebo index) na zoznam konfliktných prvkov.

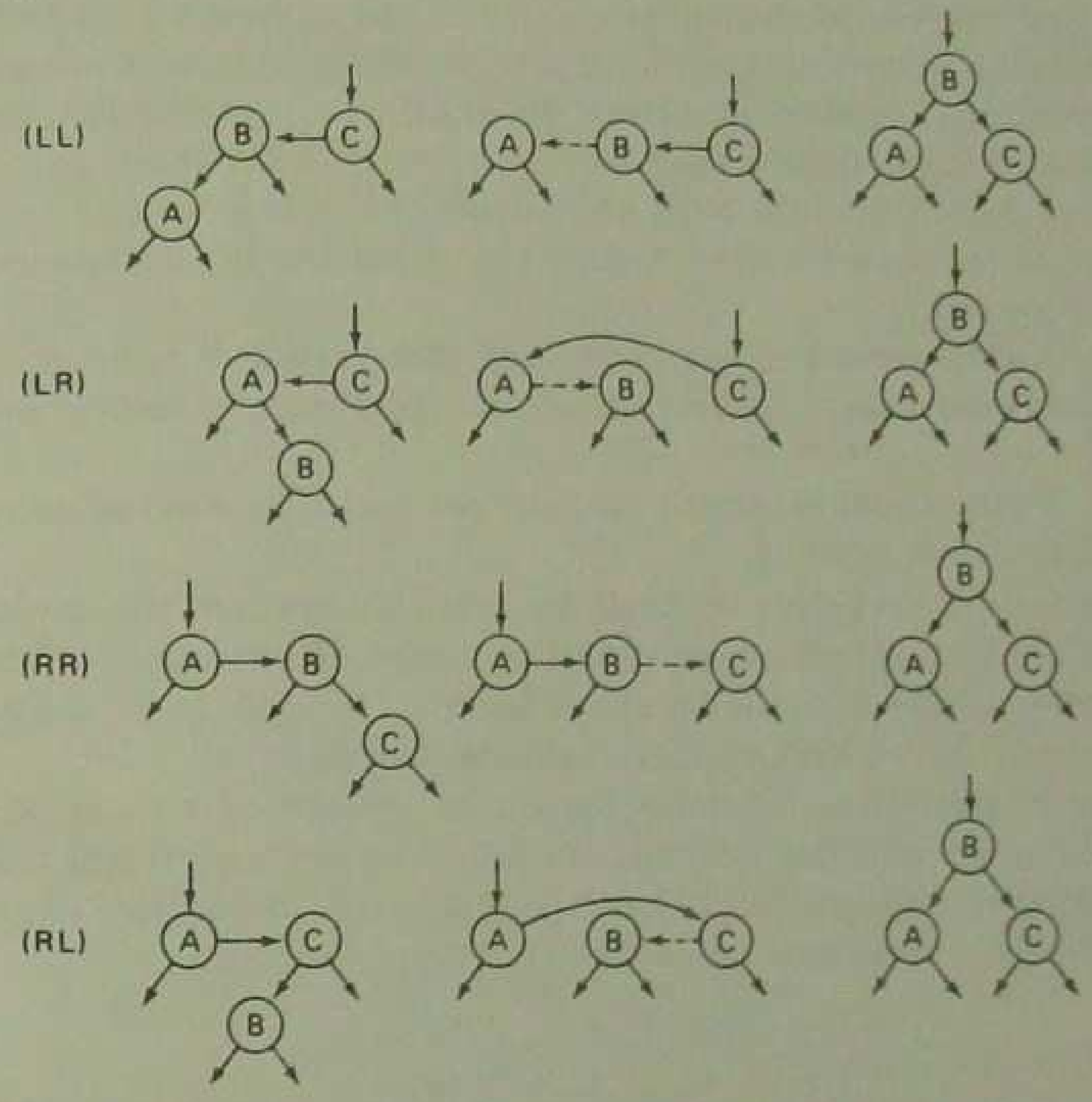
Alternatívne riešenie kolíznych situácií upúšťa od použitia zoznamov a vychádza z nasledujúcej stratégie: Tabuľka prvkov sa prehľadáva dovtedy, kým sa nenájde hľadaný prvok, alebo kým sa nevyskytuje „otvorená pozícia" v tabuľke, čo znamená, že sa hľadaný kľúč v tabuľke nenachádza. Táto metóda sa nazýva *otvorená adresácia* [4-9]. Prirodzene, postupnosť indexov sekundárnych pokusov musí byť pre daný

Najjednoduchší je prípad (1), keď rastie pravý podstrom vrcholu A, ktorý je jediným kľúčom hypotetickej stránky. Potom sa nasledovník B stane nevlastným nasledovníkom vrcholu A, t. j. zvislý smerník sa stane vodorovným smerníkom. Takéto jednoduché „zdvihnutie" pravého ramena je však možné len vtedy, keď vrchol A nemá nevlastného nasledovníka. Ak by ho už mal, tak by sme dostali stránku s tromi vrcholmi a museli by sme ju rozdeliť na dve stránky (prípad 2). Prostredný vrchol B stránky prechádza na vyššiu úroveň.

Predpokladajme teraz, že narástol ľavý podstrom vrcholu B. Ak je B opäť jediným vrcholom stránky (prípad 3), t. j. jeho pravý smerník ukazuje na nasledovníka, ľavý podstrom A sa môže stať nevlastným nasledovníkom vrcholu B. (Potrebná je pritom jednoduchá rotácia smerníkov, pretože ľavý smerník nemôže byť vodorovný.) Ak by však vrchol B už mal nevlastného nasledovníka, tak by „zdvihnutie" A spôsobilo vznik stránky s tromi vrcholmi, čo vyžaduje jej rozdelenie. Rozdelenie stránky sa uskutoční úplne priamočiaro: vrchol C sa stane nasledovníkom B, ktorý prejde na najbližšiu vyššiu úroveň (prípad 4).

Treba však poznamenať, že pri vyhľadávaní kľúča v podstate nezáleží na tom, či postupujeme v smere vodorovného alebo zvislého smerníka. Preto sú aj starosti s ľavým smerníkom, ktorý sa stal vodorovným v treťom prípade, neopodstatnené, i keď jeho stránka neobsahuje viac ako dva podstromy. Skutočne, algoritmus pridávania odhaľuje nezvyčajnú asymetriu spracúvania rastu ľavých a pravých podstromov, čím potvrdzuje neopodstatnenosť organizácie BB-stromov. Dôkaz neopodstatnenosti tejto organizácie však neexistuje; na základe zdravej intuície predsa cítime, že niečo nie je v poriadku a že túto asymetriu by bolo dobré odstrániť. To nás vedie k pojmu symetrického binárneho B-stromu (SBB-strom), preskúmaného aj BAYEROM [4-4] v roku 1972. V priemere tieto stromy vedú k trocha efektívnejšiemu vyhľadávaniu hoci za cenu zložitejších algoritmov pridávania a rušenia. Navyše každý vrchol vyžaduje ďalšie dva bity (boolovské premenné *lh* a *rh*) na označenie charakteru svojich dvoch smerníkov.

Vzhľadom na to, že naše ďalšie úvahy obmedzíme na problém pridávania, vráťme sa ešte raz k spomenutým štyrom prípadom rastu podstromov. Znázornené sú na *obr. 4.52*, ktorý súčasne zvýrazňuje nadobudnutú symetriu. Uvedomme si, že len čo narastie podstrom vrcholu A, ktorý nemá nevlastných nasledovníkov, stane sa koreň tohto podstromu nevlastným nasledovníkom vrcholu A. Tento prípad nevyžaduje ďalšie úvahy.



Obr. 4.52. Pridávanie do SBB-stromov

Všetky štyri prípady znázornené na *obr. 4.52* dokumentujú vznik preplnenia stránky, a teda i jej patričné rozdelenie. Označené sú podľa smerov vodorovných smerníkov, spájajúcich troch nevlastných nasledovníkov (pozri prostredné podoby stromov na *obr. 4.52*). Začiatočná situácia je znázornená ľavým stĺpcom; stredný stĺpec dokumentuje skutočnosť, že spodný vrchol postúpil o jednu úroveň vyššie v dôsledku rastu jeho podstromu; pravý stĺpec zobrazuje výsledok preusporiadania vrcholov (situáciu po rozdelení stránky).

Odporúča sa, aby sme v ďalšom texte už upustili od termínu stránka, aj keď sa táto stromová organizácia vyvinula práve pomocou nej, pretože teraz sú už všetky obmedzené maximálnou dĺžkou cesty do 2 . log N. Jediné, čo musíme zabezpečiť, je, aby sa pozdĺž ktorejkoľvek vyhľadávacej cesty nevyskytli dva za sebou idúce vodorovné smerníky. Pochopiteľne, to však neznamená, že zakážeme výskyt vrcholov, ktoré by mali vodorovné smerníky doprava i doľava. SBB-stromy môžeme potom definovať ako stromy, ktoré musia spĺňať tieto kritériá:

1. Každý vrchol obsahuje jeden kľúč a maximálne dva (smerníky na) podstromy.

2. Každý smerník je buď vodorovný, alebo zvislý. V rámci žiadnej vyhľadávacej cesty sa nevyskytujú dva za sebou idúce vodorovné smerníky.

3. Všetky koncové vrcholy (vrcholy bez nasledovníkov) sú na rovnakej (koncovej) úrovni.

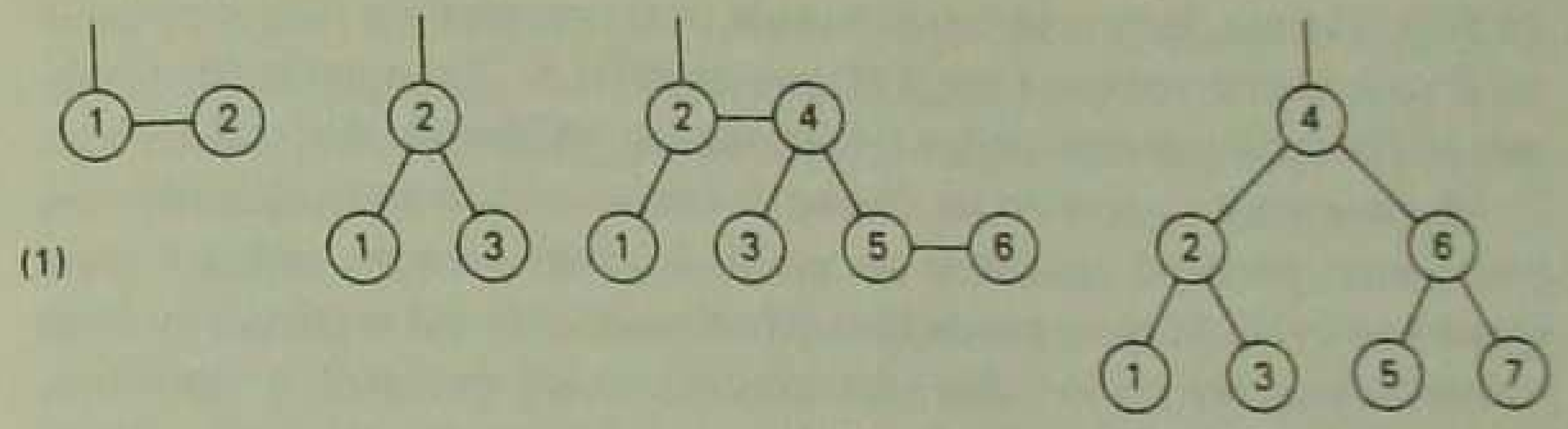
Z tejto definície vyplýva, že najdlhšia vyhľadávacia cesta nie je väčšia ako dvojnásobok výšky stromu. Pretože žiadny SBB-strom s N vrcholmi nemôže mať výšku väčšiu ako $\lceil \log N \rceil$, je jasné, že $2 \lceil \log N \rceil$ predstavuje hornú hranicu dĺžky vyhľadávacej cesty.

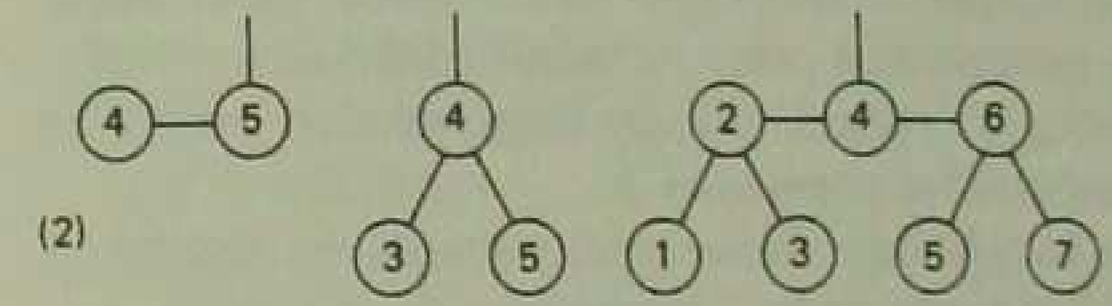
Proces narastania takéhoto stromu je znázornený na *obr. 4.53*. Riadky tohto obrázka vystihujú situáciu v strome po pridaní kľúča z nasledujúcej postupnosti kľúčov (bodkočiarkou sú oddelené okamihy, ktoré sú znázornené na *obr. 4.53*):

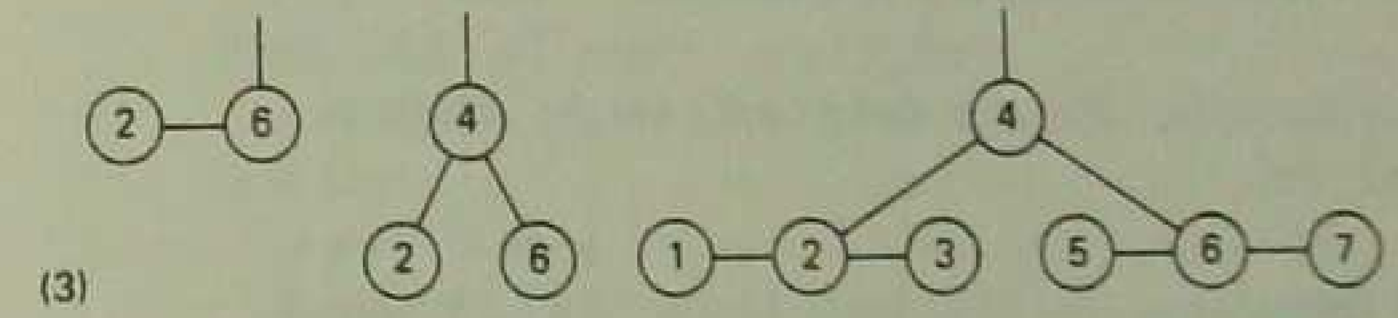
$$
\begin{array}{ll}
(1) & 1\ 2;\ 3;\ 4\ 5\ \ 6;\ 7; \\
(2) & 5\ 4;\ 3;\ 1\ 2\ \ 7\ \ 6; \\
(3) & 6\ 2;\ 4;\ 1\ 7\ \ 3\ \ 5; \\
(4) & 4\ 2\ \ 6;\ 1\ 7;\ 3\ \ 5;
\end{array}
\qquad (4.85)
$$

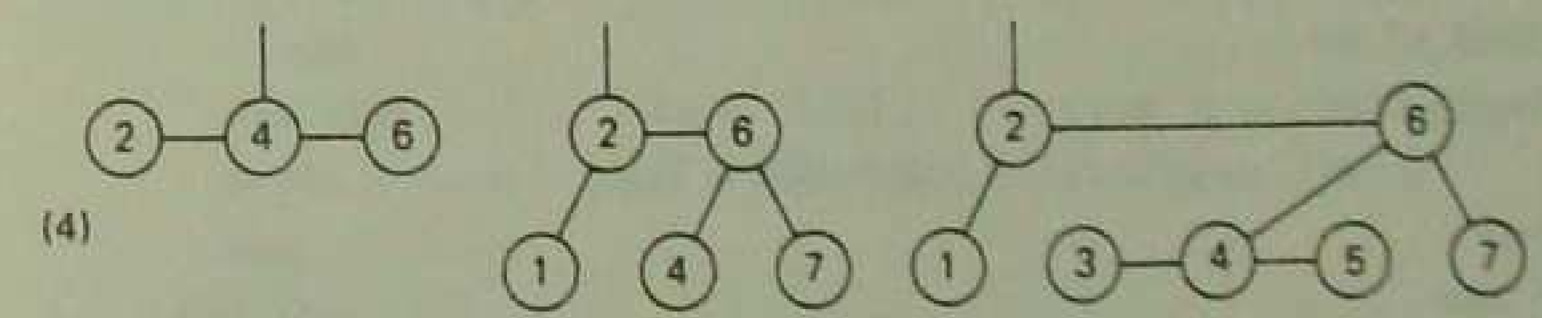
Tieto obrazce zvýrazňujú najmä tretiu vlastnosť B-stromov: všetky koncové vrcholy sú na rovnakej úrovni. Tieto štruktúry nás nabádajú k ich porovnaniu so záhradnými živými plotmi, nedávno pristrihnutými záhradníckymi nožnicami. Budeme ich preto nazývať *zarovnané stromy*.

Konštrukciu takýchto stromov opisuje algoritmus (4.87). Je založený na definícii typu vrchol (4.86), obsahujúceho dve zložky (*lh* a *rh*), ktoré vyjadrujú to, že ľavé a pravé smerníky sú vodorovné.









Obr. 4.53. Vývoj zarovnaných stromov prostredníctvom pridávania kľúčov z postupnosti (4.85)

```
type vrchol = record kľúč: integer;
                     počet: integer;
                     ľavý, pravý: ref;          (4.86)
                     lh, lr: boolean
              end
```

Rekurzívna procedúra vyhľadaj je opäť vytvorená na základe princípov algoritmu pridávania prvkov do binárneho stromu (pozri schému

(4.87)). Vidíme, že procedúra obsahuje tretí parameter h, ktorý označuje, či sa podstrom s koreňom p zmenil alebo nie. Tento parameter priamo zodpovedá parametru h v programe pre vyhľadávanie v B-strome.

Musíme však upozorniť na dôsledok reprezentácie stránok pomocou zoznamov: prechod stránkou sa uskutoční buď jedným alebo dvoma volaniami vyhľadávacej procedúry. Musíme rozlišovať medzi prípadom podstromu (určeným zvislým smerníkom), ktorý narástol, a vrcholom nevlastného potomka (určeným vodorovným smerníkom), ku ktorému pribudol ďalší nevlastný potomok, a preto vyžaduje rozdelenie stránky. Problém môžeme jednoducho vyriešiť tak, že stanovíme tri prípustné hodnoty, ktoré môže nadobúdať premenná h:

1. $h = 0$: podstrom p nevyžaduje žiadne zmeny stromovej štruktúry.
2. $h = 1$: vrchol p získal nevlastného potomka.
3. $h = 2$: výška podstromu p sa zväčšila.

```
procedure vyhľadaj (x: integer; var p: ref; var h: integer);
  var p1, p2: ref;
begin
  if p = nil then
  begin {slovo nie je v strome; pridaj ho}
    new (p); h: = 2;
    with p↑ do
    begin kľúč: = x; počet: = 1; ľavý: = nil;
          pravý: = nil; lh: = false; rh: = false
    end
  end else
  if x < p↑.kľúč then
  begin vyhľadaj (x, p↑.ľavý, h);
    if h ≠ 0 then
    if p↑.lh then
    begin p1: = p↑.ľavý; h: = 2; p↑.lh: = false;
      if p1↑.lh then
      begin {LL} p↑.ľavý: = p1↑.pravý;
        p1↑.pravý: = p; p1↑.lh: = false; p: = p1
      end else
      if p1↑.rh then                                        (4.87)
      begin {LR} p2: = p1↑.pravý; p1↑.rh: = false;
        p1↑.pravý: = p2↑.ľavý; p2↑.ľavý: = p1;
        p↑.ľavý: = p2↑.pravý; p2↑.pravý: = p; p: = p2
      end
    end else
    begin h: = h - 1; if h ≠ 0 then p↑.lh: = true
    end
  end else
  if x > p↑.kľúč then
  begin vyhľadaj (x, p↑.pravý, h);
    if h ≠ 0 then
    if p↑.rh then
    begin p1: = p↑.pravý; h: = 2; p↑.rh: = false;
      if p1↑.rh then
      begin {RR} p↑.pravý: = p1↑.ľavý;
        p1↑.ľavý: = p; p1↑.rh: = false; p: = p1
      end else
      if p1↑.lh then
      begin {RL} p2: = p1↑.ľavý; p1↑.lh: = false;
        p1↑.ľavý: = p2↑.pravý; p2↑.pravý: = p1;
        p↑.pravý: = p2↑.ľavý; p2↑.ľavý: = p; p: = p2
      end
    end else
    begin h: = h - 1; if h ≠ 0 then p↑.rh: = true
    end
  end else
  begin p↑.počet: = p↑.počet + 1; h: = 0
  end
end {vyhľadaj}
```

Všimnime si, že činnosti spojené s preusporiadaním vrcholov sú veľmi podobné činnostiam v algoritme vyhľadávania vo vyváženom strome (4.63). Zo schémy (4.87) je zjavné, že všetky štyri prípady sa dajú implementovať rotáciami smerníkov: v prípadoch LL a RR pôjde o jednoduchú rotáciu; v prípadoch LR a RL pôjde o dvojitú rotáciu. V skutočnosti sa procedúra (4.87) zdá byť jednoduchšou ako procedúra

kľúč vždy rovnaká. Algoritmus prehľadania tabuľky môžeme vyjadriť takto:

```
h: = H(k); i: = 0;
repeat
  if T[h].kľúč = k then prvok sa našiel else
  if T[h].kľúč = prázdny then prvok nie je v tabuľke else
  begin {kolízia}                                              (4.89)
    i: = i + 1; h: = H(k) + G(i)
  end
until nájdený, alebo nie je v tabuľke (alebo tabuľka je plná)
```

Literatúra uvádza rôzne funkcie na ošetrovanie kolízií. Súhrnný prehľad vypracovaný MORISSOM [4-8] v roku 1968 spôsobil v tejto oblasti značný rozruch. Najjednoduchšou metódou je cyklické prehľadávanie tabuľky dovtedy, kým sa nenájde prvok s hľadaným kľúčom alebo prázdna pozícia. Preto $G(i) = i$; indexy h_i používané v pokusoch sú v tomto prípade

$$\begin{aligned} h_0 &= H(k) \\ h_i &= (h_0 + i) \ \mathbf{mod}\ N, \qquad i = 1, \ldots, N-1 \end{aligned} \tag{4.90}$$

Tejto stratégii hovoríme *metóda lineárnych pokusov*. Jej výhoda spočíva v tom, že prvky majú sklon zhlukovať sa okolo primárnych kľúčov (t.j. kľúčov, ktoré pri ich vložení nespôsobili kolíziu). Ideálne by sa mala zvoliť taká funkcia G, ktorá by jednotlivé kľúče opäť rozptyľovala čo najrovnomernejšie po množine zostávajúcich tabuľkových pozícií. V praxi by to však bolo príliš drahé, a preto sa uprednostňujú metódy, ktoré predstavujú kompromis v tom, že príslušné transformačné funkcie sú ľahko vypočítateľné, a pritom ešte stále efektívnejšie ako lineárna funkcia (4.90). Jednou z nich je metóda využívajúca kvadratickú transformačnú funkciu, ktorá generuje nasledujúcu postupnosť alternatívnych indexov:

$$\begin{aligned} h_0 &= H(k) \\ h_i &= (h_0 + i^2) \ \mathbf{mod}\ N \qquad (i > 0) \end{aligned} \tag{4.91}$$

Všimnime si, že ak použijeme rekurentné vzťahy (4.92) pre $h_i = i^2$

a $d_i = 2i + 1$, tak výpočet ďalšieho indexu nebude zahŕňať operáciu umocnenia.

$$\begin{aligned} h_{i+1} &= h_i + d_i \\ d_{i+1} &= d_i + 2 \qquad (i > 0) \end{aligned} \tag{4.92}$$

Pritom platí $h_0 = 0$ a $d_0 = 1$. Táto stratégia sa nazýva *metóda kvadratických pokusov* a jej výhoda spočíva v tom, že zabraňuje primárnemu zhlukovaniu, a pritom nevyžaduje žiadny dodatočný výpočet. Nepatrnou nevýhodou je, že pri pokusoch sa prehľadávajú všetky prvky v tabuľke, čím môže vzniknúť situácia, že pri pridávaní prvkov sa nemusí nájsť prázdna pozícia, hoci sa v tabuľke vyskytuje. V skutočnosti sa pri kvadratických pokusoch prehľadá aspoň polovica tabuľky, ak je jej veľkosť N prvočíslo. Toto tvrdenie možno odvodiť z nasledujúcej úvahy; ak je výsledok i-tého a j-tého pokusu ten istý prvok tabuľky, tak túto skutočnosť môžeme vyjadriť rovnicou

$$i^2 \ \mathbf{mod}\ N = j^2 \ \mathbf{mod}\ N$$

alebo

$$(i^2 - j^2) \equiv 0 \ (\mathbf{mod}\ N)$$

Úpravou rozdielu mocnín i, j dostávame

$$(i + j)(i - j) \equiv 0 \ (\mathbf{mod}\ N)$$

Pretože $i \neq j$, musí sa buď i, alebo j rovnať aspoň $N/2$, aby platilo $i + j = cN$, pričom c je celé číslo.

V praxi však táto nepatrná nevýhoda nehrá podstatnú úlohu, pretože pri $N/2$ sekundárnych pokusoch sa kolízie prejavia iba v prípade, keď je tabuľka takmer plná.

Príkladom aplikácie techniky rozptyľovania prvkov v pamäti môže byť generátor krížových odkazov (vyjadrený programom 4.5) prepracovaný na základe uvedených stratégií do programu 4.8. Podstatné rozdiely predstavuje iba procedúra vyhľadávania a nahradenie smerníka typu slovref transformačnou tabuľkou T, obsahujúcou slová zdrojového textu nejakého programu. Transformačná funkcia H má tvar $H(k) = k \ \mathbf{mod}\ N$, kde N je veľkosť tabuľky; na spracovanie kolízií zvolíme metódu kvadratických pokusov. Uvedomme si, že vzhľadom

na dobrú výkonnosť algoritmu je žiadúce, aby veľkosť tabuľky bola vyjadrená prvočíslom.

I keď je v tomto prípade metóda transformácie kľúča najefektívnejšia, dokonca oveľa efektívnejšia metóda ako stromové organizácie, má predsa len aj svoju nevýhodu. Len čo sa prečíta celý zdrojový text a jednotlivé slová sa umiestnia do tabuľky, chceme vytlačiť tieto slová v abecednom poradí. Pri stromovej organizácii vybudovanej na základe princípov usporiadaných vyhľadávacích stromov je to veľmi priamočiare. Nie je to však prípad transformácie kľúča. Úplný zmysel transformácie kľúča sa stáva zrejmým. Okrem toho, že procesu vytlačenia tabuľky musí predchádzať triedenie (program 4.8 používa pre jednoduchosť metódu triedenia priamym výberom), zdá sa výhodné uchovávať informácie aj o pridávaní jednotlivých kľúčov do tabuľky vo forme špeciálneho zoznamu. Preto je vyššia efektívnosť metódy transformácie kľúča pri výberoch vyvážená dodatočnými operáciami potrebnými pri realizácii celej úlohy generovania usporiadaného zoznamu krížových odkazov.

PROGRAM 4.8. *Generátor krížových odkazov (využívajúci transformačnú tabuľku)*

```
program Križrefgen (f, output);
{generátor krížových odkazov využívajúci transformačnú tabuľku}
label 13;
const c1 = 10;      {dĺžka slov}
      c2 = 8;       {počet čísel v riadku}
      c3 = 6;       {počet číslic v jednom čísle}
      c4 = 9999;    {maximálny počet riadkov}
       p = 997;     {prvočíslo}
      prázdny = '          ';
type index = 0..p;
     polref = ↑prvok;
     slovo = record kľúč: alfa;
                    prvý, posledný: polref;
                    fol: index
             end;
     prvok = packed record
                          lno: 0..c4;
                          ďalší: polref
             end;
var i, top: index;
    k, k1: integer;
    n: integer;   {číslo momentálneho riadku}
    id: alfa;
    f: text;
    a: array [1..c1] of char;
    t: array [0..p] of slovo;   {transformačná tabuľka}
 procedure vyhľadaj;
   var h, d, i: index;
       x: polref; f: boolean;
 {globálne premenné: t, id, top}
 begin h: = ord (id) mod p;
   f: = false; d: = 1;
   new (x); x↑.lno: = n; x↑.ďalší: = nil;
   repeat
     if t[h].kľúč = id then
     begin {nájdený} f: = true;
       t[h].posledný↑.ďalší: = x; t[h].posledný: = x
     end else
     if t[h].kľúč = prázdny then
       begin {nový prvok} f: = true;
         with t[h] do
         begin kľúč: = id; prvý: = x; posledný: = x;
         fol: = top
         end;
         top: = h
       end else
       begin {kolízia} h: = h + d; d: = d + 2;
         if h ≧ p then h: = h − p;
         if d = p then
         begin writeln ('PREPLNENIE TABUĽKY'); goto 13
         end
```

```
      end
    until f
end {vyhľadaj};
procedure vytlačtabuľku;
  var i, j, m: index;
  procedure vytlačslovo (w: slovo);
    var l: integer; x: polref;
  begin write (' ', w.kľúč);
    x: = w.prvý; l: = 0;
    repeat if l = c2 then
          begin writeln;
            l: = 0; write (' ': c1 + 1)
          end;
      l: = l + 1; write (x↑.lno: c3); x: = x↑.ďalší
    until x = nil;
    writeln
  end {vytlačslovo};
begin i: = top;
  while i ≠ p do
  begin {prehľadanie zoznamu a nájdenie minimálneho kľúča}
    m: = i; j: = t[i].fol;
    while j ≠ p do
      begin if t[j].kľúč < t[m].kľúč then m: = j;
        j: = t[j].fol
      end;
    vyltačslovo (t[m]);
    if m ≠ i then
      begin t[m].kľúč: = t[i].kľúč;
        t[m].prvý: = t[i].prvý; t[m].posledný: = t[i].posledný
      end;
    i: = t[i].fol
  end
end {vytlačtabuľku};
begin n: = 0; k1: = c1; top: = p; reset (f);
  for i: = 0 to p do t[i].kľúč: = prázdny;
  while ¬ eof (f) do
```



```
  begin if n = c4 then n: = 0;
    n: = n + 1; write (n: c3);   {ďalší riadok}
    write (' ');
    while ¬ eoln (f) do
    begin {preskúmanie neprázdneho riadku}
      if f↑ in ['A' .. 'Z'] then
      begin k: = 0;
        repeat if k < c1 then
          begin k: = k + 1; a[k]: = f↑;
          end;
          write (f↑); get (f)
        until ¬ (f↑ in ['A' .. 'Z', '0' .. '9']);
        if k ≧ k1 then k1: = k else
        repeat a[k1]: = ' '; k1: = k1 - 1
        until k1 = k;
        pack (a, 1, id); vyhľadaj;
      end else
      begin {test na úvodzovku alebo poznámku}
        if f↑ = '''' then
          repeat write (f↑); get (f)
          until f↑ = '''' else
        if f↑ = '{' then
          repeat write (f↑); get (f)
          until f↑ = '}';
        write (f↑); get (f)
      end
    end;
    writeln; get (f)
  end;
13: page; vytlačtabuľku
end.
```

4.6.3 ANALÝZA TRANSFORMÁCIE KĽÚČA

Pridávanie a výber metódou transformácie kľúča má v najhoršom prípade zjavne úbohú výkonnosť. Napokon je ľahko možné, že argu-

ment vyhľadávania bude taký, že jednotlivé pokusy zasiahnu presne všetky obsadené pozície tabuľky, pričom minú žiadúce (alebo voľné) pozície. Pripomíname, že každý, kto sa rozhodne použiť metódu transformácie kľúča, by mal dôverovať zákonom teórie pravdepodobnosti. V čom sa potrebujeme uistiť? Predovšetkým v tom, že v priemere bude počet pokusov malý. A ako uvidíme ďalej (z výpočtu koeficientu pravdepodobnosti), tento počet bude dokonca veľmi malý.

Predpokladajme opäť, že všetky kľúče sú rovnako pravdepodobné a transformačná funkcia H ich distribuuje rovnomerne do množiny indexov tabuľky. Predpokladajme ďalej, že pridávame kľúč do tabuľky s veľkosťou n, ktorá už obsahuje k prvkov. Pravdepodobnosť zásahu voľného miesta na prvý pokus je $1 - k/n$. Toto je súčasne pravdepodobnosť p_1, že treba iba jediné porovnanie. Pravdepodobnosť potreby jediného sekundárneho pokusu sa rovná súčinu pravdepodobnosti kolízie v prvom pokuse a pravdepodobnosti zásahu voľného miesta v ďalšom pokuse. Vo všeobecnosti môžeme pravdepodobnosť p_i zásahu voľného miesta na i-tý pokus vyjadriť vzťahom

$$\begin{aligned}
p_1 &= \frac{n-k}{n} \\
p_2 &= \frac{k}{n}\cdot\frac{n-k}{n-1} \\
p_3 &= \frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{n-k}{n-2} \\
&\cdots\cdots\cdots \\
p_i &= \frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{k-2}{n-2}\cdot\ldots\cdot\frac{k-i+2}{n-i+2}\cdot\frac{n-k}{n-i+1}
\end{aligned} \tag{4.93}$$

Očakávaný počet pokusov potrebných na pridanie $(k+1)$-ho kľúča je potom

$$\begin{aligned}
E_{k+1} &= \sum_{i=1}^{k+1} i.p_i = 1\cdot\frac{n-k}{n} + 2\frac{k}{n}\frac{n-k}{n-1} + \ldots + (k+1)\,. \\
&\cdot\left(\frac{k}{n}\frac{k-1}{n-1}\frac{k-2}{n-2}\cdots\frac{1}{n-k+1}\right) = \frac{n+1}{n-k+1}
\end{aligned} \tag{4.94}$$

Pretože počet pokusov potrebných na pridanie prvku je zhodný s počtom pokusov potrebných na jeho vybratie, možno výsledok (4.94) použiť na vypočítanie priemerného počtu pokusov E potrebných na vyhľadanie náhodného kľúča v tabuľke. Nech je veľkosť tabuľky opäť určená konštantou n a nech symbol m vyjadruje skutočný počet kľúčov v tabuľke. Potom

$$\begin{aligned}
E &= \frac{1}{m}\sum_{k=1}^{m} E_k = \frac{n+1}{m}\sum_{k=1}^{m}\frac{1}{n-k+2} = \\
&= \frac{n+1}{m}\,(H_{n+1} - H_{n-m+1})
\end{aligned} \tag{4.95}$$

kde

$$H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$$

je harmonická funkcia. Aproximáciou H_n dostaneme $H_n \cong \ln(n) + \gamma$, kde γ je Eulerova konštanta. Substitúciou $\alpha = m/(n+1)$ dostaneme

$$\begin{aligned}
E &= \frac{1}{\alpha}(\ln(n+1) - \ln(n-m+1)) = \frac{1}{\alpha}\ln\frac{n+1}{n+1-m} = \\
&= \frac{-1}{\alpha}\ln(1-\alpha)
\end{aligned} \tag{4.96}$$

pričom α vyjadruje približne pomer obsadených a voľných miest v tabuľke. Nazýva sa faktor zaplnenia tabuľky. Ak $\alpha = 0$, tak tabuľka je prázdna; ak $\alpha = n/(n+1)$, tabuľka je plná. Očakávaný počet pokusov E pri vyberaní a pridávaní náhodne zvoleného kľúča je uvedený v *tab. 4.6.* ako funkcia faktora zaplnenia tabuľky. Číselné výsledky sú naozaj prekvapujúce a potvrdzujú výnimočne dobrú výkonnosť metódy transformácie kľúča. Dokonca aj v prípade, keď je tabuľka na 90 % zaplnená, potrebujeme na nájdenie kľúča alebo prázdneho miesta v priemere iba 2,56 pokusov. Uvedomme si najmä to, že tieto číslice nezávisia od absolútneho počtu prítomných kľúčov, ale iba od faktora zaplnenia tabuľky.

Uvedená analýza bola založená na použití metódy ošetrovania kolízií, ktorá rozptyľovala kľúče po zostávajúcich miestach tabuľky. Meto-

Očakávaný počet pokusov ako funkcia faktora zaplnenia tabuľky

Tabuľka 4.6

a	E
0,1	1,05
0,25	1,15
0,5	1,39
0,75	1,85
0,9	2,56
0,95	3,15
0,99	4,66

dy používané v praxi prinášajú o niečo horšie výsledky. Detailná analýza metódy lineárnych pokusov udáva pre očakávaný počet pokusov výsledok (pozri aj [4-10])

$$E = \frac{1 - a/2}{1 - a} \tag{4.97}$$

Niekoľko číselných hodnôt $E(a)$ je zobrazených v *tab. 4.7.*

Výsledky získané dokonca i pre najtriviálnejšiu metódu ošetrovania kolízií sú také dobré, že máme pokušenie pokladať transformáciu kľúča

Očakávaný počet pokusov pri metóde lineárnych pokusov

Tabuľka 4.7

a	E
0,1	1,06
0,25	1,17
0,5	1,50
0,75	2,50
0,9	5,50
0,95	10,50

(hašovanie) za všeliek. Je to tak predovšetkým preto, že výkonnosť tejto metódy je lepšia ako aj tá najpremyslenejšia stromová organizácia, ak by sme uvažovali aspoň o porovnaní potrebných krokov na vyhľadávanie a pridávanie. Treba však explicitne poukázať i na niektoré slabiny metódy transformácie kľúča, aj keď pôjde o nepredpojaté úvahy.

Zaiste veľkou nevýhodou metódy transformácie kľúča oproti technikám, ktoré používajú dynamické prideľovanie pamäti, je stabilná veľkosť tabuľky, čo znemožňuje jej flexibilné prispôsobovanie potrebám. Je preto dôležité dobre odhadnúť možný počet prvkov, aby sme sa vyhli buď nehospodárnemu využitiu pamäti, alebo zlej výkonnosti (alebo i preplneniu tabuľky). V prípade, že je maximálny počet prvkov presne známy (čo býva dosť zriedkavý prípad), odporúča sa, aby vzhľadom na dobrú výkonnosť bola veľkosť tabuľky o niečo väčšia (povedzme o 10 %).

Druhá veľká nevýhoda transformačných techník sa objaví vtedy, keď budeme chcieť kľúče, okrem ich pridávania a vyhľadávania, aj rušiť. Zrušenie prvku z transformačnej tabuľky je ťažkopádne, ak sa nepoužije metóda priameho zreťazenia vo zvláštnej oblasti preplnenia. Záverom môžeme teda konštatovať, že stromové organizácie sú stále atraktívne a mali by sa aj uprednostňovať, pokiaľ je objem údajov neznámy, silne variabilný, alebo sa dokonca aj zmenšuje.

Cvičenia

4.1. Zaveďme pojem rekurzívneho typu.

rectype $T = T_0$

označujúceho zjednotenie množiny hodnôt definovaných typom T_0 a jednoduchej hodnoty **none**, t. j.

$T = T_0 \cup$ {**none**}

Definícia typu rod (pozri (4.3)) sa potom môže zjednodušiť na

```
rectype rod = record meno: alfa;
                     otec, matka: rod
              end
```

Ako bude vyzerať zobrazenie pamäti pre takúto rekurzívnu štruktúru, ktoré by zodpovedalo *obr. 4.2*?

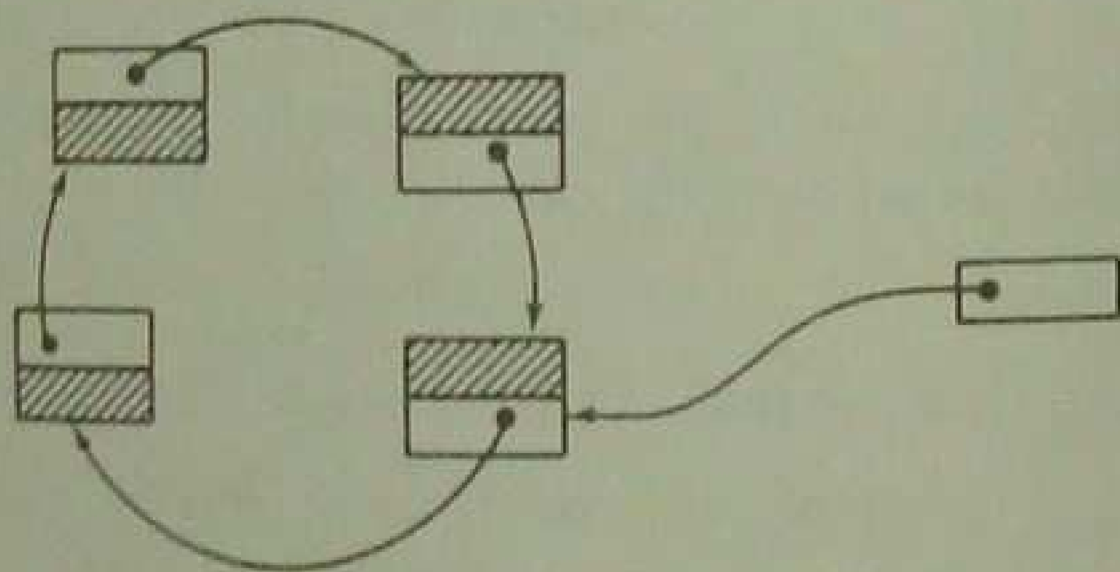
Predpokladáme, že implementácia takejto štruktúry bude založená na princípoch dynamického prideľovania pamäti, pričom zložky otec a matka budú obsahovať smerníky, ktoré budú automaticky generované a pred programátorom zostanú skryté. Aké ťažkosti vzniknú pri takejto implementácii?

4.2. Definujte štruktúru údajov opísanú v poslednom odseku článku 4.2 pomocou záznamov a smerníkov. Možno vyjadriť tamojšie rodinné vzťahy pomocou rekurzívneho typu, ktorý sme zaviedli v predchádzajúcom cvičení?

4.3. Predpokladajme, že je FIFO — front (t.j. front typu prvý-dnu-prvý-von) Q s prvkami typu T_0 implementovaný pomocou zoznamu. Definujte vhodnú štruktúru údajov, procedúry na pridávanie a rušenie prvkov z frontu Q a funkciu, ktorá zistí, či je front Q prázdny alebo nie. Tieto procedúry majú mať svoj vlastný mechanizmus pre ekonomické opätovné využitie pamäti.

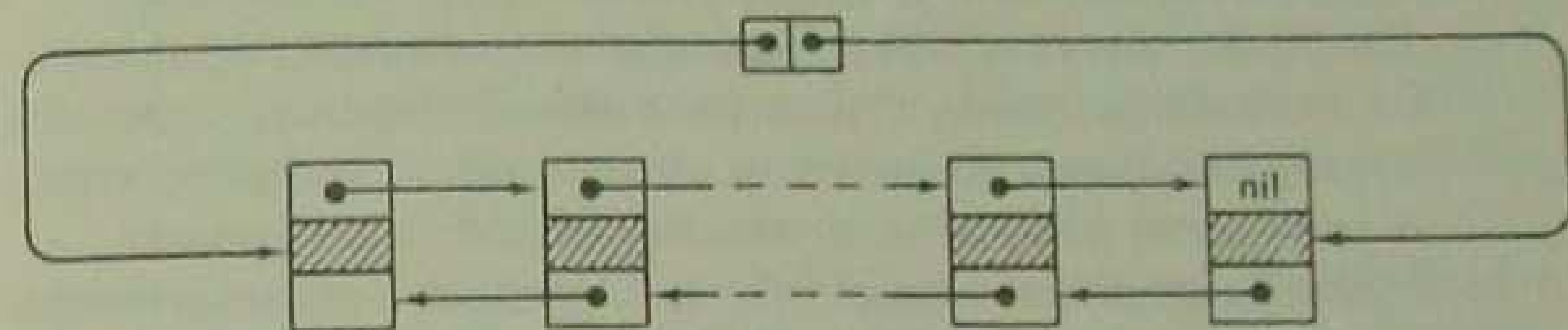
4.4. Predpokladajte, že záznamy v lineárnom zozname obsahujú položku *kľúč*, ktorá je typu integer. Napíšte program, ktorý by tento zoznam utriedil vzostupne podľa hodnôt kľúčov. Napíšte aj procedúru na obrátenie zoznamu.

4.5. Kruhové zoznamy (pozri *obr. 4.54*) obyčajne obsahujú čelo zoznamu. Aký je dôvod na jeho zavedenie? Napíšte procedúry na pridávanie, rušenie a vyhľadávanie prvkov, určených daným kľúčom. Tieto procedúry realizujte raz za predpokladu existencie čela zoznamu a raz bez neho.



Obr. 4.54. Kruhový zoznam

4.6. Obojsmerný zoznam je tvorený prvkami, ktoré sú pospájané v oboch smeroch (pozri *obr. 4.55*). Obidve spojenia vychádzajú z čela. Analogicky s predchádzajúcim príkladom vytvorte procedúry na vyhľadávanie a rušenie prvkov.



Obr. 4.55. Obojsmerný zoznam

4.7. Bude program 4.2 pracovať správne, ak sa určitá dvojica $\langle x, y \rangle$ vyskytne viackrát vo vstupnom súbore?

4.8. Správa „TÁTO MNOŽINA NIE JE ČIASTOČNE USPORIADANÁ", uvedená v programe 4.2, v mnohých prípadoch nie je veľmi užitočná. Rozšírte tento program tak, aby na výstupe produkoval postupnosť prvkov, ktoré by vytvárali cyklus, ak tam cyklus skutočne existuje.

4.9. Napíšte program, ktorý číta zdrojový text programu, identifikuje definície a volania všetkých procedúr (podprogramov) a topologicky usporiada všetky podprogramy. Nech $P \prec Q$ znamená, že P je volaná z Q.

4.10. Nakreslite strom vytvorený programom 4.3 pre prípad vstupu, ktorý by obsahoval $n + 1$ čísel

$$n, 1, 2, 3, \ldots, n$$

4.11. Ako by vyzerali postupnosti vrcholov, keby sme stromom na *obr. 4.20* prechádzali priamo, vnútorne a spätne?

4.12. Nájdite kritérium kompozície postupnosti n čísel, ktorá, ak sa použije ako vstup pre program 4.4, spôsobí vytvorenie dokonale vyváženého stromu.

4.13. Uvažujte o nasledujúcich dvoch spôsoboch prechodu binárnym stromom:

1. a) prechod pravým podstromom,

b) navštívenie koreňa,

c) prechod ľavým podstromom,

2. a) navštívenie koreňa,

b) prechod pravým podstromom,

c) prechod ľavým podstromom.

Zistite, či existujú nejaké jednoduché vzťahy medzi postupnosťami vrcholov, ktoré by vznikli pri prechode binárnym stromom uvedenými dvoma spôsobmi, a postupnosťami vrcholov, vzniknutými tromi prechodmi stromu, ktoré sme uviedli v texte.

4.14. Definujte štruktúru údajov vhodnú na reprezentáciu n-árneho stromu. Potom napíšte procedúru, ktorá týmto stromom prechádza a generuje binárny strom, obsahujúci tie isté prvky. Predpokladajte, že kľúč prvku zaberá k slov a každý smerník jedno slovo pamäti. Koľko pamäti ušetríme, ak namiesto n-árneho stromu použijeme binárny strom?

4.15. Predpokladajte, že strom je vytvorený na základe nasledujúcej definície rekurzívnej štruktúry (pozri aj cvičenie 4.1):

```
rectype strom = record x: integer;
                       ľavý, pravý: strom
                end
```

Formulujte procedúru na vyhľadanie prvku s daným kľúčom x a vykonanie operácie P nad týmto prvkom.

4.16. V systéme súborov je katalóg všetkých súborov organizovaný formou usporiadaného binárneho stromu. Každý vrchol označuje jeden súbor a špecifikuje jeho meno a okrem iného aj dátum posledného prístupu doňho, zakódovaný ako celé číslo.

Napíšte program na prechod stromu a zrušenie všetkých súborov, ktorých posledný prístup bol pred určitým dátumom.

4.17. Frekvencia prístupu k jednotlivým prvkom stromovej štruktúry sa meria empiricky, na základe počítadla prístupov pridaného ku každému vrcholu. V určitých časových intervaloch sa stromová štruktúra aktualizuje prechodom stromu a vytvorením nového, pomocou programu 4.4, pričom sa kľúče pridávajú na základe zostupného počtu frekvencií prístupov. Napíšte program, ktorý vykoná túto reorganizáciu. Bude priemerná dĺžka cesty tohto stromu rovnaká, horšia alebo oveľa horšia ako v prípade optimálneho stromu?

4.18. Metódu analýzy algoritmu stromového pridávania, opísanú v článku 4.5, možno použiť aj na výpočet očakávaného počtu porovnaní C_n a presunov (zámen) M_n, uskutočňovaných algoritmom quicksort (pozri program 2.10). Tento algoritmus triedi n-prvkové pole za predpokladu, že všetky permutácie n kľúčov $\{1, 2, \ldots, n\}$ (je ich $n!$) sú rovnako pravdepodobné. Nájdite analógiu a určte C_n a M_n.

4.19. Nakreslite vyvážený strom s 12 vrcholmi, ktorý má maximálnu výšku spomedzi všetkých vyvážených stromov s 12 vrcholmi. Ako by mala vyzerať postupnosť týchto vrcholov, aby procedúra (4.63) vytvorila takýto strom?

4.20. Nájdite postupnosť n pridávaných kľúčov, aby procedúra (4.63) vykonala každé znovuvyváženie (LL, LR, RR, RL) aspoň raz. Aká bude minimálna dĺžka n takejto postupnosti?

4.21. Nájdite vyvážený strom s kľúčmi $1, \ldots, n$ s takou permutáciou týchto kľúčov, aby pri použití procedúry rušenia (4.64) bolo každé zo štyroch znovuvyvážení použité aspoň raz. Aká bude postupnosť s minimálnou dĺžkou n?

4.22. Aká je priemerná dĺžka cesty Fibonacciho stromu T_n?

4.23. Napíšte program, ktorý by generoval skoro optimálny strom podľa algoritmu založeného na voľbe ťažiska za koreň stromu (4.78).

4.24. Predpokladajte, že kľúče 1, 2, 3, ... sa pridávajú do prázdneho B-stromu druhého rádu (program 4.7). Ktoré kľúče spôsobia rozdelenie stránky? Ktoré kľúče spôsobia zväčšenie výšky stromu?

Ak sa budú všetky kľúče rušiť rovnakým spôsobom, ktoré z nich spôsobia zlúčenie stránok (a tým aj ich zrušenie) a ktoré z nich zmenšenie výšky stromu? Odpovedajte na túto otázku:

a) so zreteľom na schému rušenia s vyvážením (ako v prípade programu 4.7),

b) so zreteľom na schému bez vyváženia (t. j. ak dôjde k výskytu nezaplnenej stránky, zoberie sa prvok zo susednej stránky).

4.25. Napíšte program na vyhľadávanie, pridávanie a rušenie kľúčov

v binárnom B-strome. Použite definíciu typu vrchol (4.84). Schému pridávania vidno na *obr. 4.51*.

4.26. Nájdite postupnosť pridávaných kľúčov, aby procedúra (4.87), začínajúca prázdnym symetrickým binárnym B-stromom, vykonala každé zo štyroch znovuvyvážení (*LL*, *LR*, *RR*, *RL*) aspoň raz. Aká bude najkratšia takáto postupnosť?

4.27. Napíšte procedúru pre rušenie prvkov v symetrickom binárnom B-strome. Nájdite potom strom a krátku postupnosť rušení, spôsobujúcich každé jedno zo znovuvyvážení aspoň raz.

4.28. Porovnajte výkonnosť algoritmu pridávania a rušenia v binárnom strome, vo vyváženom AVL-strome a v symetrickom binárnom B-strome na vašom počítači. Preskúmajte predovšetkým účinok zhustenia údajov, t.j. voľby ekonomickej reprezentácie údajov s použitím iba dvoch bitov na vyjadrenie informácie o vyvážení každého vrcholu.

4.29. Modifikujte algoritmus tlače v programe 4.6 takým spôsobom, aby sa dal použiť na zobrazenie symetrického binárneho B-stromu s vodorovnými a zvislými hranami.

4.30. Ak je množstvo informácií spojených s každým kľúčom pomerne veľké (v porovnaní so samotným kľúčom), nemali by sa informácie uchovávať v transformačnej tabuľke. Vysvetlite prečo a navrhnite schému reprezentácie takejto množiny informácií.

4.31. Pouvažujte nad návrhom riešenia problému zhlukovania pomocou stromov preplnenia namiesto zoznamov preplnenia, t.j. nad organizovaním tých kľúčov, ktoré ako stromové štruktúry spôsobovali kolízie. Teda každý prvok rozptýlenej (transformačnej) tabuľky sa dá pokladať za koreň (snáď i prázdneho) stromu (stromové transformácie).

4.32. Navrhnite schému, ktorá realizuje pridávanie a rušenie v transformačnej tabuľke s použitím metódy kvadratických pokusov na riešenie kolízií. Porovnajte experimentálne túto schému s organizáciou priamych binárnych stromov použitím náhodnej postupnosti kľúčov na pridávanie a rušenie.

4.33. Hlavnou nevýhodou techniky transformačnej tabuľky je pevná veľkosť tabuľky, ktorú treba zvoliť v okamihu, keď ešte nie je známy skutočný počet jej prvkov. Predpokladajte, že váš počítač

poskytuje možnosť dynamického prideľovania pamäti v ľubovoľnom okamihu. Teda keď je transformačná tabuľka *H* plná (alebo skoro plná), vytvorí sa väčšia tabuľka *H'* a všetky kľúče tabuľky *H* sa premiestnia do tabuľky *H'*, čím sa pamäť potrebná pre tabuľku *H* vráti systému na opätovné použitie. Tejto technike hovoríme transformácia kľúča so sekundárnou tabuľkou. Napíšte program, ktorý túto techniku aplikuje na tabuľku *H* s veľkosťou *n*.

4.34. V praxi sa často stáva, že kľúče nie sú reprezentované celými číslami, ale postupnosťami písmen. Tieto slová bývajú často rôzne veľké, a preto sa nedajú vhodne a ekonomicky umiestniť do kľúčových prvkov pevnej dĺžky. Napíšte program, ktorý narába s transformačnou tabuľkou obsahujúcou kľúče premenlivej dĺžky.

Zoznam použitej literatúry

4-1. Adelson-Velskij, G. M. — Landis, E. M.: Doklady Akademia Nauk SSSR. 146, (1962), 263—266; Anglický preklad v Soviet Math. 3, 1259—1263.

4-2. Bayer, R. — McCreight, E.: Organization and Maintenance of Large Ordered Indexes. Acta Informatica 1, No. 3 (1972), s. 173—189.

4-3. Bayer, R. — McCreight, E.: Binary B-trees for Virtual Memory. Proc. 1971 ACM SIGFIDET Workshop, San Diego, nov. 1971, s. 219—235.

4-4. Bayer, R. — McCreight, E.: Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. Acta Informatica, 1, No. 4 (1972), s. 290—306.

4-5. Hu, T. C. — Tucker, A. C.: SIAM J. Applied Math, 21, No. 4 (1971), s. 514—532.

4-6. Knuth, D. E.: Optimum Binary Search Trees. Acta Informatica 1, No. 1 (1971), s. 14—25.

4-7. Maurer, W. D.: An Improved Hash Code for Scatter Storage. Comm. ACM, 11, No. 1 (1968), s. 35—38.

4-8. Morris, R.: Scatter Storage Techniques. Comm. ACM, 11, No. 1 (1968), s. 38—43.

4-9. Peterson, W. W.: Addressing for Random-access Storage. IBM J. Res. and Dev., 1 (1957), s. 130—146.

4-10. Schay, G. — Spruth, W.: Analysis of a File Addressing Method. Comm. ACM, 5, No. 8 (1962), s. 459—462.

4-11. Walker, W. A. — Gotlieb, C. C.: A Top-down Algorithm for Constructing Nearly Optimal Lexicographie Trees v Graph Theory and Computing. New York: Academic Press (1972), s. 303—323.

4-12. Wiedermann, J.: Vyhľadávanie. Zborník referátov SOFSEM '81, VVS, Bratislava 1981, s. 277—326.

5 JAZYKOVÉ ŠTRUKTÚRY A KOMPILÁTORY

V tejto kapitole sa budeme snažiť vyvinúť *kompilátor* (*prekladač*) jednoduchého programovacieho jazyka. Proces tvorby tohto kompilátora nám súčasne poslúži ako príklad systematického, dobre štruktúrovaného vývoja zložitého a veľkého programu. V tomto zmysle je prekladač vítanou aplikáciou metód štruktúrovania programov a údajov, uvedených a rozpracovaných v predchádzajúcich kapitolách. Navyše naším ďalším cieľom je všeobecný úvod do problematiky štruktúry a činnosti kompilátorov. Jej poznanie a zvládnutie umožní hlbšie pochopiť umenie programovania vo vyšších programovacích jazykoch a programátorovi zjednoduší celkový proces vývoja programovacích systémov pre špecifické ciele a aplikačné oblasti. Pretože zložitosť kompilátorovej problematiky je všeobecne známa, treba túto kapitolu pokladať za úvodnú a prehľadovú. Azda za najdôležitejšiu možno považovať skutočnosť, že štruktúra jazyka sa odzrkadľuje v štruktúre jeho kompilátora, z čoho vyplýva, že jazyková zložitosť (alebo jednoduchosť) určuje zložitosť jeho kompilátora. Začneme preto opisom skladby jazyka, a potom sa sústredíme výlučne na jednoduché štruktúry, ktorých použitie vedie k jednoduchým modulárnym prekladačom. Ukazuje sa, že takéto jazykové konštrukcie sú vhodne aplikovateľné takmer vo všetkých praktických programovacích jazykoch.

5.1 DEFINÍCIA A ŠTRUKTÚRA JAZYKA

Každý jazyk je založený na nejakom slovníku. Jeho prvky sa obyčajne nazývajú slová; v teórii formálnych jazykov im hovoríme (základné) symboly. Pre každý jazyk je charakteristické, že určité postupnosti slov

sa považujú za správne, dobre vytvorené vety tohto jazyka, pričom iné sa považujú za nesprávne alebo zle vytvorené. Na základe čoho sa dá rozhodnúť, či daná postupnosť slov je správna veta alebo nie? Je to gramatika, *syntax* alebo štruktúra jazyka. Všeobecne definujeme syntax ako množinu pravidiel alebo formúl, ktorá definuje množinu (formálne konkrétnych) viet. Oveľa dôležitejšou vlastnosťou množiny takýchto pravidiel, okrem rozhodnutia, či daná postupnosť slov je veta alebo nie, je, že každej vete pripisujú istú štruktúru, ktorá je nápomocná pri určovaní významu vety. Je teda jasné, že syntax a *sémantika* (t. j. význam) sú úzko späté. Štrukturálne definície sa preto považujú za pomocné, vzhľadom na určité vyššie zámery. To nás, samozrejme, nesmie odradiť od začiatočného štúdia výlučne štrukturálnych aspektov bez uvažovania sémantiky a interpretácie.

Zoberme si napr. túto vetu: *Mačky spia.* Slovo *mačky* je podmet a *spia* prísudok tejto vety. Táto veta patrí do jazyka, ktorého syntax môže byť definovaná nasledujúcim spôsobom:

⟨veta⟩ : : = ⟨podmet⟩ ⟨prísudok⟩
⟨podmet⟩ : : = *mačky* | *psy*
⟨prísudok⟩ : : = *spia* | *jedia*

Význam týchto troch riadkov je takýto:

1. Vetu tvorí podmet, za ktorým nasleduje prísudok.
2. Podmetom môže byť buď slovo *mačky*, alebo slovo *psy*.
3. Prísudkom môže byť buď slovo *spia*, alebo slovo *jedia*.

Hlavnou myšlienkou uvedenej syntaxe je, že vetu možno odvodiť zo začiatočného symbolu ⟨veta⟩ opakovanou aplikáciou prepisovacích pravidiel.

Formalizmus alebo zápis (notácia) použitý na vyjadrenie týchto pravidiel sa nazýva *Backusova-Naurova forma* (BNF). Prvý raz sa použila pri definovaní jazyka algol 60 [5-7]. Vetné členy ⟨veta⟩, ⟨podmet⟩ a ⟨prísudok⟩ sa nazývajú *neterminálne symboly*; slová mačky, psy, spia a jedia nazývame *terminálne symboly* a prepisovacím pravidlám sa niekedy zvykne hovoriť, že sú to *produkcie*. Symboly : : = a | nazývame *metasymboly* jazyka BNF. Ak by sme sa rozhodli používať stručnejšiu formu zápisu, neterminálne symboly označíme veľkými

písmenami a terminálne malými písmenami, mohol by náš príklad vety vyzerať takto:

Príklad 1.

$$S ::= AB$$
$$A ::= x/y \quad (5.1)$$
$$B ::= z/w$$

Jazyk definovaný uvedenou syntaxou pozostáva zo štyroch viet xz, yz, xw, yw.

Na upresnenie uvedieme tieto matematické definície:

1. Nech je jazyk $L = L(T, N, P, S)$ špecifikovaný pomocou
a) slovníka T terminálnych symbolov,
b) množiny N neterminálnych symbolov (gramatických kategórií),
c) množiny P prepisovacích pravidiel (syntaktických pravidiel),
d) symbolu S (z množiny N) nazývaného *začiatočný symbol*.

2. Jazyk $L(T, N, P, S)$ je množina reťazcov terminálnych symbolov ξ, ktoré možno generovať (derivovať) zo začiatočného symbolu S podľa definície 3.

$$L = \{\xi | S \overset{*}{\Rightarrow} \xi \quad \text{a} \quad \xi \in T^*\} \quad (5.2)$$

(Na označenie reťazcov symbolov budeme používať grécke písmená.) T^* označuje množinu všetkých postupností symbolov zo slovníka T.

3. Reťazec σ_n možno generovať z reťazca σ_0 vtedy a len vtedy, ak existujú také reťazce $\sigma_1, \sigma_2, \ldots, \sigma_{n-1}$, že každý reťazec σ_i môže byť priamo generovaný z reťazca σ_{i-1} podľa definície 4.

$$(\sigma_0 \overset{*}{\Rightarrow} \sigma_n) \leftrightarrow ((\sigma_{i-1} \Rightarrow \sigma_i) \quad \text{pre} \quad i = 1, \ldots, n) \quad (5.3)$$

4. Reťazec η možno priamo generovať z reťazca ξ vtedy a len vtedy, ak existujú také reťazce α, β, ξ', η', že platí:
a) $\xi = \alpha\xi'\beta$,
b) $\eta = \alpha\eta'\beta$,
c) P obsahuje prepisovacie pravidlo $\xi' ::= \eta'$.

Poznamenávame, že zápis $\alpha ::= \beta_1 | \beta_2 | \ldots | \beta_n$ budeme používať ako skrátenú formu zápisu množiny prepisovacích pravidiel $\alpha ::= \beta_1$, $\alpha ::= \beta_2, \ldots, \alpha ::= \beta_n$.

Napríklad reťazec xz z príkladu 1 možno generovať nasledujúcou postupnosťou priamych krokov generovania: $S \Rightarrow AB \Rightarrow xB \Rightarrow xz$; teda $S \overset{*}{\Rightarrow} xz$, a pretože $xz \in T^*$, xz je veta jazyka, t. j. $xz \in L$. Všimnime si, že neterminálne symboly A a B sa objavia iba v nekoncových krokoch, zatiaľ čo koncové kroky musia viesť k reťazcu, ktorý obsahuje iba terminálne symboly. Gramatické pravidlá nazývame preto prepisovacie, lebo určujú, akým spôsobom možno nové formy generovať alebo prepisovať.

Jazyk nazývame bezkontextovým vtedy a len vtedy, ak sa dá definovať prostredníctvom bezkontextovej (nezávislej od kontextu) množiny prepisovacích pravidiel. Množina prepisovacích pravidiel je bezkontextová vtedy a len vtedy, ak všetky jej prvky majú tvar

$$A ::= \xi \qquad (A \in N,\ \xi \in (N \cup T)^*)$$

t. j. ak ľavá strana pozostáva z jednoduchého neterminálneho symbolu, ktorý možno nahradiť (prepísať) symbolom ξ bez ohľadu na kontext, v ktorom sa A vyskytuje. Ak má prepisovacie pravidlo tvar

$$\alpha A \beta ::= \alpha \xi \beta$$

hovoríme, že je kontextové a náhrada symbolu A symbolom ξ je možná len v rámci kontextu α a β. My sa sústredíme iba na bezkontextové systémy.

Príklad 2 nám názorne ukazuje, ako možno rekurzívne generovať prostredníctvom konečnej množiny prepisovacích pravidiel nekonečne veľa viet.

Príklad 2.

$$S ::= xA \quad (5.4)$$
$$A ::= z | yA$$

Zo začiatočného symbolu S možno potom generovať nasledujúce vety:

xz
xyz
$xyyz$
$xyyyz$
…

5.2 ANALÝZA VETY

Úlohou jazykových prekladačov alebo procesorov je najprv rozpoznávanie viet a vetných štruktúr a potom ich generovanie. To znamená, že všetky kroky generovania, ktoré vedú k vete, sa musia po jej rozpoznaní zrekonštruovať a súčasne použiť ako návratová cesta. To je však vo všeobecnosti veľmi zložitá a niekedy dokonca nemožná úloha. Jej zložitosť silne závisí od druhu prepisovacích pravidiel použitých v definícii jazyka. Úlohou teórie syntaktickej analýzy je vyvinúť rozpoznávacie algoritmy pre jazyky so značne zložitými štrukturálnymi pravidlami. Našou úlohou však bude načrtnúť spôsob tvorby analyzátorov, ktoré budú dostatočne jednoduché a efektívne pre praktické účely. To znamená, že výpočtové úsilie potrebné na analýzu vety musí byť lineárnou funkciou dĺžky vety; v najhoršom prípade môže byť funkčná závislosť $n.\log n$, pričom n je dĺžka vety. Pochopiteľne, nemôžeme sa zaoberať problémami nájdenia rozpoznávacieho algoritmu pre ľubovoľný jazyk. Budeme preto pracovať pragmaticky a v opačnom smere: najskôr budeme definovať efektívny algoritmus, a potom určíme triedu jazykov, ktoré možno pomocou neho analyzovať [5-3].

Prvým dôsledkom základnej požiadavky efektívnosti je, že výber každého kroku analýzy musí závisieť iba od súčasného stavu výpočtu a od nasledujúceho (práve načítaného) symbolu. Ďalšia a najdôležitejšia požiadavka je, aby žiadny krok nebol neskôr odvolaný. Tieto dve požiadavky sú všeobecne známe pod pojmom *metóda s predsnímaním jedného symbolu dopredu bez návratu*.

Základná metóda, ktorou sa budeme zaoberať, sa nazýva *syntaktická analýza zhora nadol*. Charakteristická je úsilím o rekonštrukciu krokov generovania (ktoré vo všeobecnosti tvoria štrukturálny strom) z ich začiatočného symbolu do konečnej vety, t.j. zhora nadol [5-5] a [5-6]. Začnime opäť príkladom 1: Máme danú vetu *Psy jedia*, o ktorej musíme rozhodnúť, či patrí do jazyka alebo nie. Podľa definície je to jediné v prípade, ak danú vetu možno (alebo nemožno) generovať zo začiatočného symbolu ⟨veta⟩. Z gramatických pravidiel vieme, že každá veta musí mať podmet, za ktorým nasleduje prísudok. Zvyšnú časť úlohy rozdelíme na dve časti: najprv rozhodneme, či sa určitá začiatočná časť vety dá (alebo nedá) generovať zo symbolu ⟨podmet⟩. Ako vidíme, symbol *psy* sa dá priamo generovať, preto ho môžeme považovať za spracovaný; zo vstupnej postupnosti ho vylúčime (t.j. načítame ďalší symbol) a prikročíme k druhej časti úlohy, a to k rozhodnutiu, či sa zvyšná časť vety dá (nedá) generovať zo symbolu ⟨prísudok⟩. Pretože sa dá generovať, môžeme výsledok analýzy považovať za správny. Celý proces analýzy si môžeme názorne zobraziť nasledujúcou schémou; v ľavej časti tejto schémy sú uvedené úlohy, ktoré treba rozriešiť, v pravej časti zvyšok vstupnej postupnosti:

⟨veta⟩	*psy jedia*
⟨podmet⟩ ⟨prísudok⟩	*psy jedia*
psy ⟨prísudok⟩	*psy jedia*
⟨prísudok⟩	*jedia*
jedia	*jedia*
—	—

Druhý príklad znázorňuje schému analýzy vety *xyyz* podľa prepisovacích pravidiel uvedených v príklade 2.

S	$xyyz$
xA	$xyyz$
A	yyz
yA	yyz
A	yz
yA	yz
A	z
z	z
—	—

Pretože proces sledovania jednotlivých krokov generovania vety sa nazýva *syntaktická analýza*, uvedený postup predstavuje *algoritmus syntaktickej analýzy*. Jednotlivé odvodzovacie kroky možno v uvedených dvoch príkladoch uskutočniť na základe prezretia ďalšieho symbolu vstupnej postupnosti. Žiaľ, ako uvidíme z nasledujúceho príkladu, nie je to vždy možné.

Príklad 3.

$$S ::= A \mid B$$
$$A ::= xA \mid y \qquad (5.5)$$
$$B ::= xB \mid z$$

Ak sa pokúsime o syntaktickú analýzu vety $xxxz$

S	$xxxz$
A	$xxxz$
xA	$xxxz$
A	xxz
xA	xxz
A	xz
xA	xz
A	z

zistíme, že sme uviazli. Ťažkosť vznikne už pri prvom kroku, v ktorom nedokážeme prezretím jediného ďalšieho symbolu rozhodnúť, či symbol S treba nahradiť symbolom A alebo symbolom B. Možné riešenie spočíva v pokračovaní podľa jednej z možných alternatív a v návrate po prejdenej ceste, ak ďalší postup už nie je možný. Táto metóda sa nazýva *prehľadávanie s návratom* (hovorili sme o nej v predchádzajúcej kapitole — pozn. prekl.). Pre jazyk z príkladu 3 neexistuje obmedzenie počtu nevybavených krokov. Taká situácia je rozhodne najmenej želateľná, a preto by sme sa v praxi mali vyhýbať jazykovým štruktúram spôsobujúcim prehľadávanie s návratom. V dôsledku toho sa budeme zaoberať iba takými gramatickými systémami, ktoré spĺňajú obmedzenie, že začiatočné symboly alternatívnych pravých častí prepisovacích pravidiel budú odlišné.

Tvrdenie 1.

Ak máme dané prepisovacie pravidlo

$$A ::= \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n$$

tak množiny začiatočných symbolov všetkých viet, ktoré možno generovať zo symbolov ξ_i', musia byť disjunktné, t. j.

$$\text{first}(\xi_i) \cap \text{first}(\xi_j) = \emptyset \quad \text{pre všetky} \quad i \neq j$$

Množina first (ξ) je množinou všetkých terminálnych symbolov, ktoré sa môžu vyskytnúť ako prvý symbol viet odvodených zo symbolu ξ. Nech je táto množina vypočítateľná podľa týchto zákonitostí:

1. Prvý symbol argumentu je terminálny symbol:

$$\text{first}(a\xi) = \{a\}$$

2. Prvý symbol je neterminálny symbol s prepisovacím pravidlom

$$A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Potom

$$\text{first}(A\xi) = \text{first}(\alpha_1) \cup \text{first}(\alpha_2) \cup \ldots \cup \text{first}(\alpha_n)$$

Všimnime si, že v príklade 3 platí $x \in \text{first}(A)$ a súčasne $x \in \text{first}(B)$. Na základe toho je prvým pravidlom porušené tvrdenie 1. Je však skutočne triviálne nájsť syntax jazyka uvedeného v príklade 3, ktorá by vyhovovala tvrdeniu 1. Riešenie spočíva v odložení faktorizácie dovtedy, kým sa „nevybavia" všetky symboly x. Nasledujúce prepisovacie pravidlá sú ekvivalentné s prepisovacími pravidlami (5.5) v tom zmysle, že generujú tú istú množinu viet:

$$S ::= C \mid xS \qquad (5.5a)$$
$$C ::= y \mid z$$

Žiaľ, tvrdenie 1 nás neuchráni pred ďalšou ťažkosťou. Majme takýto príklad:

Príklad 4.

$$S ::= Ax \qquad (5.6)$$
$$A ::= x \mid \varepsilon$$

Symbol ε označuje prázdny reťazec symbolov. Keď sa pokúsime analyzovať vetu x, môžeme sa dostať do „slepej uličky":

S	x
Ax	x
xx	x
x	—

Ťažkosť vzniká preto, lebo sme nepokračovali prepisovacím pravidlom $A ::= x$, ale prepisovacím pravidlom $A ::= \varepsilon$. Táto situácia sa nazýva problém prázdneho reťazca a vzniká iba v prípade neterminálnych symbolov, ktoré môžu generovať prázdny reťazec. Aby sme sa vyhli tejto situácii, zavedieme ďalšie tvrdenie.

Tvrdenie 2.

Pre každý symbol $A \in N$, ktorý generuje prázdny reťazec ($A \overset{*}{\Rightarrow} \varepsilon$), musí byť množina jeho začiatočných symbolov disjunktná s množinou symbolov, ktoré môžu nasledovať za ktorýmkoľvek reťazcom, generovaným zo symbolu A, t. j.

$$\text{first}(A) \cap \text{follow}(A) = \emptyset$$

Množina follow(A) sa vypočíta zohľadnením každého prepisovacieho pravidla P_i tvaru

$$X ::= \xi A \eta$$

a množiny $S_i = \text{first}(\eta_i)$. Množina follow$(A)$ je zjednotením všetkých množín S_i. Ak je aspoň jeden reťazec η_i schopný generovať prázdny reťazec, musí byť množina follow(X) obsiahnutá v množine follow(A). V príklade 4 je porušené tvrdenie 2 pre symbol A, pretože

$$\text{first}(A) = \text{follow}(A) = \{x\}$$

Zaužívaným spôsobom vyjadrenia opakujúceho sa reťazca symbolov je použitie rekurzívnej definície vetnej konštrukcie. Napríklad prepisovacie pravidlo

$$A ::= B \mid AB$$

opisuje množinu reťazcov B, BB, BBB, ... Jeho použitie je však znemožnené na základe tvrdenia 1, pretože

$$\text{first}(B) \cap \text{first}(AB) = \text{first}(B) \neq \emptyset$$

Ak nahradíme prepisovacie pravidlo mierne modifikovanou verziou

$$A ::= \varepsilon \mid AB$$

generujúcou reťazce ε, B, BB, BBB, ..., porušíme tvrdenie 2, pretože

$$\text{first}(A) = \text{first}(B)$$

a teda

$$\text{first}(A) \cap \text{follow}(A) \neq \emptyset$$

Dve spomenuté tvrdenia nepochybne zakazujú použitie definície s ľavou rekurziou. Jednoduchou metódou, ako sa vyhnúť takýmto tvarom, je buď použitie pravej rekurzie

$$A ::= \varepsilon \mid BA$$

alebo rozšírenie syntaxe jazyka BNF, aby umožňoval explicitné vyjadrenie opakovania; uskutočníme to tak, že zápisom $\{B\}$ označíme množinu reťazcov

$$\varepsilon, B, BB, BBB, \ldots$$

Prirodzene, musíme si byť vedomí toho, že každá takáto konštrukcia je schopná generovať prázdny reťazec. Zložené zátvorky { a } predstavujú metasymboly rozšíreného jazyka BNF.

Z uvedeného dôvodu a z transformácie prepisovacích pravidiel (5.5) na (5.5a) sa môže zdať, že „trik" transformácii gramatík by mohol byť všeliekom na všetky problémy týkajúce sa syntaktickej analýzy. Musíme však pamätať na to, že vetná štruktúra je pomocným prostriedkom pri určovaní vetného významu; teda vysvetlenia významu vetnej konštrukcie sú obyčajne vyjadrené pomocou významov vetných komponentov. Uvažujme napr. o gramatike, kde sa jazykové výrazy skladajú z operandov a, b, c a zo znamienka mínus znamenajúceho odčítanie:

$$S ::= A \mid S - A$$
$$A ::= a \mid b \mid c$$

V súlade s touto gramatikou má veta $a - b - c$ štruktúru, ktorá sa dá vyjadriť pomocou zátvoriek takto: $((a - b) - c)$. Keby sme však túto gramatiku transformovali na syntakticky ekvivalentný tvar bez ľavej rekurzie

$$S ::= A \mid A - S$$
$$A ::= a \mid b \mid c$$

tá istá veta by nadobudla inú štruktúru, ktorá sa dá vyjadriť v tvare $(a - (b - c))$. Ak uvážime zvyčajný význam odčítania, zistíme, že uvedené dva tvary sú sémanticky neekvivalentné.

Aké ponaučenie teda pre nás z uvedeného vyplýva? Predovšetkým také, že keď definujeme zmysluplný jazyk, musíme mať pri návrhu jeho syntaxe na zreteli vždy aj jeho sémantické štruktúry, pretože syntax musí zodpovedať sémantike.

5.3 KONŠTRUKCIA SYNTAKTICKÉHO GRAFU

V predchádzajúcom článku sme uviedli algoritmus syntaktickej analýzy zhora nadol vhodnej pre gramatiky, ktoré spĺňajú obmedzujúce tvrdenia 1 a 2. Teraz sa budeme snažiť transformovať tento algoritmus na konkrétny program. Existujú dve podstatne rozdielne techniky, ktoré sa dajú použiť. Prvá spočíva v návrhu všeobecného programu syntaktickej analýzy zhora nadol, platného pre všetky možné gramatiky (spĺňajúce tvrdenia 1 a 2). V tomto prípade musia byť jednotlivé gramatiky reprezentované nejakou štruktúrou údajov, s ktorou je daný program schopný pracovať. Takýto všeobecný syntaktický analyzátor je v istom zmysle riadený štruktúrou údajov; program je teda riadený tabuľkou. Druhá technika spočíva vo vyvinutí programu syntaktickej analýzy zhora nadol, ktorý je špecifický pre daný jazyk, a v jeho systematickej konštrukcii v súlade s množinou prepisovacích pravidiel zobrazujúcich danú syntax do postupnosti príkazov, t. j. do programu. Obidve techniky majú svoje výhody i nevýhody. Pri vývoji kompilátora daného programovacieho jazyka sa vyžaduje vysoký stupeň flexibility a parametrizácie všeobecného syntaktického analyzátora, pričom v prípade špecifického syntaktického analyzátora dospejeme k oveľa efektívnejším a ľahšie ovládateľným systémom. Preto sa uprednostňuje tento druhý spôsob. V oboch prípadoch je výhodné reprezentovať danú syntax pomocou syntaktického (derivačného) grafu, ktorý zobrazuje tok riadenia v rámci syntaktickej analýzy vety.

Charakteristickou vlastnosťou analýzy zhora nadol je, že jej cieľ je známy už na začiatku. Cieľom je rozpoznať vetu, t. j. reťazec symbolov

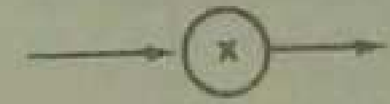
generovateľný zo začiatočného symbolu. Aplikácia prepisovacieho pravidla, t. j. nahradenie jednoduchého symbolu reťazcom symbolov, zodpovedá rozdeleniu jednoduchého cieľa na reťazec podcieľov, sledovaných v špecifickom poradí. Metóda zhora nadol sa potom nazýva aj cieľovo orientovaná syntaktická analýza. Pri konštrukcii syntaktického analyzátora sa dá jednoducho využiť zrejmá súvislosť medzi neterminálnymi symbolmi a cieľmi: zostrojíme podprogram syntaktickej analýzy pre každý neterminálny symbol. Cieľom každého z týchto podprogramov je rozpoznanie podvety, generovateľnej z jej zodpovedajúceho neterminálneho symbolu. Pretože však chceme zostrojiť graf reprezentujúci celý syntaktický analyzátor, bude potrebné, aby každý neterminálny symbol bol zobrazený do podgrafu. To nás vedie k nasledujúcim zásadám konštrukcie syntaktického grafu.

A1. Každý neterminálny symbol A so zodpovedajúcou množinou pravidiel

$$A ::= \xi_1 | \xi_2 | \ldots | \xi_n$$

sa zobrazí do syntaktického grafu A, ktorého štruktúra sa určuje na základe pravej strany prepisovacieho pravidla podľa zásad A2 až A6.

A2. Každý výskyt terminálneho symbolu x v ξ_i zodpovedá príkazu na rozpoznanie tohto symbolu a načítanie ďalšieho symbolu zo vstupnej vety. V grafe sa to zobrazuje týmto spôsobom (symbol x sa uvádza v krúžku, do ktorého smeruje a z ktorého vychádza orientovaná hrana):

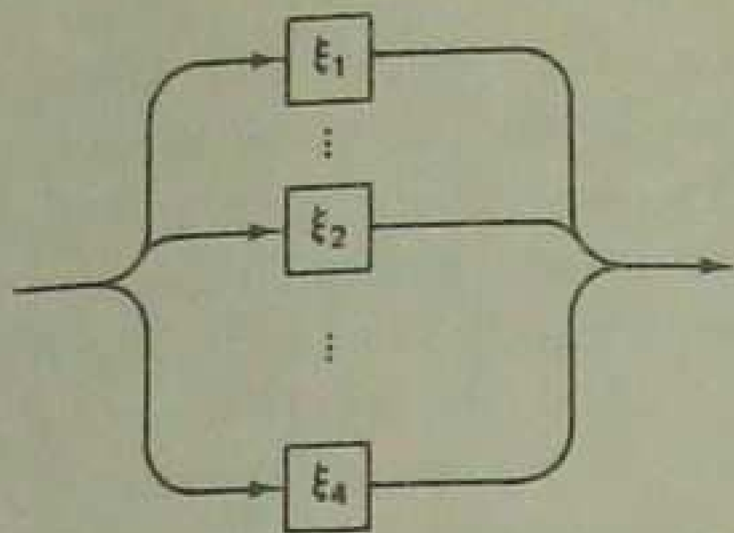


A3. Každý výskyt neterminálneho symbolu B v ξ_i zodpovedá volaniu programu pre analýzu B. V grafe to zobrazujeme takto:

A4. Prepisovacie pravidlo tvaru

$$A ::= \xi_1 | \ldots | \xi_n$$

sa zobrazí na graf,



v ktorom sa každý reťazec $\boxed{\xi_i}$ získa aplikáciou konštrukčných zásad A2 až A6 na reťazec ξ_i.

A5. Reťazec ξ tvaru

$$\xi = \alpha_1 \alpha_2 \ldots \alpha_m$$

sa zobrazí na graf

v ktorom sa každá $\boxed{\alpha_i}$ získa aplikáciou konštrukčných zásad A2 až A6 na symbol α_i.

A6. Reťazec ξ tvaru

$$\xi = \{\alpha\}$$

sa zobrazí na graf

v ktorom sa $\boxed{\alpha}$ získa aplikáciou konštrukčných zásad A2 až A6 na symbol α.

Príklad 5.

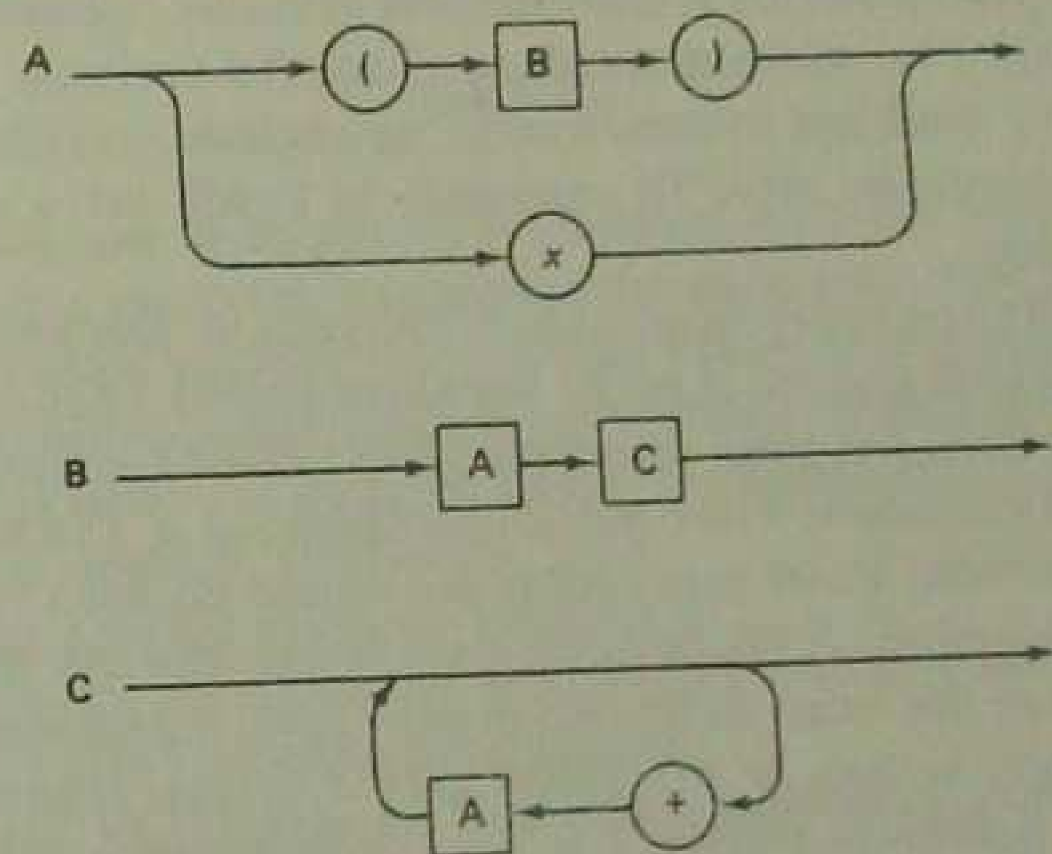
$$\begin{aligned} A &::= x \mid (B) \\ B &::= AC \\ C &::= \{+A\} \end{aligned} \tag{5.7}$$

Symboly +, x, (,) predstavujú terminálne symboly. Zložené zátvorky {, } patria do rozšírenej syntaxe BNF, teda sú to metasymboly. Jazyk

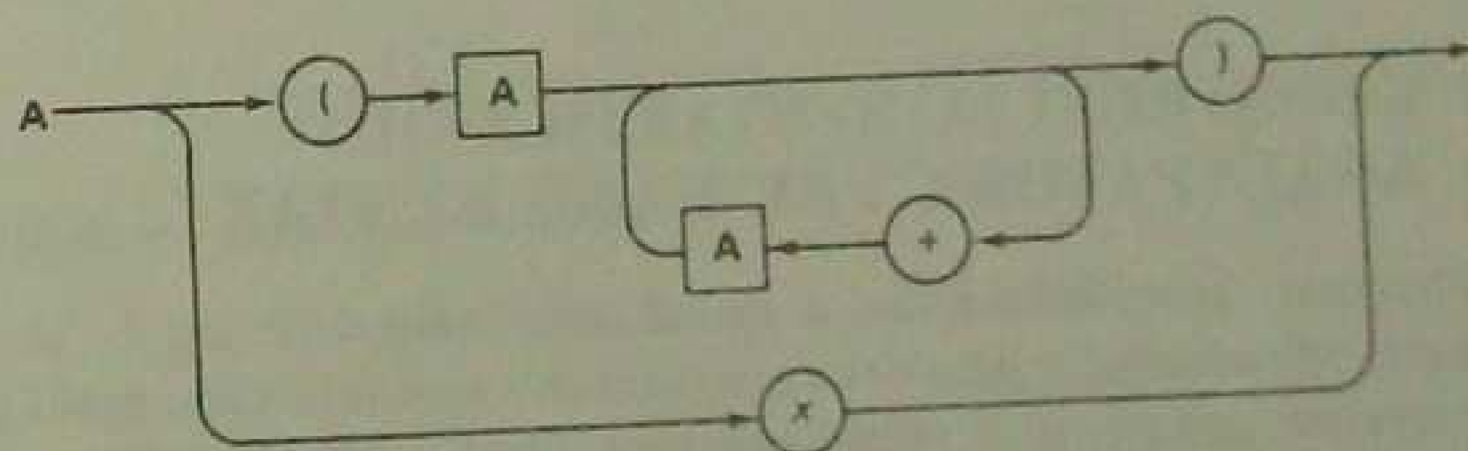
generovateľný z A pozostáva z výrazov, ktoré sa skladajú z operandov x, operátora + a zátvoriek. Príkladmi viet sú

$$\begin{gathered} x \\ (x) \\ (x + x) \\ ((x)) \\ \vdots \end{gathered}$$

Grafy, ktoré získame aplikáciou šiestich konštrukčných zásad, sú znázornené na *obr. 5.1*. Všimnime si, že vhodnou substitúciou C na B a B na A môžeme tento systém grafov nahradiť jediným grafom (*obr. 5.2*).



Obr. 5.1. Syntaktické grafy k príkladu 5



Obr. 5.2. Redukovaný syntaktický graf k príkladu 5

Syntaktický graf je ekvivalentnou reprezentáciou gramatiky jazyka. Môžeme ho použiť namiesto množiny prepisovacích pravidiel vyjadrených v jazyku BNF. Je to veľmi vhodná forma a v mnohých (ak nie vo všetkých) prípadoch uprednostňovaná pred BNF. Určite poskytuje jasnejší a stručnejší obraz o jazykovej štruktúre a navyše umožňuje oveľa rýchlejšie pochopiť proces syntaktickej analýzy. Graf je pre tvorcov jazyka vhodná forma v procese jeho návrhu. Príklady syntaktických špecifikácií úplných jazykov sú uvedené v článku 5.7 (pre jazyk PL/0) a v prílohe B (pre jazyk pascal).

Obmedzujúce tvrdenia 1 a 2 sme zaviedli preto, aby sme umožnili deterministickú syntaktickú analýzu s predsnímaním jedného symbolu dopredu. Ako sa tieto tvrdenia prejavia v syntaktickom grafe? So zreteľom na prehľadnosť a zrozumiteľnosť grafovej reprezentácie takto:

1. Tvrdenie 1 sa interpretuje ako požiadavka vetvenia grafu: výber vhodnej vetvy musí byť uskutočniteľný na základe prezretia najbližšieho ďalšieho symbolu vety. To znamená, že každá vetva musí začínať iným symbolom.

2. Tvrdenie 2 sa interpretuje ako požiadavka na prechod grafom: ak je možné nejakým grafom *A* prejsť bez akéhokoľvek načítania vstupného symbolu, musí byť táto „prázdna vetva" charakterizovaná množinou všetkých symbolov, ktoré môžu nasledovať za *A*. (To bude ovplyvňovať rozhodnutie, ktoré treba robiť pri vstupe do tejto vetvy.)

Nie je ťažké overiť, či systém grafov spĺňa uvedené dve prispôsobené pravidlá, a to aj bez toho, že by sme použili BNF reprezentáciu gramatiky. Pomôžeme si tým, že pre každý graf *A* určíme množiny first(*A*) a follow(*A*). Aplikácia tvrdení 1 a 2 je potom bezprostredná. Systém grafov, ktorý spĺňa uvedené dve tvrdenia, nazývame *deterministickým syntaktickým grafom*.

5.4 KONŠTRUKCIA SYNTAKTICKÉHO ANALYZÁTORA PRE DANÚ SYNTAX

Program, ktorý akceptuje a syntakticky analyzuje jazyk, je bez ťažkostí odvoditeľný z jeho deterministického syntaktického grafu (ak, samozrejme, takýto graf vôbec existuje). Graf v zásade reprezentuje

vývojový diagram programu. Pri vývoji takéhoto programu sa však odporúča postupovať presne podľa danej množiny pravidiel podobných pravidlám, ktoré usmerňujú prechod od BNF ku grafovej reprezentácii syntaxe jazyka. (Tieto pravidlá sú uvedené v ďalšom.) Sú použiteľné v špecifických prípadoch, ktoré môžu byť reprezentované hlavným programom, do ktorého sú začlenené procedúry zodpovedajúce rôznym podcieľom a procedúra umožňujúca prístup k ďalšiemu symbolu.

Pre jednoduchosť predpokladajme, že veta, ktorú máme analyzovať, je reprezentovaná súborom input a terminálnymi symbolmi sú jednotlivé znaky. Požadujeme, aby existovala premenná typu znak reprezentujúca ďalší načítaný symbol. Načítanie ďalšieho symbolu môžeme vyjadriť príkazom

read (*ch*)

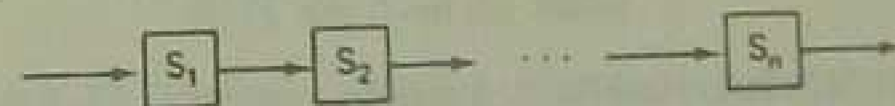
Hlavný program bude potom pozostávať zo začiatočného príkazu načítania prvého symbolu, nasledovaného príkazom, ktorý zaháji celý proces syntaktickej analýzy (čo je naším hlavným cieľom). Jednotlivé procedúry zodpovedajúce príslušným cieľom analýzy alebo grafom získame na základe nasledujúcich zásad. Označme príkaz, ktorý získame prekladom grafu *S*, symbolom $T(S)$.

Zásady pre preklad grafu do programu:

B1. Vhodnými substitúciami zredukovať systém grafov na čo najväčší možný počet individuálnych grafov.

B2. Preložiť každý graf do deklarácie procedúry podľa zásad B3 až B7.

B3. Postupnosť prvkov



sa preloží do zloženého príkazu

begin $T(S_1)$; $T(S_2)$; ...; $T(S_n)$ **end**

B4. Výber prvkov

Syntaktický graf je ekvivalentnou reprezentáciou gramatiky jazyka. Môžeme ho použiť namiesto množiny prepisovacích pravidiel vyjadrených v jazyku BNF. Je to veľmi vhodná forma a v mnohých (ak nie vo všetkých) prípadoch uprednostňovaná pred BNF. Určite poskytuje jasnejší a stručnejší obraz o jazykovej štruktúre a navyše umožňuje oveľa rýchlejšie pochopiť proces syntaktickej analýzy. Graf je pre tvorcov jazyka vhodná forma v procese jeho návrhu. Príklady syntaktických špecifikácií úplných jazykov sú uvedené v článku 5.7 (pre jazyk PL/0) a v prílohe B (pre jazyk pascal).

Obmedzujúce tvrdenia 1 a 2 sme zaviedli preto, aby sme umožnili deterministickú syntaktickú analýzu s predsnímaním jedného symbolu dopredu. Ako sa tieto tvrdenia prejavia v syntaktickom grafe? So zreteľom na prehľadnosť a zrozumiteľnosť grafovej reprezentácie takto:

1. Tvrdenie 1 sa interpretuje ako požiadavka vetvenia grafu: výber vhodnej vetvy musí byť uskutočniteľný na základe prezretia najbližšieho ďalšieho symbolu vety. To znamená, že každá vetva musí začínať iným symbolom.

2. Tvrdenie 2 sa interpretuje ako požiadavka na prechod grafom: ak je možné nejakým grafom *A* prejsť bez akéhokoľvek načítania vstupného symbolu, musí byť táto „prázdna vetva" charakterizovaná množinou všetkých symbolov, ktoré môžu nasledovať za *A*. (To bude ovplyvňovať rozhodnutie, ktoré treba robiť pri vstupe do tejto vetvy.)

Nie je ťažké overiť, či systém grafov spĺňa uvedené dve prispôsobené pravidlá, a to aj bez toho, že by sme použili BNF reprezentáciu gramatiky. Pomôžeme si tým, že pre každý graf *A* určíme množiny first(*A*) a follow(*A*). Aplikácia tvrdení 1 a 2 je potom bezprostredná. Systém grafov, ktorý spĺňa uvedené dve tvrdenia, nazývame *deterministickým syntaktickým grafom*.

5.4 KONŠTRUKCIA SYNTAKTICKÉHO ANALYZÁTORA PRE DANÚ SYNTAX

Program, ktorý akceptuje a syntakticky analyzuje jazyk, je bez ťažkostí odvoditeľný z jeho deterministického syntaktického grafu (ak, samozrejme, takýto graf vôbec existuje). Graf v zásade reprezentuje

vývojový diagram programu. Pri vývoji takéhoto programu sa však odporúča postupovať presne podľa danej množiny pravidiel podobných pravidlám, ktoré usmerňujú prechod od BNF ku grafovej reprezentácii syntaxe jazyka. (Tieto pravidlá sú uvedené v ďalšom.) Sú použiteľné v špecifických prípadoch, ktoré môžu byť reprezentované hlavným programom, do ktorého sú začlenené procedúry zodpovedajúce rôznym podcieľom a procedúra umožňujúca prístup k ďalšiemu symbolu.

Pre jednoduchosť predpokladajme, že veta, ktorú máme analyzovať, je reprezentovaná súborom input a terminálnymi symbolmi sú jednotlivé znaky. Požadujeme, aby existovala premenná typu znak reprezentujúca ďalší načítaný symbol. Načítanie ďalšieho symbolu môžeme vyjadriť príkazom

read (*ch*)

Hlavný program bude potom pozostávať zo začiatočného príkazu načítania prvého symbolu, nasledovaného príkazom, ktorý zaháji celý proces syntaktickej analýzy (čo je naším hlavným cieľom). Jednotlivé procedúry zodpovedajúce príslušným cieľom analýzy alebo grafom získame na základe nasledujúcich zásad. Označme príkaz, ktorý získame prekladom grafu *S*, symbolom $T(S)$.

Zásady pre preklad grafu do programu:

B1. Vhodnými substitúciami zredukovať systém grafov na čo najväčší možný počet individuálnych grafov.

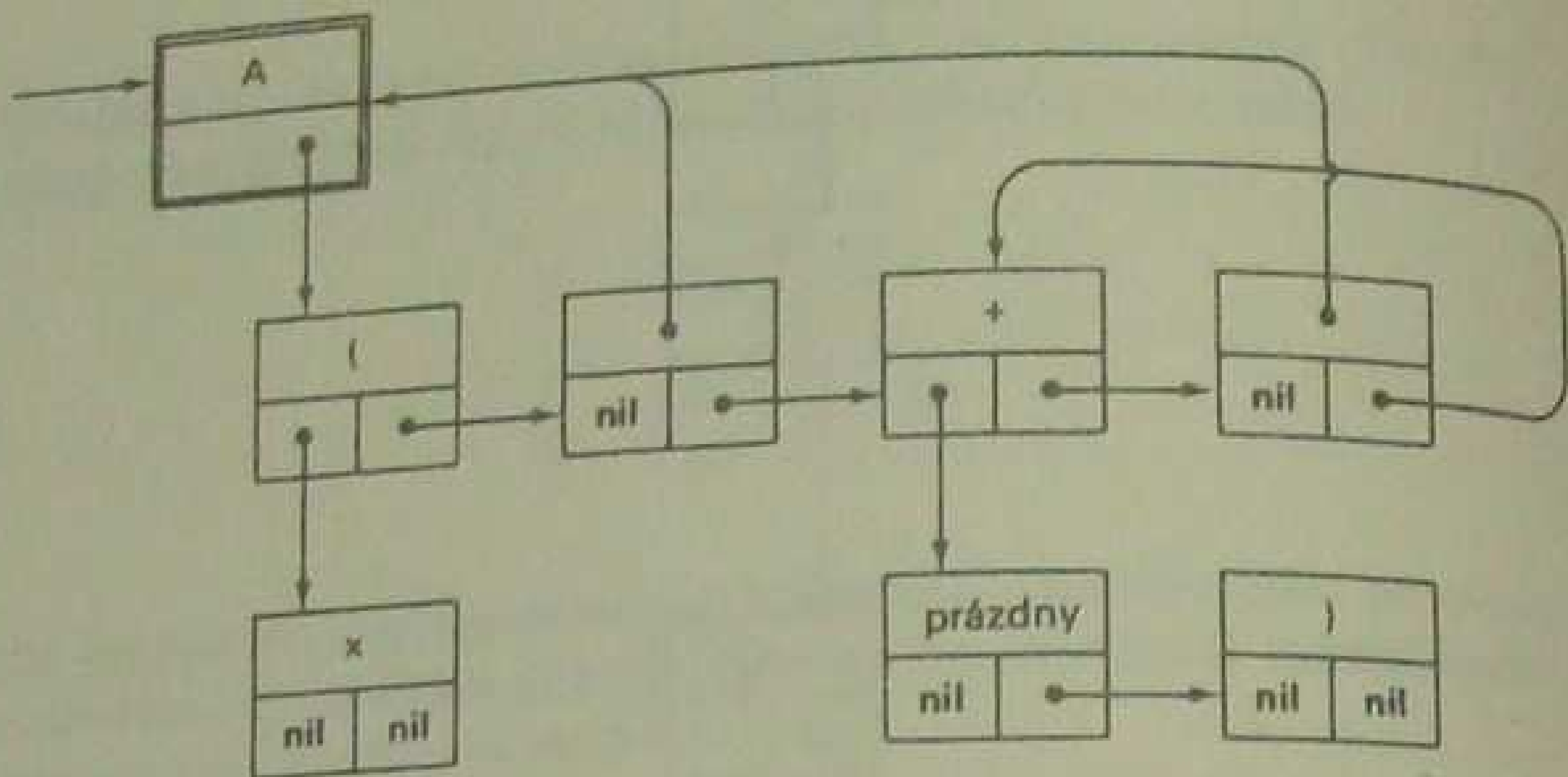
B2. Preložiť každý graf do deklarácie procedúry podľa zásad B3 až B7.

B3. Postupnosť prvkov

sa preloží do zloženého príkazu

begin $T(S_1)$; $T(S_2)$; ...; $T(S_n)$ **end**

B4. Výber prvkov



Obr. 5.3. Štruktúra údajov reprezentujúca graf z *obr. 5.2*

Program na analýzu vety reprezentovanej reťazcom znakov vstupného súboru obsahuje príkaz cyklu, ktorý opisuje prechod od jedného vrcholu k ďalšiemu. Program je vyjadrený procedúrou opisujúcou interpretáciu grafu; ak sa pri nej narazí na vrchol, reprezentujúci neterminálny symbol, vykoná sa najprv interpretácia tohto grafu a až potom sa dovŕši interpretácia rozpracovaného grafu. Vidíme, že interpretačná procedúra je volaná rekurzívne. Ak sa bežný symbol (*sym*) vstupného súboru zhoduje so symbolom v momentálnom vrchole štruktúry údajov, tak ďalší krok analýzy je určený zložkou *suc*, v opačnom prípade zložkou *alt*.

```
procedure analyzátor (cieľ: čsmerník; var zhodný: boolean);
  var s: smerník;
begin s: = cieľ↑.vstup;
  repeat
    if s↑.terminál then
    begin if s↑.tsym = sym then
        begin zhodný: = true; getsym                          (5.11)
        end
        else zhodný: = (s↑.tsym = prázdny)
    end
    else analyzátor (s↑.nsym, zhodný);
    if zhodný then s: = s↑.suc else s: = s↑.alt
  until s = nil
end
```

Tento program má tú vlastnosť, že len čo sa určí nový podcieľ analýzy *G*, hneď ho začne „sledovať" bez ohľadu na to, či sa momentálny symbol nachádza v množine začiatočných symbolov first(*G*) alebo nie. To znamená, že zodpovedajúci syntaktický graf musí byť zbavený výberov rôznych alternatívnych neterminálnych symbolov. Presnejšie, ak je neterminálny symbol schopný generovať prázdny reťazec, žiadna z jeho pravých časti nesmie začínať neterminálnym symbolom.

Z uvedenej schémy (5.11) sa dajú odvodiť oveľa dômyselnejšie syntaktické analyzátory riadené tabuľkou, pracujúce s menej obmedzenými triedami gramatík. Nepatrnými úpravami algoritmu sa dá dosiahnuť aj prehľadávanie s návratom, ale za cenu zmenšenia celkovej efektívnosti.

Grafová reprezentácia syntaxe má jednu závažnú nevýhodu: počítače nedokážu priamo čítať grafy. Preto štruktúry údajov, ktoré riadia syntaktický analyzátor, sa musia vytvoriť predtým, než sa zaháji proces samotnej analýzy. V tomto zmysle je reprezentácia gramatík pomocou BNF ideálnou formou vstupu pre všeobecný program syntaktickej analýzy. Ďalší článok je preto venovaný návrhu programu, ktorý číta postupnosť prepisovacích pravidiel a prekladá ich podľa zásad B1 až B6 do vnútorných štruktúr, s ktorými dokáže analyzátor (5.11) pracovať [5-8].

5.6 PREKLADAČ Z BNF DO ŠTRUKTÚR ÚDAJOV

Prekladač, ktorý na vstupe pripúšťa prepisovacie pravidlá BNF a tieto mení na inú reprezentáciu, je skutočným príkladom programu, ktorého vstupné údaje predstavujú vety nejakého jazyka. Je naozaj rozumné pokladať BNF za jazyk charakteristický svojou syntaxou,

ktorá môže byť opäť vyjadrená prostredníctvom prepisovacích pravidiel BNF. V dôsledku toho môže tento prekladač slúžiť ako ďalší príklad konštrukcie analyzátora, ktorý je navyše rozšírený na procesor svojho vstupu. Budeme preto postupovať takýmto spôsobom:

Krok 1. Budeme definovať syntax metajazyka nazývaného EBNF (rozšírený jazyk BNF).

Krok 2. Zostrojíme analyzátor EBNF podľa zásad uvedených v článku 5.4.

Krok 3. V kombinácii so syntaktickým analyzátorom riadeným tabuľkou rozšírime tento analyzátor na prekladač.

Nech je metajazyk — jazyk syntaktických prepisovacích pravidiel — opísaný týmito pravidlami:

$$\begin{aligned} \langle\text{pravidlo}\rangle &::= \langle\text{symbol}\rangle = \langle\text{výraz}\rangle \\ \langle\text{výraz}\rangle &::= \langle\text{term}\rangle \{, \langle\text{term}\rangle\} \\ \langle\text{term}\rangle &::= \langle\text{faktor}\rangle \{\langle\text{faktor}\rangle\} \\ \langle\text{faktor}\rangle &::= \langle\text{symbol}\rangle \mid [\langle\text{term}\rangle] \end{aligned} \tag{5.12}$$

Všimnime si, že symboly, ktoré sú odlišné oproti obvyklým metasymbolom BNF, sme použili na označenie práve týchto symbolov v prepisovacích pravidlách vstupného jazyka. Existujú na to dva dôvody:

1. Rozlíšiť metasymboly od jazykových symbolov v (5.12).
2. Použiť všeobecne dostupné znaky počítačového systému, najmä jednoduchého znaku = namiesto ::=.

Tab. 5.1 zobrazuje zodpovedajúce si symboly obvyklého jazyka BNF a nášho rozšíreného vstupného jazyka EBNF. Každé prepisovacie pravidlo je navyše ukončené explicitnou bodkou.

Metajazykové a jazykové symboly *Tabuľka 5.1*

BNF	Vstupný EBNF
::=	=
|	,
{	[
}	]

Použitím tohto vstupného jazyka na opis syntaxe príkladu 5 (5.7) dostávame

$$\begin{aligned} A &= x, (B). \\ B &= AC. \\ C &= [+A]. \end{aligned} \tag{5.13}$$

Pri zjednodušovaní vytváraného prekladača budeme požadovať, aby terminálne symboly boli jednoduchými písmenami a každé prepisovacie pravidlo bolo napísané na osobitnom riadku. To nám umožní používať medzery vo vstupnom texte (čím sa tento text stane čitateľnejším). Medzery však prekladač ignoruje. V dôsledku toho sa musí príkaz read (*ch*) v zásade B7 nahradiť volaním procedúry, ktorá načíta najbližší relevantný znak. Táto činnosť prináleží lexikálnemu analyzátoru, ktorého úlohou je určiť ďalší symbol — v súlade s definovanými jazykovými pravidlami — zo vstupného reťazca znakov. Doteraz sme uvažovali o tom, že symboly sú identické so znakmi; to je však len špeciálny prípad a v praxi dosť zriedkavý.

Posledná zásada, ktorá sa bude týkať vstupu BNF, bude požadovať, aby neterminálne symboly boli reprezentované písmenami *A* až *H* a terminálne symboly písmenami *I* až *Z*. Túto zásadu však použijeme iba preto, že je výhodná, inak nemá žiadne hlbšie opodstatnenie. Jej uplatnením nepotrebujeme napr. vytvárať slovníky terminálnych a neterminálnych symbolov pred vlastným zoznamom produkcií.

Program 5.2 predstavuje syntaktický analyzátor jazyka, definovaného produkciami (5.12), ktorý sme získali na základe zásad B1 až B7 konštrukcie analyzátora a overením, či definície (5.12) spĺňajú obmedzenia 1 a 2. Poznamenávame, že lexikálny analyzátor predstavuje procedúra getsym.

PROGRAM 5.2. *Syntaktický analyzátor jazyka (5.12)*

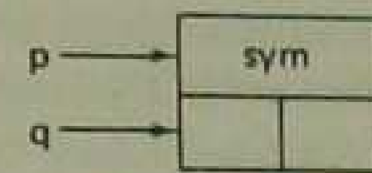
```
program Analyzátor (input, output);
label 99;
const prázdny = '*';
var sym: char;
procedure getsym;
begin
```

```
  repeat read (sym); write (sym) until sym ≠ ' '
end {getsym};
procedure error;
begin writeln;
    writeln ('NESPRÁVNY VSTUP'); goto 99
end {error};
procedure term;
    procedure factor;
    begin
      if sym in ['A' .. 'Z', prázdny] then getsym else
      if sym = '[' then
      begin getsym; term;
        if sym = ']' then getsym else error
      end else error
    end {factor};
begin factor;
    while sym in ['A' .. 'Z', '[', prázdny] do factor
end {term};
procedure výraz;
begin term;
    while sym = ',' do
    begin getsym; term
    end
end {výraz};
  begin {hlavný program}
    while ¬ eof(input) do
    begin getsym;
      if sym in ['A' .. 'Z'] then getsym else error;
      if sym = '=' then getsym else error;
      výraz;
      if sym ≠ '.' then error;
      writeln; readln;
    end;
  99: end.
```

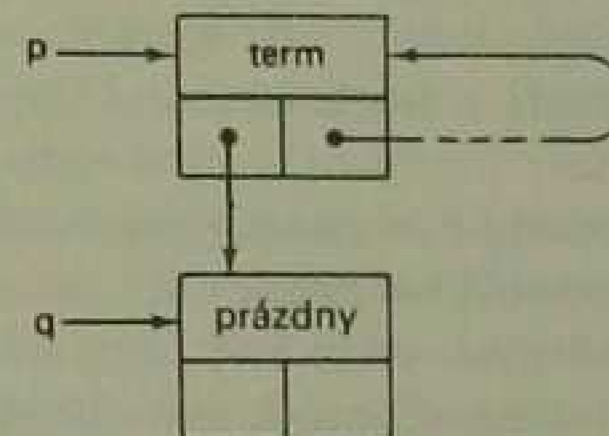
Tretí krok tvorby prekladača sa týka konštrukcie požadovanej štruktúry údajov, ktorá reprezentuje práve prečítané prepisovacie pravidlá BNF a umožňuje ich interpretáciu prostredníctvom procedúry (5.11). Žiaľ, tento krok sa nedá až tak formalizovať, ako to bolo v prípade druhého kroku, týkajúceho sa tvorby analyzátora EBNF. Pretože nám chýba formalizmus, uvedieme ešte raz (vo forme obrázku) štruktúry, ktoré sú potrebné na reprezentáciu každej jazykovej konštrukcie. Tieto štruktúry sa potom odovzdávajú vo forme výstupných parametrov zodpovedajúcim procedúram analyzátora (vylepšených na procedúry prekladača). Pochopiteľne, že návratom nie sú samotné štruktúry, ale iba smerníky p, q, r na tieto štruktúry.

Faktory:

1. ⟨symbol⟩

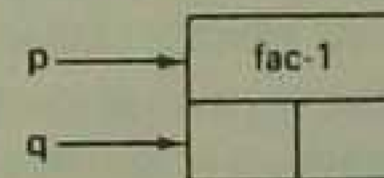


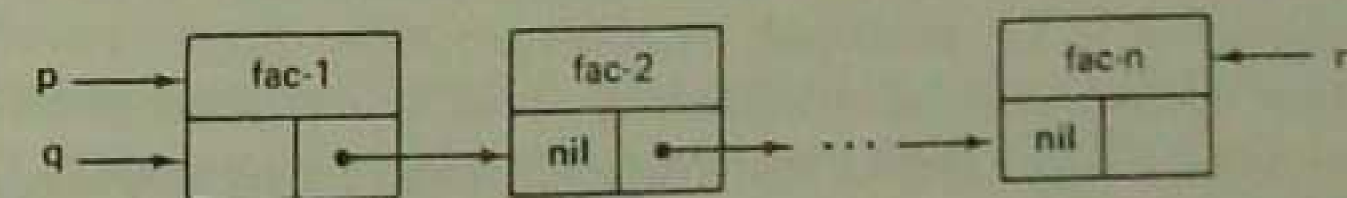
2. [⟨term⟩]



Termy:

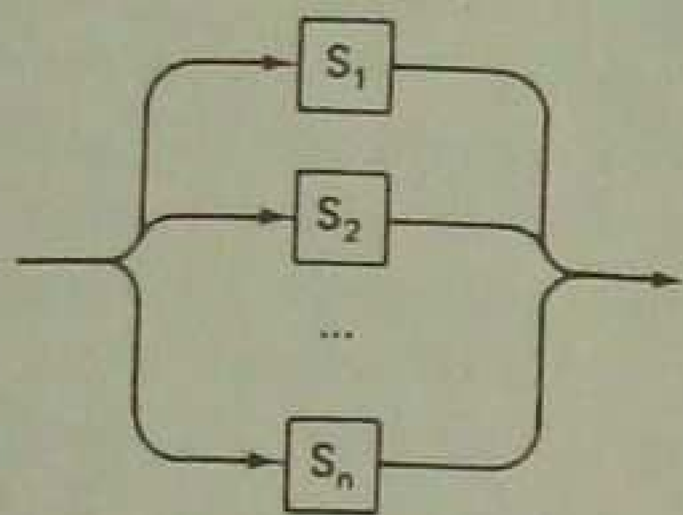
⟨faktor – 1⟩ ... ⟨faktor – n⟩





Výrazy:

⟨term – 1⟩⟨term – 2⟩ ... ⟨term – n⟩



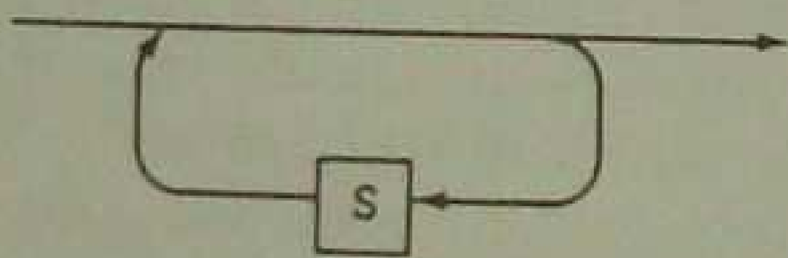
sa preloží buď na príkaz výberu, alebo na podmienený príkaz

case *ch* **of**	**if** *ch* **in** L_1 **then** $T(S_1)$ **else**
L_1: $T(S_1)$;	**if** *ch* **in** L_2 **then** $T(S_2)$ **else**
L_2: $T(S_2)$;	⋮
⋮	**if** *ch* **in** L_n **then** $T(S_n)$ **else**
L_n: $T(S_n)$	error
end	

kde L_i označuje množinu začiatočných symbolov konštrukcie S_i (L_i = = first (S_i)).

Poznámka: Ak L_i pozostáva z jednoduchého symbolu *a*, tak možno výraz *ch* **in** L_i vyjadriť ako *ch* = *a*.

B5. Cyklus v tvare



sa preloží na príkaz

while *ch* **in** *L* **do** $T(S)$

kde $T(S)$ je výsledok prekladu *S* podľa zásad B3 až B7 a *L* je množina *L* = first (*S*) (pozri predchádzajúcu poznámku).

B6. Prvok grafu, ukazujúci na iný graf *A*

sa preloží na príkaz vyvolania procedúry *A*.

B7. Prvok grafu, ukazujúci na terminálny symbol *x*,

sa preloží na príkaz

if *ch* = *x* **then** read (*ch*) **else** error

kde error predstavuje procedúru, ktorá sa vyvoláva v prípade výskytu chybnej syntaktickej konštrukcie.

Použitie uvedených zásad si môžeme ukázať na príklade prekladu redukovaného grafu (príklad 5, *obr. 5.2*) na program syntaktického analyzátora (program 5.1).

PROGRAM 5.1. *Program syntaktického analyzátora pre gramatiku z príkladu 5*

```
program Analyzátor (input, output);
  var ch: char;
  procedure A;
  begin if ch = 'x' then read (ch) else
        if ch = '(' then
        begin read (ch); A;
           while ch = '+' do
              begin read (ch); A
              end;
           if ch = ')' then read (ch) else error
        end else error
  end;
 begin read (ch); A
 end.
```

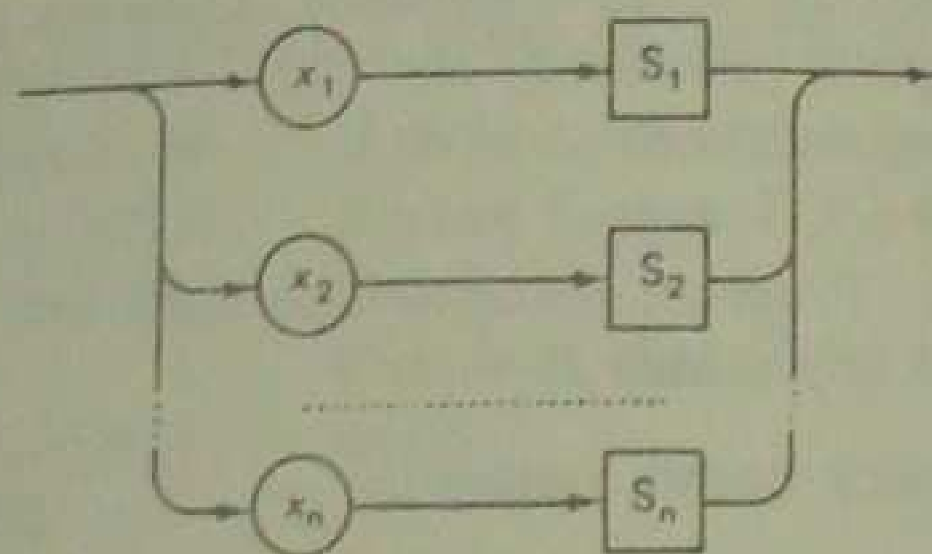
V uvedenom preklade sme využili niektoré zaužívané programovacie zásady, ktoré zjednodušujú program. Preklad literálu by napr. mohol vyústiť do takejto podoby:

```
if ch = 'x' then
   if ch = 'x' then read (ch) else error
   else ...
```

Použitím spomenutých programovacích zásad sme získali jednoduchšiu formu uvedenú v programe. Príkazy čítania read v piatom a siedmom riadku programu sú výsledkom podobných redukcií.

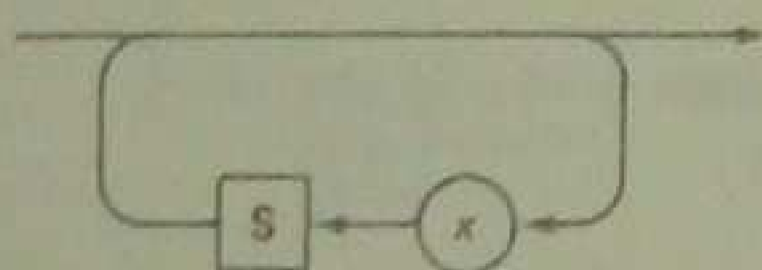
Zdá sa, že by bolo rozumné zistiť všetky prípady, kde vo všeobecnosti sú takéto redukcie možné, a tieto potom vyjadriť priamo pomocou grafov. Nasledujúce dve dodatočné zásady znázorňujú dva relevantné prípady:

B4a.




```
if ch = 'x1' then begin read(ch); T(S1) end else
if ch = 'x2' then begin read(ch); T(S2) end else
   ...
if ch = 'xn' then begin read(ch); T(Sn) end else error
```

B5a.




```
while ch = 'x' do
   begin read(ch); T(S) end
```

Navyše často sa vyskytujúcu konštrukciu

```
read(ch); T(S);
while B do
   begin read(ch); T(S) end
```

môžeme vyjadriť v zjednodušenom tvare


```
repeat read(ch); T(S) until B                    (5.8)
```

Ako ste si už iste všimli, nezaoberali sme sa doteraz podrobnejšie procedúrou error. Pretože teraz nás zaujíma iba to, či je daný vstupný reťazec syntakticky správny alebo nie, môžeme túto procedúru pokladať za ukončenie realizácie programu.

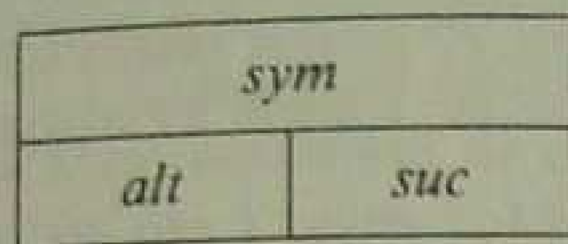
Prirodzene, v praxi sa používajú oveľa dômyselnejšie prostriedky spracúvania syntaktických chýb. Tieto budú predmetom článku 5.9.

5.5 KONŠTRUKCIA PROGRAMU SYNTAKTICKEJ ANALÝZY RIADENÉHO TABUĽKOU

Namiesto zostrojenia špecifického programu syntaktickej analýzy pre každý jazyk a jeho syntax, podľa pravidiel uvedených v predchádzajúcej kapitole, je možné vytvoriť jednoduchý, všeobecný program syntaktickej analýzy. Gramatiky jednotlivých jazykov predstavujú začiatočné údaje pre takýto všeobecný analyzátor a poskytnú sa mu ešte pred samotnými vetami, ktoré sa majú analyzovať. Všeobecný program sa dôsledne riadi pravidlami metódy jednoduchej syntaktickej analýzy zhora nadol. Je priamočiary, ak je príslušný syntaktický graf deterministický, t. j. zodpovedajúca gramatika umožňuje analýzu viet s predsnímaním jedného symbolu dopredu bez návratu.

Gramatika, o ktorej predpokladáme, že je reprezentovaná v tvare deterministickej množiny syntaktických grafov, sa teda prekladá do príslušnej štruktúry údajov, a nie do programovej štruktúry [5-2]. Prirodzený spôsob reprezentácie grafu spočíva v zavedení vrcholu pre každý symbol a v pospájaní takýchto vrcholov smerníkmi. Preto tabuľka už nie je iba jednoduchým poľom. Pravidlá usmerňujúce preklad sú uvedené ďalej a sú samozrejmé. Vrcholy grafu sú reprezentované štruktúrou záznam s dvoma variantmi: jedným pre terminálne symboly a druhým pre neterminálne symboly. Prvý variant je identifikovateľný tým symbolom, ktorý zastupuje, druhý variant smerníkom na štruktúru údajov reprezentujúcu príslušný neterminálny symbol. Obidva varianty obsahujú dva smerníky, z ktorých prvý ukazuje na nasledujúci symbol (t. j. na nasledovníka) a druhý na zoznam možných alternatív.

Výsledné definície typov vrchol a smerník zobrazuje schéma (5.9). Pri zobrazovaní grafovej reprezentácie bude vrchol vyzerať takto:



Ako sa ukazuje, budeme potrebovať aj prvok, reprezentujúci prázdnu postupnosť, t.j. prázdny symbol. Budeme ho označovať terminálnym prvkom, ktorý nazývame *prázdny*.

```
type smerník = ↑vrchol;
    vrchol = record suc, alt: smerník;
        case terminál: boolean of
            true: (tsym: char);                    (5.9)
            false: (nsym: psmerník)
        end
```

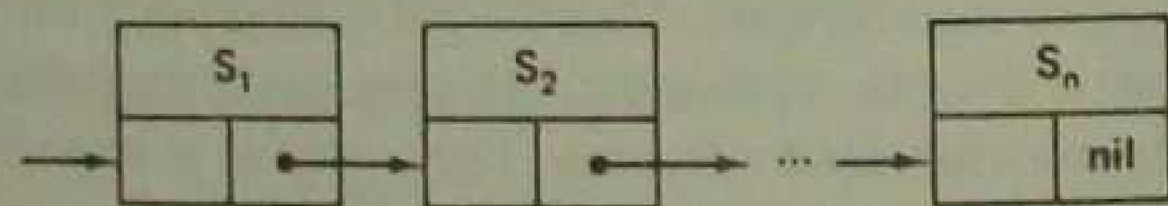
Zásady prekladu z grafov do štruktúr údajov sú analogické ako zásady B1 až B7.

Zásady prekladu grafov do štruktúr údajov:

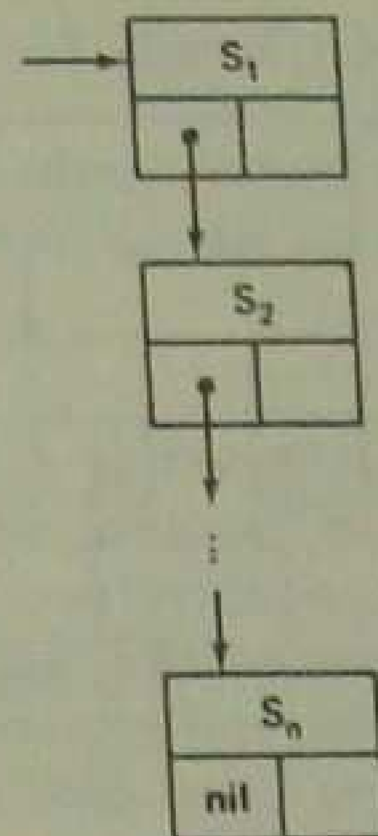
C1. Vhodnými substitúciami zredukuj systém grafov na čo najväčší možný počet individuálnych grafov.

C2. Prelož každý graf do štruktúry údajov podľa zásad C3 až C5.

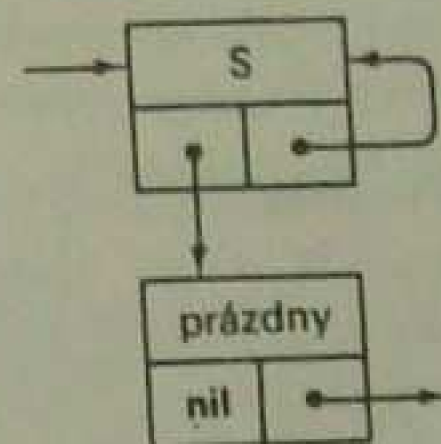
C3. Postupnosť prvkov (pozri obrázok pre zásadu B3) sa preloží na nasledujúci zoznam vrcholov:



C4. Zoznam alternatív (pozri obrázok pre zásadu B4) sa preloží na štruktúru:



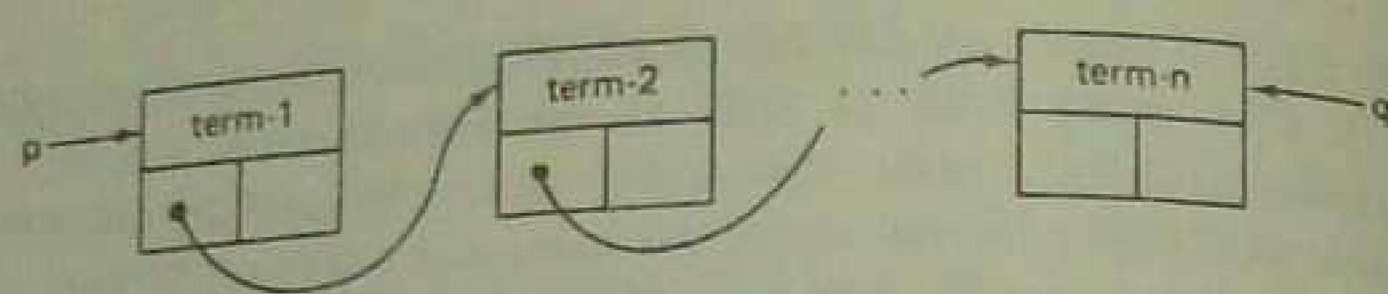
C5. Cyklus (pozri obrázok pre zásadu B5) sa preloží na štruktúru:



Ako príklad nám môže poslúžiť štruktúra na *obr. 5.3* reprezentujúca graf, ktorý zodpovedá syntaxi z príkladu 5 (*obr. 5.2*).

Štruktúra údajov je identifikovateľná prostredníctvom vrcholu nazývaného *čelo*, ktorý obsahuje meno neterminálneho symbolu (*cieľ*) reprezentovaného danou štruktúrou. Tento vrchol je v podstate nepotrebný, pretože smerník ukazujúci naň môže vlastne ukazovať priamo na začiatok príslušnej štruktúry. Môžeme ho však využiť v tom zmysle, že by obsahoval názov patričnej štruktúry.

```
type čsmerník = ↑čelo;
    čelo = record vstup: smerník;                  (5.10)
                  sym: char
           end
```



Zrejme, že generovanie nových prvkov štruktúry údajov je úlohou procedúry faktor; úlohou zvyšných dvoch procedúr je pospájať ich do lineárneho zoznamu, v ktorom term používa na zreťazenie zložku *suc* a výraz zložku *alt*. Podrobnosti sú vidieť v programe 5.3.

Spôsob spracúvania neterminálnych symbolov vyžaduje ďalšie vysvetlenie. Neterminálny symbol sa môže vyskytnúť ako faktor predtým, než sa zjaví ako ľavá časť prepisovacieho pravidla. Na vyhľadanie symbolu *sym* v lineárnom zozname všetkých neterminálnych symbolov sa používa procedúra nájdi (*sym*, *h*). Ak sa nájde v zozname príslušný symbol, priradí sa smerník naň parametru *h*; v opačnom prípade sa symbol *sym* pridá do zoznamu. Procedúra nájdi využíva techniku zarážky, o ktorej sme písali vo štvrtej kapitole.

Program 5.3 sa skladá z troch častí, z ktorých každá zodpovedá príslušnej vstupnej sekcii. Prvá časť sa týka spracovania prepisovacích pravidiel a ich transformácie do príslušných štruktúr údajov. V druhej časti sa číta a identifikuje symbol definovaný ako začiatočný, generujúci vety jazyka. (Predchádza mu znak $, ktorým sú oddelené časti 1 a 2 vstupných údajov.) Tretia časť predstavuje vlastný program syntaktickej analýzy (5.11), ktorý riadený štruktúrou údajov, vygenerovanou v prvej časti, číta a analyzuje vstupné vety.

Je pozoruhodné, že program 5.3 vznikol iba pridaním ďalších príkazov do nezmeneného programu 5.2. Program 5.2, ktorého cieľom je výlučne rozpoznanie správne vytvorených viet, sa dá použiť aj v širšom kontexte, t.j. v programe, ktorý nielen rozpoznáva, ale aj spracúva a prekladá správne vytvorené vety. Takáto metóda konštrukcie jazykových procesorov, t.j. postupné zjemňovanie alebo (lepšie povedané) postupné rozširovanie (obohacovanie), sa veľmi odporúča. Umožňuje tvorcovi kompilátora jazyka venovať sa v každej vývojovej etape výlučne vybranému okruhu problémov spracovania jazyka (pričom sa abstrahuje od nepodstatných detailov). Tým sa súčasne podstatne zjednoduší proces overovania správnosti prekladacieho programu, alebo sa

aspoň udrží značný stupeň dôvery v úspešný priebeh vývoja prekladacieho programu. Tento príklad prekladača možno považovať za jednoduchý, pretože jeho vývoj pozostával iba z dvoch krokov. Pochopiteľne, zložitejšie jazyky a zložitejšie preklady vyžadujú podstatne väčší počet jednotlivých rozširovacích krokov. V článkoch 5.8 až 5.11 sa budeme zaoberať veľmi podobným vývojom prekladača, ktorý však vyžaduje tri kroky.

Z vývoja programu 5.3 vidíme, že syntaxou riadený preklad, alebo skôr štruktúrou riadený preklad, poskytuje oveľa väčší stupeň voľnosti a flexibility ako program pre špecifický syntaktický analyzátor. Táto dodatočná flexibilita, aj keď sa vo všeobecnosti nepožaduje, je základnou charakteristikou kompilátora rozšíriteľných jazykov. Rozšíriteľný jazyk sa dá rozšíriť o ďalšie syntaktické konštrukcie viac-menej podľa uváženia programátora. Podobne ako vstup pre program 5.3 bude vstup kompilátora rozšíriteľného jazyka obsahovať sekciu definujúcu jazykové rozšírenia použité v nasledujúcom programe. Náročnejšia schéma dokonca umožňuje zmeny v jazyku počas procesu kompilácie, a to na základe vkladania sekcií s novými jazykovými špecifikáciami do prekladaného programu.

Aj keď sa takéto myšlienky zdajú byť príťažlivé, snahy realizovať takéto kompilátory boli málo úspešné. Dôvodom je skutočnosť, že syntaktická analýza vety je len malou časťou celého prekladacieho procesu a dá sa najjednoduchšie formalizovať, a tým aj ľahko reprezentovať prostredníctvom systematizovanej tabuľkovej štruktúry. Náročnejšou časťou na formalizáciu je význam jazyka, t.j. vstup alebo výsledok prekladu. Tento problém doteraz nebol uspokojivo vyriešený, čím sa vysvetľuje to, prečo zvyknú byť tvorcovia kompilátorov nadšení rozšíriteľnými jazykmi skôr, ako dospejú k ich prvému dohotoveniu. Zvyšnú časť tejto kapitoly venujeme vývoju jednoduchého kompilátora pre jeden špecifický malý programovací jazyk.

PROGRAM 5.3. *Prekladač jazyka (5.12)*

```
program Prekladač (input, output);
label 99;
const prázdny = '*';
type smerník = ↑vrchol;
```

```
        čsmerník = ↑čelo;
        vrchol = record suc, alt: smerník;
                   case terminál: boolean of
                     true: (tsym: char);
                     false: (nsym: čsmerník)
               end;
        čelo = record sym: char;
                     vstup: smerník;
                     suc: čsmerník
               end;
var zoznam, zarážka, h: čsmerník;
    p: smerník;
    sym: char;
    ok: boolean;
  procedure getsym;
  begin
    repeat read(sym); write(sym) until sym ≠ ' '
  end {getsym};
  procedure nájdi (s: char; var h: čsmerník);
    {vyhľadá neterminálny symbol s v zozname; ak v ňom nie je,
     pridá ho doňho}
    var h1: čsmerník;
  begin h1: = zoznam; zarážka↑.sym: = s;
    while h1↑.sym ≠ s do h1: = h1↑.suc;
    if h1 = zarážka then
      begin {pridaj symbol} new (zarážka);
        h1↑.suc: = zarážka; h1↑.vstup: = nil
      end;
    h: = h1
  end {nájdi};
  procedure error;
  begin writeln;
    writeln ('NESPRÁVNA SYNTAX'); goto 99
  end {error};
  procedure term (var p, q, r: smerník);
    var a, b, c: smerník;
```



```
    procedure factor (var p, q: smerník);
      var a, b: smerník; h: čsmerník;
    begin if sym in ['A' .. 'H', prázdny] then
      begin {symbol} new (a);
        if sym in ['A' .. 'H'] then
        begin {neterminál} nájdi (sym, h);
          a↑.terminál: = false; a↑.nsym: = h
        end else
        begin {terminál}
          a↑.terminál: = true; a↑.tsym: = sym
        end;
        p: = a; q: = a; getsym
      end else
      if sym = '[' then
      begin getsym; term (p, a, b); b↑.suc: = p;
        new (b); b↑.terminál: = true; b↑.tsym: = prázdny;
        a↑.alt: = b; q: = b;
        if sym ']' then getsym else error
      end else error
    end {factor};
  begin factor (p, a); q: = a;
    while sym in ['A' .. 'Z', '[', prázdny] do
    begin factor (a↑.suc, b); b↑.alt: = nil; a: = b
    end;
    r: = a
  end {term};
  procedure výraz (var p, q: smerník);
    var a, b, c: smerník;
  begin term (p, a, c); c↑.suc: = nil;
    while sym = ',' do
    begin getsym;
      term (a↑.alt, b, c); c↑.suc: = nil; a: = b
    end;
    q: = a
  end {výraz};
  procedure analýza (cieľ: čsmerník; var zhodný: boolean);
```

```
    var s: smerník:
  begin s: = cieľ↑.vstup;
    repeat
      if s↑.terminál then
      begin if s↑.tsym = sym then
        begin zhodný: = true; getsym
        end
        else zhodný: = (s↑.tsym = prázdny)
      end
      else analýza (s↑.nsym, zhodný);
      if zhodný then s: = s↑.suc else s: = s↑.alt
    until s = nil
  end {analýza};
begin {pravidlá}
  getsym; new(zarážka); zoznam: = zarážka;
  while sym ≠ '$' do
  begin nájdi (sym, h);
    getsym; if sym = '=' then getsym else error;
    výraz (h↑.vstup, p); p↑.alt: = nil;
    if sym ≠ '.' then error;
    writeln; readln; getsym
  end;
  h: = zoznam; ok: = true;
  {kontrola, či sú všetky symboly definované}
  while h ≠ zarážka do
  begin if h↑.vstup = nil then
    begin writeln ('NEDEFINOVANÝ SYMBOL', h↑.sym);
      ok: = false
    end;
    h: = h↑.suc
  end;
  if ¬ ok then goto 99;
{cieľový symbol}
  getsym; nájdi (sym, h); readln; writeln;
{vety}
  while ¬ eof(input) do
```



```
    begin write (' '); getsym; analýza (h, ok);
      if ok ∧ (sym = '.') then writeln ('SPRÁVNA')
                          else writeln ('NESPRÁVNA');
      readln
    end;
99: end.
```

5.7 PROGRAMOVACÍ JAZYK PL/0

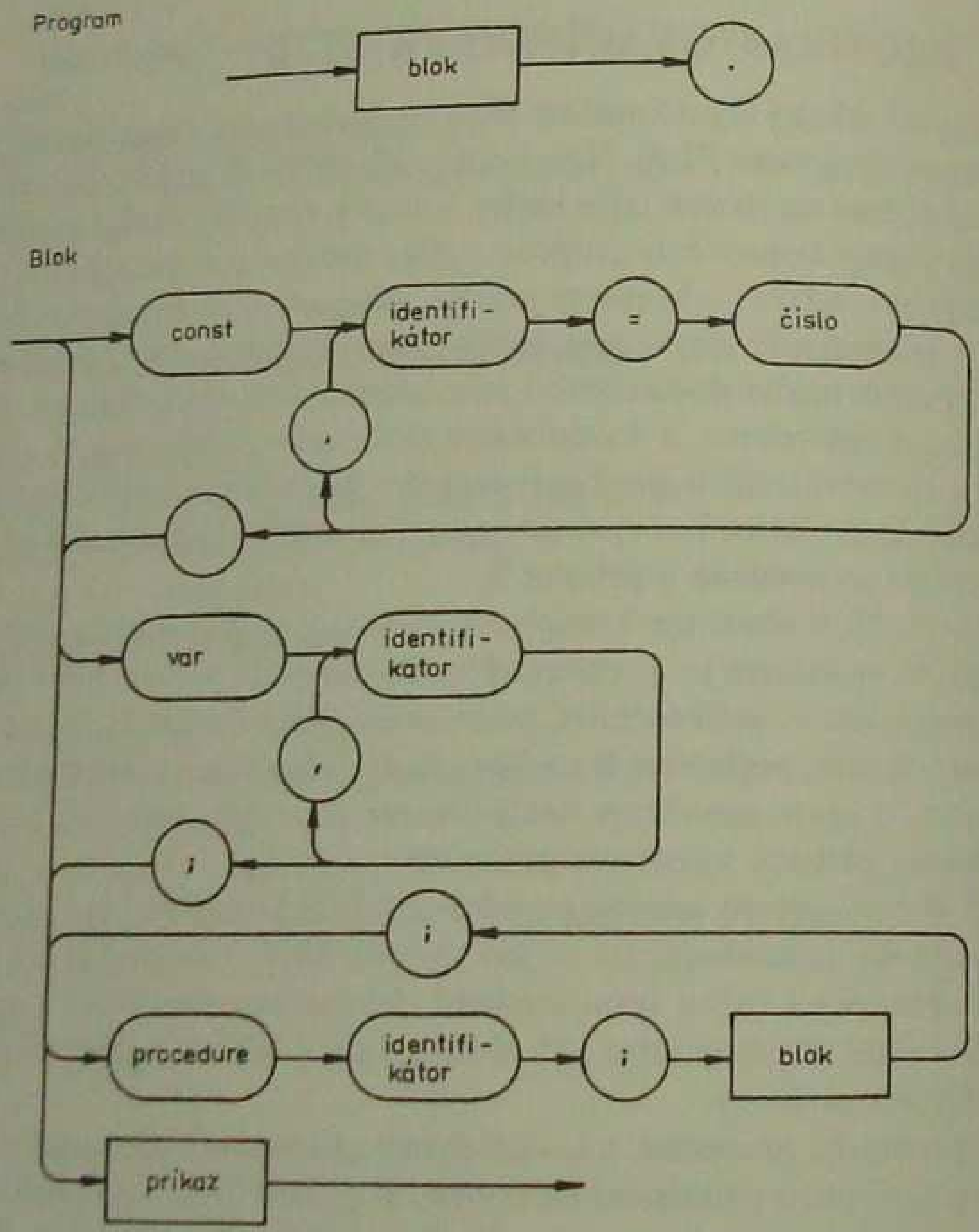
Zvyšné články tejto kapitoly sú venované vývoju kompilátora jazyka, ktorý nazývame PL/0. Nevyhnutnosť primerane malého kompilátora, vzhľadom na rozsah tejto knihy, a snaha vysvetliť všetky najzákladnejšie pojmy kompilácie jazykov vyššej úrovne predstavujú okrajové podmienky návrhu takéhoto jazyka. Nepochybne by sa dal vybrať úplne jednoduchý jazyk, ako aj veľmi zložitý jazyk; PL/0 predstavuje kompromis medzi dostatočnou jednoduchosťou, vzhľadom na zrozumiteľnosť vysvetlenia, a dostatočnou zložitosťou, vzhľadom na celkovú hodnotu vytváraného jazyka. Podstatne zložitejším jazykom je pascal, ktorého kompilátor bol vyvinutý pomocou tých istých techník a ktorého syntax je uvedená v prílohe B.

Jazyk PL/0 obsahuje kompletne všetky programové štruktúry. Základným príkazom je, pochopiteľne, priraďovací príkaz. Štruktúrovanie programu — sekvenčnosť, podmienený výpočet a cyklus — dosiahneme príkazmi **begin**/**end**, **if** a **while**. PL/0 podporuje koncepciu podprogramov, a preto umožňuje deklarovanie procedúr, resp. ich aktiváciu pomocou príkazu vyvolania procedúry.

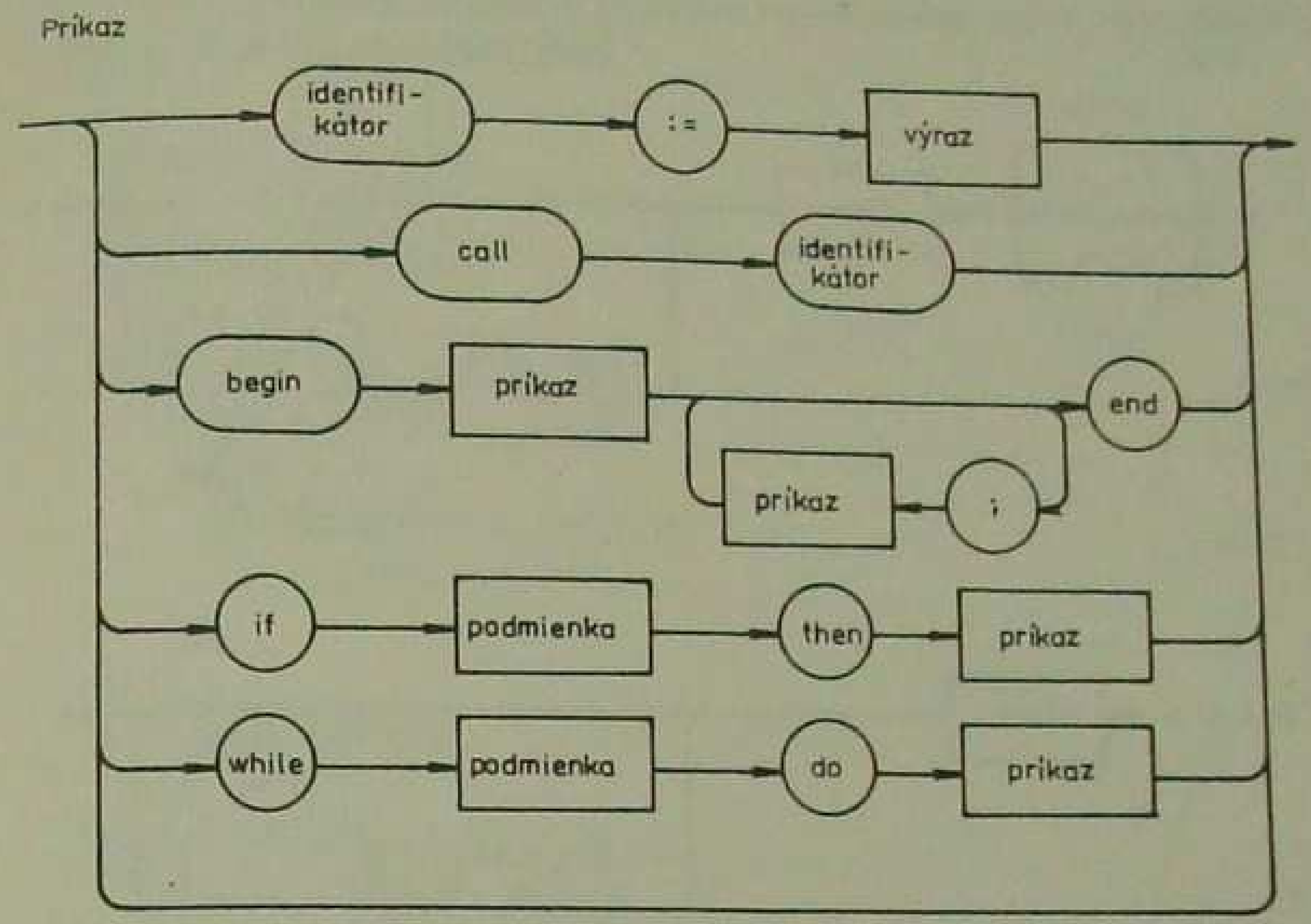
V oblasti typov údajov sa však PL/0 nekompromisne pridŕžava požiadavky jednoduchosti — jediným možným typom údajov sú celé čísla. Pomocou tohto typu môžeme deklarovať celočíselné konštanty a premenné. Pochopiteľne, PL/0 dovoľuje používať bežné aritmetické a relačné operátory.

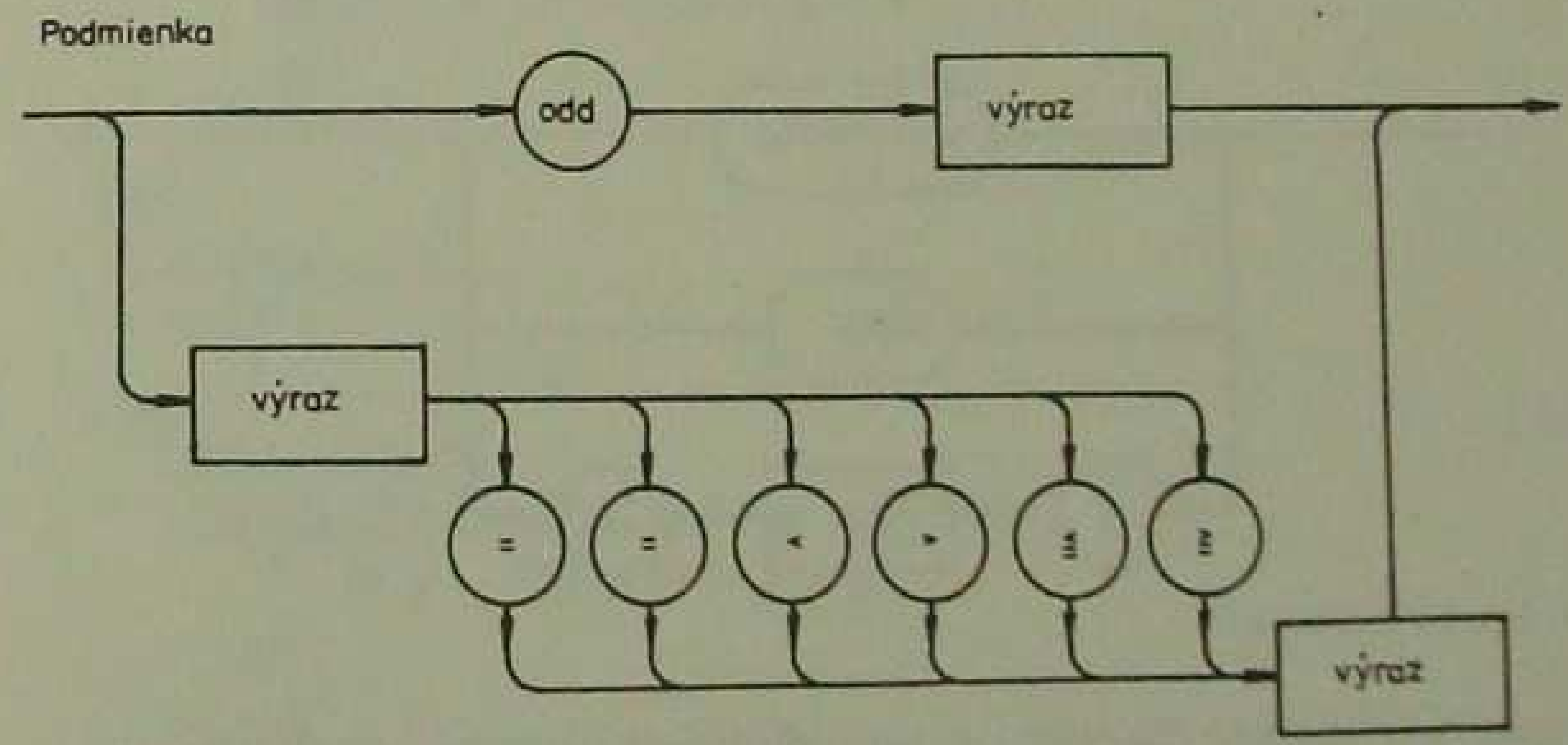
Prítomnosť procedúr, t.j. viac-menej „samostatných" úsekov programu, poskytuje príležitosť na zavedenie pojmu lokalita objektov (konštánt, premenných a procedúr). PL/0 vyžaduje, aby deklarácie všetkých objektov boli uvedené v záhlaví každej procedúry, čím sa tieto objekty stávajú lokálnymi v rámci tejto procedúry.

Tento stručný úvod a prehľad predstavujú nevyhnutné minimum znalostí potrebných na porozumenie syntaxe jazyka PL/0. Táto syntax je zobrazená na *obr. 5.4* formou siedmich diagramov. Transformáciu týchto diagramov do množiny ekvivalentných prepisovacích pravidiel BNF ponechávame na čitateľa. *Obr. 5.4* je presvedčivým príkladom



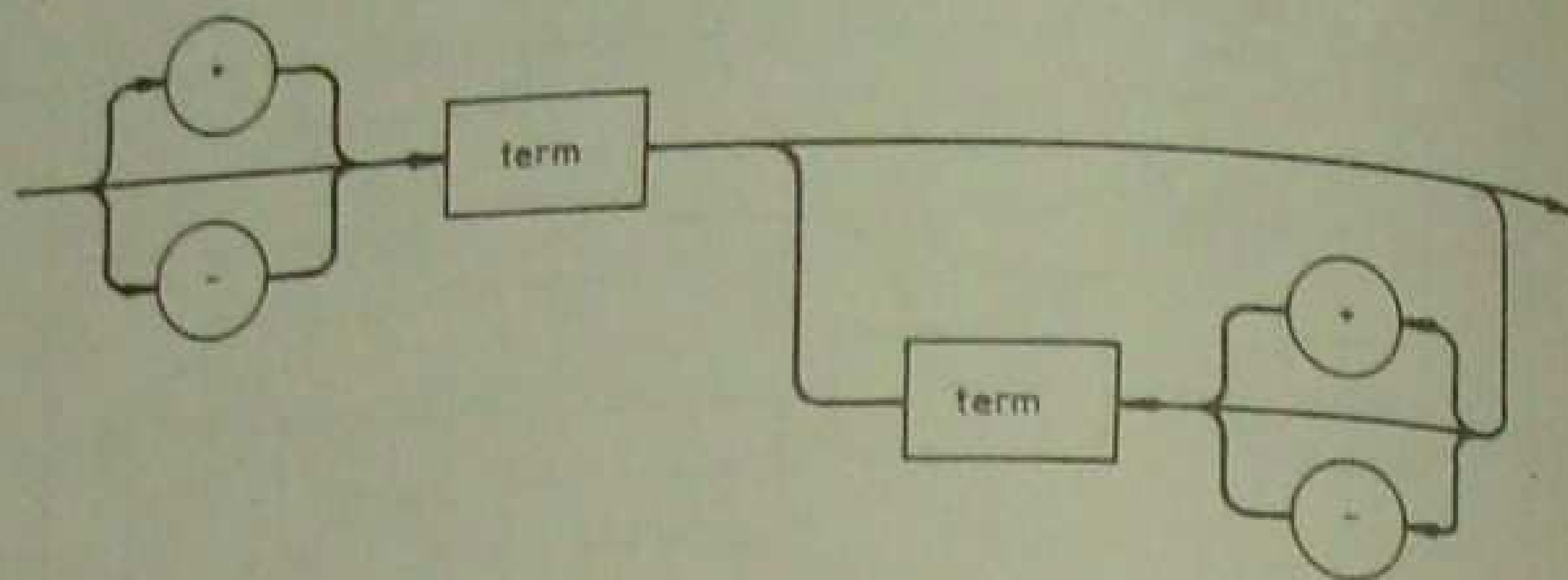
Obr. 5.4. Syntax jazyka PL/0



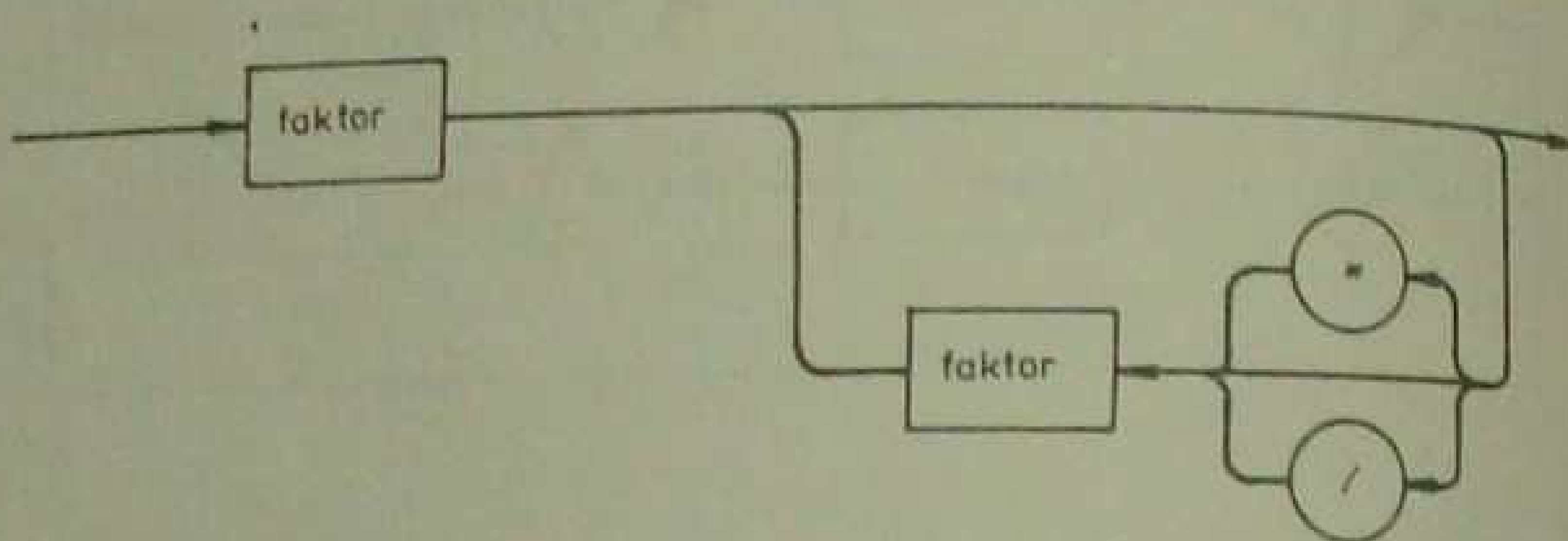


Obr. 5.4. Pokračovanie

Výraz



Term



Faktor

Obr. 5.4. Pokračovanie

výraznej sily týchto diagramov, umožňujúcich stručnú, prehľadnú zrozumiteľnú formuláciu syntaxe celého programovacieho jazyka.

Účelom nasledujúceho programu je preukázať niektoré charakteristické vlastnosti minijazyka PL/0. Program obsahuje dobre známe algoritmy na násobenie, delenie a nájdenie najväčšieho spoločného deliteľa (gcd) dvoch prirodzených čísel.


```
const m = 7; n = 85;
var x, y, z, q, r;
procedure násobenie;
   var a, b;
begin a: = x; b: = y; z: = 0;                                  (5.14)
      while b > 0 do
      begin
         if odd b then z: = z + a;
         a: = 2 * a; b: = b/2;
      end
end;
procedure delenie;
   var w;
begin r: = x; q: = 0; w: = y;
   while w ≦ r do w: = 2 * w;                                   (5.15)
   while w > y do
      begin q: = 2 * q; w: = w/2;
      if w ≦ r then
         begin r: = r − w; q: = q + 1
         end
      end
end;
procedure gcd;
   var f, g;
begin f: = x; g: = y;                                          (5.16)
   while f ≠ g do
      begin if f < g then g: = g − f;
                 if g < f then f: = f − g;
      end;
   z: = f
end;
begin
   x: = m; y: = n; call násobenie;
```

```
   x: = 25; y: = 3; call delenie;
   x: = 84; y: = 36; call gcd;
end.
```

5.8 SYNTAKTICKÝ ANALYZÁTOR JAZYKA PL/0

Vytvorenie syntaktického analyzátora bude prvý krok procesu tvorby kompilátora jazyka PL/0. Dá sa uskutočniť presne podľa zásad tvorby syntaktického analyzátora B1 až B7, ktoré boli opísané v článku 5.4. Túto metódu však možno použiť iba v tom prípade, ak sú zodpovedajúcou syntaxou splnené obmedzujúce tvrdenia 1 a 2. Musíme preto overiť, či zodpovedajúce syntaktické grafy vyhovujú uvedeným podmienkam.

Tvrdenie 1 určuje, že každá vetva vychádzajúca z určitého bodu vetvenia musí viesť k jednoznačnému začiatočnému symbolu. Toto

Začiatočné a nasledujúce symboly jazyka PL/0 *Tabuľka 5.2*

Neterminálny symbol S	Začiatočné symboly $L(S)$	Nasledujúce symboly $F(S)$
Blok	**const var** **procedure** identifikátor **if call begin while**	. ;
Príkaz	identifikátor **call** **begin if while**	. ; **end**
Podmienka	**odd** + − (identifikátor číslo	**then do**
Výraz	+ − (identifikátor číslo	. ;) R **end then do**
Term	identifikátor číslo (	. ;) R + − **end then do**
Faktor	identifikátor číslo (	. ;) R + − * / **end** **then do**

tvrdenie sa dá veľmi jednoducho overiť na príslušných syntaktických diagramoch znázornených na *obr. 5.4.* Tvrdenie 2 sa vzťahuje na všetky grafy, ktoré možno prejsť bez prečítania akéhokoľvek symbolu. Jediným takýmto grafom v syntaxi PL/0 je graf zobrazujúci príkazy jazyka. Tvrdenie 2 vyžaduje, aby všetky prvé symboly, ktoré môžu nasledovať za príkazom, boli disjunktné so začiatočným symbolom príkazov. Vzhľadom na to, že neskôr bude užitočné poznať množiny začiatočných a nasledujúcich symbolov pre všetky grafy, stanovíme si tieto množiny pre všetkých sedem neterminálnych symbolov (grafov) syntaxe jazyka PL/0 (okrem programu). *Tab. 5.2* predstavuje záruku disjunktnosti množín začiatočných a nasledujúcich symbolov príkazov. Tým je overená aplikácia zásad konštrukcie syntaktického analyzátora B1 až B7.

Pozorný čitateľ si iste všimol, že základnými symbolmi jazyka PL/0 už nie sú iba jednoduché znaky, ako to bolo v predchádzajúcich príkladoch, ale postupnosti znakov, napr. BEGIN alebo : =. Podobne ako v programe 5.3 využijeme lexikálny analyzátor, ktorý bude mať na starosti reprezentačné alebo lexikálne aspekty vstupného reťazca symbolov. V kompilátore PL/0 je lexikálny analyzátor reprezentovaný procedúrou getsym, ktorej hlavnou úlohou je vyprodukovať ďalší symbol. Lexikálny analyzátor slúži na tieto ciele:

1. Ignoruje oddeľovače (medzery).
2. Rozpoznáva kľúčové slová, akými sú napr. BEGIN, END atď.
3. Rozpoznáva ostatné slová (ako sú identifikátory). Momentálny identifikátor je priradený globálnej premennej *id*.
4. Rozpoznáva reťazce číslic ako čísla. Bežná hodnota čísla je priradená globálnej premennej *num*.
5. Rozpoznáva dvojice zvláštnych znakov, ako je napr. : =.

Počas analýzy vstupnej postupnosti znakov používa procedúra getsym lokálnu procedúru getch, ktorej úlohou je načítanie ďalšieho znaku. Okrem tejto hlavnej úlohy vykonáva procedúra getch aj tieto ďalšie činnosti:

1. Rozpoznáva a potláča informáciu o ukončení vstupného riadku.
2. Kopíruje vstup na výstupný súbor, čím vytvára protokol zdrojového programu.
3. Na začiatok každého riadku vypíše jeho poradové číslo.

Lexikálny analyzátor svojou činnosťou zabezpečuje predsnímanie

jedného symbolu dopredu. Pomocná procedúra getch navyše reprezentuje pozretie sa na jeden znak dopredu. Celkový počet pohľadov dopredu tohto kompilátora bude potom jeden symbol a jeden znak.

Podrobnosti týchto procedúr nájdeme v programe 5.4, ktorý predstavuje úplný syntaktický analyzátor jazyka PL/0. Tento analyzátor je už navyše rozšírený v tom zmysle, že všetky deklarované identifikátory konštánt, premenných a procedúr ukladá do tabuľky. Výskyt identifikátora v nejakom príkaze potom spôsobí prehľadanie tejto tabuľky za účelom zistenia, či príslušný identifikátor bol (alebo nebol) správne deklarovaný. Ak sa príslušná deklarácia nenachádza v tabuľke symbolov, došlo k syntaktickej chybe, pretože ide o formálnu chybu pri zostavovaní programu (z dôvodu použitia nedefinovaného symbolu).

Skutočnosť, že sa takáto chyba dá odhaliť iba na základe informácií z tabuľky, je dôsledkom kontextovej závislosti jazyka, vyjadrenej pravidlom, že všetky identifikátory musia byť deklarované v príslušnom kontexte. Z tohto hľadiska sú prakticky všetky programovacie jazyky kontextovo závislé; napriek tomu je bezkontextová syntax pre tieto jazyky najvýhodnejším modelom a je veľkou pomôckou pri systematickej tvorbe ich analyzátorov. Jednoduchou úpravou sa potom dokážeme vysporiadať aj s tými niekoľkými kontextovo závislými prvkami jazyka, ako to dokazuje zavedenie tabuľky identifikátorov v uvedenom syntaktickom analyzátore.

Predtým ako sa pustíme do tvorby jednotlivých procedúr syntaktického analyzátora, bude užitočné, ak uvedieme, ako príslušné syntaktické grafy navzájom závisia. Na tento účel zostrojíme diagram závislosti, ktorý zobrazuje závislosti medzi jednotlivými grafmi, t.j. pre každý graf G udáva všetky grafy $G_1, \ldots, G_n$, pomocou ktorých je graf G definovaný. Obdobne zobrazuje tie procedúry, ktoré môžu byť vyvolávané z iných procedúr. Diagram závislosti jazyka PL/0 znázorňuje *obr. 5.5.*

Cykly na *obr. 5.5* znamenajú výskyty rekurzie. Preto je dôležité, aby jazyk, v ktorom má byť implementovaný kompilátor PL/0, umožňoval použitie rekurzie. Diagram závislosti navyše predstavuje užitočnú pomôcku pri návrhu hierarchickej organizácie programu syntaktickej analýzy. Napríklad všetky procedúry analyzátora môžu byť obsiahnuté (deklarované ako lokálne) v procedúre, ktorá analyzuje konštrukciu ⟨program⟩ (a je preto hlavnou programovou časťou syntaktického

Obr. 5.5. Diagram závislosti pre jazyk PL/0

analyzátora). Podobne všetky procedúry aktivované pri analýze bloku môžu byť definované lokálne v procedúre, ktorej cieľom analýzy je ⟨blok⟩. Pochopiteľne, všetky takéto procedúry volajú lexikálny analyzátor reprezentovaný procedúrou getsym, ktorá zasa vyvoláva procedúru getch.

PROGRAM 5.4. *Syntaktický analyzátor jazyka PL/0*

```
program PL0 (input, output);
{kompilátor jazyka PL/0 — syntaktická analýza}
label 99;
const
     norw = 11;       {počet kľúčových slov}
     txmax = 100;     {veľkosť tabuľky identifikátorov}
     nmax = 14;       {maximálny počet číslic v čísle}
     al = 10;         {dĺžka identifikátorov}
```

```
type symbol =
  (nul, identifikátor, číslo, plus, mínus, krát, deleno, oddsym, eql, neq,
   lss, leq, gtr, geq, ľzátvorka, pzátvorka, čiarka, bodkočiarka, bodka,
   nadobudne, beginsym, endsym, ifsym, thensym, whilesym, dosym,
   callsym, constsym, varsym, procsym);
   alfa = packed array [1 .. al] of char;
   objekt = (konštanta, premenná, procedúra);
var ch: char;          {posledný prečítaný znak}
   sym: symbol;        {posledný prečítaný symbol}
   id: alfa;           {posledný prečítaný identifikátor}
   num: integer;       {posledné prečítané číslo}
   cc: integer;        {počet znakov}
   ll: integer;        {dĺžka riadku}
   kk: integer;
   riadok: array [1 .. 81] of char;
   a: alfa;
   slovo: array [1 .. norw] of alfa;
   wsym: array [1 .. norw] of symbol;
   ssym: array [char] of symbol;
   tab: array [0 .. txmax] of
        record meno: alfa;
               druh: objekt
        end;
  procedure error (n: integer);
  begin writeln (' ': cc, '↑', n: 2); goto 99
  end {error};
  procedure getsym;
    var i, j, k: integer;
    procedure getch;
    begin if cc = ll then
      begin if eof (input) then
          begin write ('NEÚPLNÝ PROGRAM'); goto 99
          end;
        ll: = 0; cc: = 0; write (' ');
        while ¬ eoln (input) do
          begin ll: = ll + 1; read (ch); write (ch); riadok [ll]: = ch
```



```
          end;
          writeln; ll: = ll + 1; read (riadok [ll])
      end;
      cc: = cc + 1; ch: = riadok [cc]
    end {getch};
  begin {getsym}
  while ch = ' ' do getch;
    if ch in ['A' .. 'Z'] then
    begin {identifikátor alebo kľúčové slovo}
      k: = 0;
      repeat if k < al then
        begin k: = k + 1; a[k]: = ch
        end;
        getch
      until ¬ (ch in ['A' .. 'Z', '0' .. '9']);
      if k ≧ kk then kk: = k else
        repeat a[kk]: = ' '; kk: = kk − 1
        until kk = k;
      id: = a; i: = 1; j: = norw;
      repeat k: = (i + j) div 2;
        if id ≦ slovo [k] then j: = k − 1;
        if id ≧ slovo [k] then i: = k + 1
      until i > j;
      if i − 1 > j then sym: = wsym [k] else sym: = identifikátor
    end else
    if ch in ['0' .. '9'] then
    begin {číslo} k: = 0; num: = 0; sym: = číslo;
      repeat num: = 10 * num + (ord (ch) + ord ('0'));
        k: = k + 1; getch
      until ¬ (ch in ['0' .. '9']);
      if k > nmax then error (30)
    end else
    if ch = ':' then
    begin getch;
      if ch = '=' then
      begin sym: = nadobudne; getch
```

```
      end else sym: = nul;
    end else
    begin sym: = ssym[ch]; getch
    end
  end {getsym};
  procedure blok(tx: integer);
    procedure vstup(k: objekt);
    begin {zaradenie objektu do tabuľky}
      tx: = tx + 1;
      with tab[tx] do
      begin meno: = id; druh: = k;
      end
    end {vstup};
    function pozícia (id: alfa): integer;
      var i: integer;
    begin {vyhľadanie identifikátora id v tabuľke}
      tab[0].meno: = id; i: = tx;
      while tab[i].meno ≠ id do i: = i − 1;
      pozícia: = i
    end {pozícia};
  procedure constdeklarácia;
  begin if sym = identifikátor then
    begin getsym;
      if sym = eql then
      begin getsym;
        if sym = číslo then
          begin vstup(konštanta); getsym
          end
        else error (2)
      end else error (3)
    end else error (4)
  end {constdeklarácia};
  procedure vardeklarácia;
  begin if sym = identifikátor then
    begin vstup(premenná); getsym
    end else error (4)
```



```
  end {vardeklarácia};
  procedure príkaz;
    var i: integer;
    procedure výraz;
      procedure term;
        procedure factor;
          var i: integer;
        begin
        if sym = identifikátor then
        begin i: = pozícia(id);
          if i = 0 then error (11) else
          if tab[i].druh = procedúra then error (21);
          getsym
        end else
        if sym = číslo then
        begin getsym
        end else
        if sym = ľzátvorka then
        begin getsym; výraz;
          if sym = pzátvorka then getsym else error (22)
        end
        else error (23)
        end {factor};
      begin {term} factor;
        while sym in [krát, deleno] do
          begin getsym; factor
          end
      end {term};
    begin {výraz}
      if sym in [plus, mínus] then
      begin getsym; term
      end
    end {výraz};
    procedure podmienka;
    begin
      if sym = oddsym then
```

```
    begin getsym; výraz
    end else
    begin výraz;
      if ¬ (sym in [eql, neq, lss, leq, gtr, geq]) then
        error (20) else
      begin getsym; výraz
      end
    end
  end {podmienka};
  begin {príkaz}
  if sym = identifikátor then
  begin i: = pozícia (id);
    if i = 0 then error (11) else
    if tab [i] . druh ≠ premenná then error (12);
    getsym; if sym = nadobudne then getsym else error (13);
    výraz
  end else
  if sym = callsym then
  begin getsym;
    if sym ≠ identifikátor then error (14) else
    begin i: = pozícia (id);
      if i = 0 then error (11) else
      if tab [i] . druh ≠ procedúra then error (15);
      getsym
    end
  end else
  if sym = ifsym then
  begin getsym; podmienka;
    if sym = thensym then getsym else error (16);
    príkaz;
  end else
  if sym = beginsym then
  begin getsym; príkaz;
    while sym = bodkočiarka do
      begin getsym; príkaz
      end;
```



```
      if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin getsym; podmienka;
      if sym = dosym then getsym else error (18);
      príkaz
    end
  end {príkaz};
begin {blok}
  if sym = constsym then
  begin getsym; constdeklarácia;
    while sym = čiarka do
      begin getsym; constdeklarácia
      end;
    if sym = bodkočiarka then getsym else error (5)
  end;
  if sym = varsym then
  begin getsym; vardeklarácia;
    while sym = čiarka do
      begin getsym; vardeklarácia
      end;
    if sym = bodkočiarka then getsym else error (5)
  end;
  while sym = procsym do
  begin getsym;
    if sym = identifikátor then
      begin vstup (procedúra); getsym
      end
    else error (4);
    if sym = bodkočiarka then getsym else error (5);
    blok (tx);
    if sym = bodkočiarka then getsym error (5);
  end;
  príkaz
end {blok};
begin {hlavný program}
```

```
  for ch: = 'A' to ';' do ssym[ch]: = nul;
  slovo   [1]: = 'BEGIN';          slovo   [2]: = 'CALL';
  slovo   [3]: = 'CONST';          slovo   [4]: = 'DO';
  slovo   [5]: = 'END';            slovo   [6]: = 'IF';
  slovo   [7]: = 'ODD';            slovo   [8]: = 'PROCEDURE';
  slovo   [9]: = 'THEN';           slovo  [10]: = 'VAR';
  slovo  [11]: = 'WHILE';
  wsym    [1]: = beginsym;         wsym    [2]: = callsym;
  wsym    [3]: = constsym;         wsym    [4]: = dosym;
  wsym    [5]: = endsym;           wsym    [6]: = ifsym;
  wsym    [7]: = oddsym;           wsym    [8]: = procsym;
  wsym    [9]: = thensym;          wsym   [10]: = varsym;
  wsym   [11]: = whilesym;
  ssym  ['+']: = plus;             ssym  ['-']: = minus;
  ssym  ['*']: = krát;             ssym  ['/']: = deleno;
  ssym  ['(']: = ľzátvorka;        ssym  [')']: = pzátvorka;
  ssym  ['=']: = eql;              ssym  [',']: = čiarka;
  ssym  ['.']: = bodka;            ssym  ['≠']: = neq;
  ssym  ['<']: = lss;              ssym  ['>']: = gtr;
  ssym  ['≦']: = leq;              ssym  ['≧']: = geq;
  ssym  [';']: = bodkočiarka;
  page(output);
  cc: = 0; ll: = 0; ch: = ' '; kk: = al; getsym;
  blok(0);
  if sym ≠ bodka then error (9);
99: writeln
end.
```

5.9 ZOTAVENIE SA ZO SYNTAKTICKÝCH CHÝB

Zistiť, či daný vstupný reťazec symbolov patrí do jazyka alebo nie, to bola doteraz jediná úloha syntaktického analyzátora. Vedľajšou úlohou syntaktickej analýzy bolo určiť štruktúru vety. Keď sa však vyskytla nesprávna syntaktická konštrukcia, ktorú kompilátor dokázal

odhaliť, bol cieľ syntaktickej analýzy prakticky splnený a program mohol byť ukončený. Takéto správanie kompilátora by však v praxi ťažko obstálo. Prakticky použiteľný kompilátor musí vyprodukovať primeranú diagnostiku chyby a pokračovať v analytickom procese za účelom objavenia ďalších chýb. Pokračovať v analýze je možné buď na základe určitého predpokladu o povahe chyby a úmysle autora nesprávneho programu, alebo preskočením určitej časti vstupného reťazca. Niekedy sa pokračovanie analytického procesu uskutočňuje na základe obidvoch uvedených možností. Voľba správneho predpokladu je však dosť zložitá a doteraz bezúspešne formalizovaná, pretože formalizácie syntaxe a syntaktickej analýzy neposkytujú možnosť vziať do úvahy viaceré faktory, ktoré silne ovplyvňujú ľudskú myseľ. Bežnou chybou býva napr. zanedbanie interpunkčných symbolov, akým je bodkočiarka (a to nielen pri programovaní). Ale oveľa zriedkavejšie sa stane, že niekto zabudne napísať operátor + v aritmetickom výraze. Bodkočiarka aj symbol + sú pre syntaktický analyzátor terminálnymi symbolmi bez ďalšieho rozlíšenia. Pre programátora nemá bodkočiarka prakticky žiadny význam a na konci riadku sa mu javí dokonca ako nepotrebná, pričom pre aritmetický operátor je nenahraditeľná. Existuje, pochopiteľne, oveľa viac takýchto úvah, ktoré treba uvážiť pri návrhu primeraného systému zotavovania sa zo syntaktických chýb. Všetky závisia od konkrétneho jazyka, preto ich nemožno zovšeobecniť pre všetky bezkontextové jazyky.

Predsa však existujú niektoré pravidlá a pokyny, ktoré by sa mali požadovať a dodržiavať, pretože majú platnosť aj mimo rámca jednoduchých jazykov, akým je aj náš jazyk PL/0. Ich charakteristickou črtou je, že sa týkajú jednak začiatočnej koncepcie jazyka, ako aj návrhu mechanizmu syntaktického analyzátora na zotavenie sa z chýb. Predovšetkým je jasné, že realizácia výkonného a najmä citlivého mechanizmu zotavenia sa z chýb je možná iba v prípade jednoduchej štruktúry jazyka. Odporúča sa, aby jazyk, ktorého kompilátor obsahuje mechanizmus zotavenia sa z chýb, pracujúci na základe princípu ignorovania určitej časti vstupnej postupnosti pri výskyte syntaktickej chyby, obsahoval také kľúčové slová, ktoré pravdepodobne nebudú nesprávne použité, a tým poslúžia obnoveniu chodu syntaktického analyzátora. Jazyk PL/0 sa riadi týmto pravidlom: každý štruktúrova-

ný príkaz začína jednoznačným kľúčovým slovom, akým je **begin**, **if** alebo **while**, čo platí takisto pre deklarácie, ktoré začínajú slovami **var**, **const** alebo **procedure**. Toto pravidlo budeme nazývať preto *pravidlom kľúčového slova*.

Druhé pravidlo sa už priamo týka konštrukcie syntaktického analyzátora. Syntaktická analýza metódou zhora nadol je charakteristická tým, že ciele sa rozdeľujú na podciele a analyzátory cieľov vyvolávajú analyzátory podcieľov za účelom splnenia týchto podcieľov. Druhé pravidlo špecifikuje, čo má urobiť syntaktický analyzátor v prípade zistenia syntaktickej chyby. Rozhodne by nemal iba ohlásiť zistenú chybu svojmu nadriadenému analyzátoru a prestať v procese analýzy. Požaduje sa, aby analyzátor pokračoval vo svojej analýze textu až do okamihu, keď už znova môže nasledovať prijateľná syntaktická analýza. Tejto stratégii sa zvykne hovoriť *pravidlo úniku z chybového stavu*. Jeho praktickým programátorským dôsledkom je skutočnosť, že ukončenie činnosti analyzátora môže nastať iba v jeho regulárnom mieste ukončenia. Možná presná interpretácia uvedeného pravidla pozostáva z preskočenia vstupného textu pri zistení chybnej syntaktickej konštrukcie až po prvý taký symbol, ktorý môže korektne nasledovať za rozanalyzovanou vetnou konštrukciou. To znamená, že každý analyzátor pri svojej aktivácii pozná množinu svojich nasledujúcich symbolov.

V prvom zjemňovacom (alebo obohacovacom) kroku zabezpečíme, aby každá procedúra analyzátora obsahovala explicitný parameter fsys, ktorého hodnota bude určovať množinu možných nasledujúcich symbolov. Na koniec každej procedúry zaradíme explicitný test, ktorým sa overí, či ďalší symbol vstupného textu patrí do množiny nasledujúcich symbolov (ak už splnenie tejto podmienky nevyplýva priamo z logiky programu).

Boli by sme však veľmi krátkozrakí, keby sme sa za každých okolností snažili preskakovať vstupný text až po ďalší výskyt jedného z nasledujúcich symbolov. Veď programátor môže omylom vynechať presne jeden symbol (povedzme bodkočiarku); v takomto prípade by preskočenie textu až po ďalší nasledujúci symbol mohlo znamenať katastrofu. Preto rozšírime množiny symbolov, určujúcich ukončenie preskočenia vstupného textu, o kľúčové slová jazyka, ktoré špecifikujú začiatok syntaktickej konštrukcie tak, aby sa nedal symbol prehliadnuť. Symboly, ktoré sa formou parametrov posielajú analyzujúcim procedúram, budeme nazývať stop-symboly (namiesto pojmu nasledujúce symboly). Množina stop-symbolov je inicializovaná rôznymi kľúčovými symbolmi a pri prechode hierarchiou podcieľov analýzy postupne dopĺňaná prípustnými nasledujúcimi symbolmi. Vzhľadom na flexibilitu zavedieme všeobecnú procedúru test, ktorá overí, či sa symbol nachádza (alebo nenachádza) v množine stop-symbolov. Táto procedúra (5.17) má tri parametre:

1. Množinu $s1$ prípustných nasledujúcich symbolov; ak sa momentálny symbol nenachádza v tejto množine, dôjde k výskytu chyby.

2. Množinu $s2$ prídavných stop-symbolov, ktorých prítomnosť je rozhodne chybou, ale ktoré neslobodno v žiadnom prípade ignorovať a preskočiť.

3. Číslo n určujúce príslušnú chybovú diagnostiku.

```
procedure test (s1, s2: symset; n: integer);
begin if ¬ (sym in s1) then
      begin error (n); s1 := s1 + s2;                    (5.17)
            while ¬ (sym in s1) do getsym
      end
end
```

Procedúru 5.17 možno s výhodou použiť i na vstupe do analyzujúcich procedúr, a to na zistenie, či momentálny symbol patrí do množiny začiatočných symbolov. To sa odporúča vo všetkých prípadoch, v ktorých sa analyzujúca procedúra X vyvoláva nepodmienene, ako napr. v príkaze

if $sym = a_1$ **then** S_1 **else**
⋮
if $sym = a_n$ **then** S_n **else** X

čo je výsledkom prekladu prepisovacieho pravidla

$$A ::= a_1 S_1 | \dots | a_n S_n | X \tag{5.18}$$

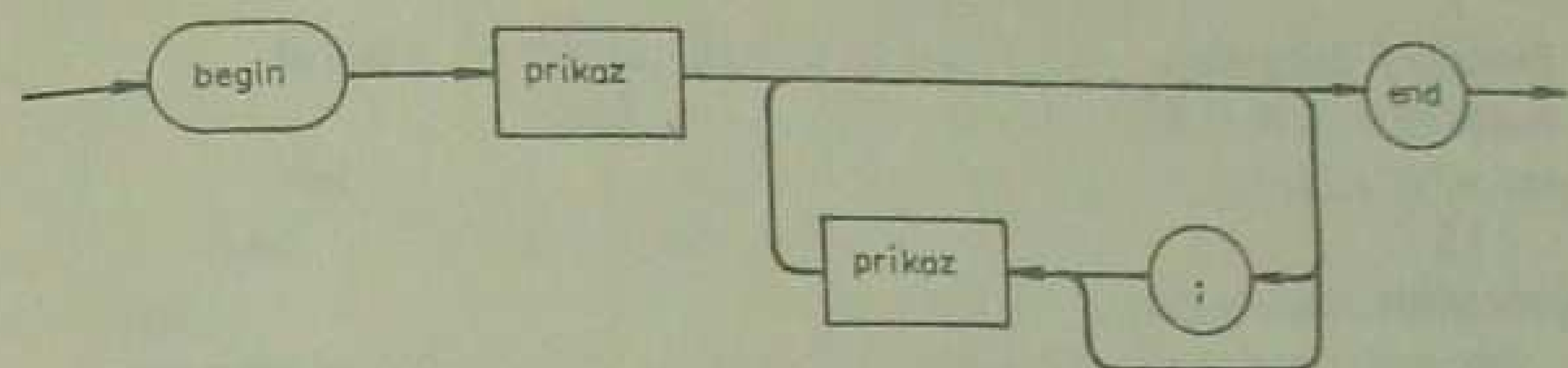
V týchto prípadoch musí byť parametrom $s1$ množina začiatočných symbolov procedúry X, pričom sa za parameter $s2$ vyberie množina nasledujúcich symbolov A (*tab. 5.2*). Podrobnosti tejto procedúry mož-

no nájsť v programe 5.5, ktorý je rozšírenou verziou programu 5.4. Pre čitateľovo pohodlie uvádzame opäť celý program syntaktického analyzátora okrem inicializácie globálnych premenných a procedúry getsym, ktoré ostali nezmenené.

Doterajšia schéma zotavovania sa zo syntaktických chýb je charakteristická tým, že proces analýzy sa obnovuje ignorovaním jedného alebo viacerých symbolov vstupného textu. To je však nevyhovujúca stratégia vo všetkých tých prípadoch, v ktorých bola chyba spôsobená vynechaním nejakého symbolu. Skúsenosť potvrdzuje, že takéto chyby sa prakticky výlučne vzťahujú na symboly, ktoré majú iba syntaktickú funkciu a nereprezentujú nijakú činnosť. Príkladom je bodkočiarka v jazyku PL/0. Skutočnosť, že sa množiny nasledujúcich symbolov rozšíria o určité kľúčové slová, spôsobuje, že syntaktický analyzátor predčasne ukončuje preskakovanie vstupných symbolov a vyzerá to tak, ako keby bol chýbajúci symbol pridaný do uvedenej množiny. Tento fakt možno vidieť v časti programu (5.19), ktorá analyzuje zložené príkazy. Chýbajúce bodkočiarky sa skutočne pridávajú pred kľúčové slová. Množina nazývaná statbegsys predstavuje množinu začiatočných symbolov konštrukcie „príkaz".

```
if sym = beginsym then
begin getsym;
    príkaz ([bodkočiarka, endsym] + fsys);
    while sym in [bodkočiarka] + statbegsys do
      begin                                                    (5.19)
        if sym = bodkočiarka then getsym else error;
        príkaz ([bodkočiarka, endsym] + fsys)
      end;
    if sym = endsym then getsym else error
end
```

Stupeň úspešnosti, s akým tento program diagnostikuje syntaktické chyby a zotavuje sa z nich, možno odhadnúť pomocou programu (5.20) napísaného v jazyku PL/0. Protokol zdrojového textu programu (5.20) predstavuje výstup programu 5.5. *Tab. 5.3* obsahuje množinu možných diagnostických správ, zodpovedajúcich číslam chýb v programe 5.5.



Obr. 5.6. Pozmenená syntax zloženého príkazu

Zoznam chybových správ kompilátora jazyka PL/0 *Tabuľka 5.3*

1. Namiesto symbolu : = treba použiť symbol =.
2. Za symbolom = musí nasledovať číslo.
3. Za identifikátorom musí nasledovať symbol =.
4. Za **const**, **var**, **procedure** musí nasledovať identifikátor.
5. Chýbajúca bodkočiarka alebo čiarka.
6. Za deklaráciou procedúry nasleduje nesprávny symbol.
7. Očakáva sa príkaz.
8. Nesprávny symbol za príkazovou časťou bloku.
9. Očakáva sa symbol bodka.
10. Medzi dvoma príkazmi chýba bodkočiarka.
11. Nedeklarovaný identifikátor.
12. Neprípustné priradenie konštante alebo procedúre.
13. Očakáva sa operátor priradenia : =.
14. Za príkazom **call** musí nasledovať identifikátor.
15. Nesprávne použitie konštanty alebo premennej.
16. Očakáva sa symbol **then**.
17. Očakáva sa symbol bodkočiarka alebo **end**.
18. Očakáva sa symbol **do**.
19. Za príkazom nasleduje nesprávny symbol.
20. Očakáva sa relačný operátor.
21. Výraz nesmie obsahovať identifikátor procedúry.
22. Chýbajúca pravá zátvorka.
23. Za faktorom nasleduje nesprávny symbol.
24. Nesprávny začiatok výrazu.
30. Číslo je väčšie ako maximálne zobraziteľné číslo v počítači.

Nasledujúci program (5.20) vznikol úmyselným „vyrobením" niektorých syntaktických chýb v programoch (5.14) až (5.16).

Neúplný program:

```
const m = 7, n = 85
var x, y, z, q, r;
   ↑5
procedure násobenie;
   var a, b
begin a: = u; b: = y; z: = 0
↑5
         ↑11
   while b > 0 do
     ↑10
   begin
   if odd b do z: = z + a;
                       ↑16
                       ↑19
   a: = 2a; b: = b/2;
         ↑23
   end
end;
procedure delenie
   var w;
   ↑5
   const dva = 2, tri: = 3;
     ↑7
                       ↑1
begin r = x; q: = 0; w: = y;
           ↑13
           ↑24                                              (5.20)
   while w ≦ r do w: = dva * w;
   while w > y
     begin q: = (2 * q; w: = w/2);
        ↑18
                ↑22
                          ↑23
     if w ≦ r then
                begin r: = r − w; q: = q + 1
                                ↑23
                end
             end
   end;
   procedure gcd;
     var f, g;
   begin f: = x; g: = y
     while f ≠ g do
         ↑17
       begin if f < g then g: = g − f;
                 if g < f then f: = f − g;
     z: = f
   end;
   begin
     x: = m; y: = n; call násobenie;
     x: = 25; y: = 3; call delenie;
     x: = 84; y: = 36; call gcd;
     call x; x: = gcd; gcd = x
       ↑15
               ↑21
                       ↑12
                           ↑13
                           ↑24
   end.
      ↑17
      ↑5
      ↑7
```

Treba si uvedomiť, že žiadna schéma, ktorá primerane efektívne prekladá správne vytvorené vety, nebude schopná takisto efektívne spracúvať všetky možné nesprávne syntaktické konštrukcie. A prečo by aj mala! Každá schéma implementovaná s primeraným úsilím zlyhá, t.j. bude neprimerane spracúvať niektoré nesprávne konštrukcie. Charakteristickou vlastnosťou dobrého kompilátora je, že:

1. Žiadna vstupná postupnosť nespôsobí jeho haváriu.

2. Odhalí a označí všetky konštrukcie, ktoré sú podľa definície jazyka nepripustné.

3. Chyby, ktoré sa objavujú pomerne často a sú čisto programátorské (zapríčinené prehliadnutím alebo nedorozumením), správne diagnostikuje a nepripustí, aby spôsobili vznik ďalších, falošných chybových správ.

Uvedená schéma pracuje uspokojivo, aj keď by sa dalo ešte čo-to vylepšiť. Cenná je rozhodne tá skutočnosť, že bola vytvorená systematickým spôsobom podľa malého počtu základných pravidiel. Tieto pravidlá boli obohatené iba o niektoré vybrané parametre, získané na základe heuristiky a zo skúseností z praktického používania programovacieho jazyka.

PROGRAM 5.5. *Syntaktický analyzátor jazyka PL/0 so systémom zotavenia sa z chýb*

```
program PL/0 (input, output);
{kompilátor jazyka PL/0, ktorého syntaktický analyzátor obsahuje
systém zotavenia sa zo syntaktických chýb}
label 99;
const norw = 11;      {počet kľúčových slov}
      txmax = 100;    {veľkosť tabuľky identifikátorov}
      nmax = 14;      {maximálny počet číslic v čísle}
      al = 10;        {dĺžka identifikátorov}
type symbol =
 (nul, identifikátor, číslo, plus, mínus, krát, deleno, oddsym, eql, neq,
  lss, leq, gtr, geq, ľzátvorka, pzátvorka, čiarka, bodkočiarka, bodka,
  nadobudne, beginsym, endsym, ifsym, thensym, whilesym, dosym,
  callsym, constsym, varsym, procsym);
  alfa = packed array [1..al] of char;
  objekt = (konštanta, premenná, procedúra);
  symset = set of symbol;
var ch: char;          {posledný prečítaný znak}
    sym: symbol;       {posledný prečítaný symbol}
    id: alfa;          {posledný prečítaný identifikátor}
    num: integer;      {posledné prečítané číslo}
    cc: integer;       {počet znakov}
    ll: integer;       {dĺžka riadku}
    kk: integer;
    riadok: array [1..81] of char;
    a: alfa;
    slovo: array [1..norw] of alfa;
    declbegsys, statbegsys, facbegsys: symset;
    wsym: array [1..norw] of symbol;
    ssym: array [char] of symbol;
    tab: array [0..txmax] of
           record meno: alfa;
                  druh: objekt
           end;
procedure error (n: integer);
begin writeln (' ': cc, '↑', n: 2);
end {error};
procedure test (s1, s2: symset; n: integer);
begin if ¬ (sym in s1) then
   begin error (n); s1 := s1 + s2;
         while ¬ (sym in s1) do getsym
   end
end {test};
procedure blok (tx: integer);
  procedure vstup (k: objekt);
  begin {zaradenie objektu do tabuľky}
    tx := tx + 1;
    with tab[tx] do
    begin meno := id; druh := k;
    end
  end {vstup};
  function pozícia (id: alfa): integer;
    var i: integer;
  begin {vyhľadanie identifikátora id v tabuľke}
    tab[0].meno := id; i := tx;
    while tab[i].meno ≠ id do i := i − 1;
    pozícia := i
```

```
end {pozicia};
procedure constdeklarácia;
begin if sym = identifikátor then
  begin getsym;
    if sym in [eql, nadobudne] then
    begin if sym = nadobudne then error (1);
      getsym;
      if sym = číslo then
      begin vstup (konštanta); getsym
      end
      else error (2)
    end else error (3)
  end else error (4)
end {constdeklarácia};
procedure vardeklarácia;
begin if sym = identifikátor then
    begin vstup (premenná); getsym
    end else error (4)
end {vardeklarácia};
procedure príkaz (fsys: symset);
var i: integer;
procedure výraz (fsys: symset);
  procedure term (fsys: symset);
  procedure factor (fsys: symset);
    var i: integer;
  begin test (facbegsys, fsys, 24);
      while sym in facbegsys do
      begin
        if sym = identifikátor then
        begin i: = pozicia (id);
          if i = 0 then error (11) else
          if tab[i].druh = procedúra then error (21);
          getsym
        end else
        if sym = číslo then
        begin getsym;
```



```
          end else
          if sym = lzátvorka then
          begin getsym; výraz ([pzátvorka] + fsys);
            if sym = pzátvorka then getsym else error (22)
          end;
          test (fsys, [lzátvorka], 23)
        end
    end {factor};
    begin {term} factor (fsys + [krát, deleno]);
      while sym in [krát, deleno] do
        begin getsym; factor (fsys + [krát, deleno])
        end
    end {term};
  begin {výraz}
    if sym in [plus, mínus] then
      begin getsym; term (fsys + [plus, mínus])
      end else term (fsys + [plus, mínus]);
    while sym in [plus, mínus] do
      begin getsym; term (fsys + [plus, mínus])
      end
  end {výraz};
  procedure podmienka (fsys: symset);
  begin
    if sym = oddsym then
    begin getsym; výraz (fsys);
    end else
    begin výraz ([eql, neq, lss, gtr, leq, geq] + fsys);
      if ¬ (sym in [eql, neq, lss, leq, gtr, geq]) then
        error (20) else
      begin getsym; výraz (fsys)
      end
    end
  end {podmienka};
  begin {príkaz}
    if sym = identifikátor then
    begin i: = pozicia (id);
```

```
      if i = 0 then error (11) else
      if tab[i].druh ≠ premenná then error (12);
      getsym; if sym = nadobudne then getsym else error (13);
      výraz (fsys);
    end else
    if sym = callsym then
    begin getsym;
      if sym ≠ identifikátor then error (14) else
        begin i: = pozícia (id);
          if i = 0 then error (11) else
          if tab[i].druh ≠ procedúra then error (15);
          getsym
        end
    end else
    if sym = ifsym then
    begin getsym; podmienka ([thensym, dosym] + fsys);
      if sym = thensym then getsym else error (16);
      príkaz (fsys)
    end else
    if sym = beginsym then
    begin getsym; príkaz ([bodkočiarka, endsym] + fsys);
      while sym in [bodkočiarka] + statbegsys do
      begin
        if sym = bodkočiarka then getsym else error (10);
        príkaz ([bodkočiarka, endsym] + fsys)
      end;
      if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin getsym; podmienka ([dosym] + fsys);
      if sym = dosym then getsym else error (18);
      príkaz (fsys);
    end;
    test (fsys, [ ], 19)
end {príkaz};
begin {blok}
```



```
repeat
  if sym = constsym then
  begin getsym;
    repeat constdeklarácia;
      while sym = čiarka do
        begin getsym; constdeklarácia
        end;
      if sym = bodkočiarka then getsym else error (5)
    until sym ≠ identifikátor
  end;
  if sym = varsym then
  begin getsym;
    repeat vardeklarácia;
      while sym = čiarka do
        begin getsym; vardeklarácia
        end;
      if sym = bodkočiarka then getsym else error (5)
    until sym ≠ identifikátor;
  end;
  while sym = procsym do
  begin getsym;
    if sym = identifikátor then
      begin vstup (procedúra); getsym
      end
    else error (4);
    if sym = bodkočiarka then getsym else error (5);
    blok (tx, [bodkočiarka] + fsys);
    if sym = bodkočiarka then
      begin getsym; test (statbegsys + [identifikátor, procsym],
                          fsys, 6)
      end
    else error (5)
  end;
  test (statbegsys + [identifikátor], declbegsys, 7)
until ¬ (sym in declbegsys);
príkaz ([bodkočiarka, endsym] + fsys);
test (fsys, [ ], 8);
```

```
end {blok};
begin {hlavný program}
    ...... Inicializácia (pozri program 5.4) ......
    cc: = 0; ll: = 0; ch: = ' '; kk: = al; getsym;
blok (0, [bodka] + declbegsys + statbegsys);
if sym ≠ bodka then error (9);
99: writeln
end.
```

5.10 PROCESOR JAZYKA PL/0

Je skutočne pozoruhodné, že doterajší vývoj kompilátora jazyka PL/0 prebiehal bez akýchkoľvek vedomostí o počítači, pre ktorý má generovať cieľový kód. Ale prečo by mala štruktúra cieľového počítača ovplyvniť schému syntaktickej analýzy a mechanizmus zotavenia sa zo syntaktických chýb! Naozaj by nemala. Naopak vhodná schéma generovania kódu pre ľubovoľný počítač by mala byť odvodená z existujúceho syntaktického analyzátora metódou postupného zjemňovania programu. Pretože práve uvedená myšlienka je naším najbližším cieľom, musíme vybrať procesor, pre ktorý budeme generovať kód.

Vzhľadom na to, že chceme, aby opis kompilátora bol primerane jednoduchý a zbavený akýchkoľvek špeciálnych úvah prameniacich z rôznych vlastností reálneho, existujúceho procesora, zvolíme si počítač podľa našich predstáv, špeciálne prispôsobený potrebám jazyka PL/0. Pretože takýto procesor v skutočnosti fyzicky neexistuje, je to hypotetický procesor a nazývame ho počítačom PL/0.

Cieľom tohto článku nie je podrobne vysvetliť dôvody, prečo sme zvolili práve takúto architektúru počítača, ale poskytnúť neformálny opis, pozostávajúci z intuitívneho úvodu a podrobnej definície procesora vo forme algoritmu. Takáto formalizácia nám môže poslúžiť ako príklad presného a podrobného algoritmického opisu skutočného procesora. Algoritmus interpretuje inštrukcie jazyka PL/0 sekvenčným spôsobom a nazýva sa *interpret*.

Počítač PL/0 sa skladá z dvoch druhov pamätí: inštrukčného registra a troch adresových registrov. Pamäť pre program nazývaná *kód*

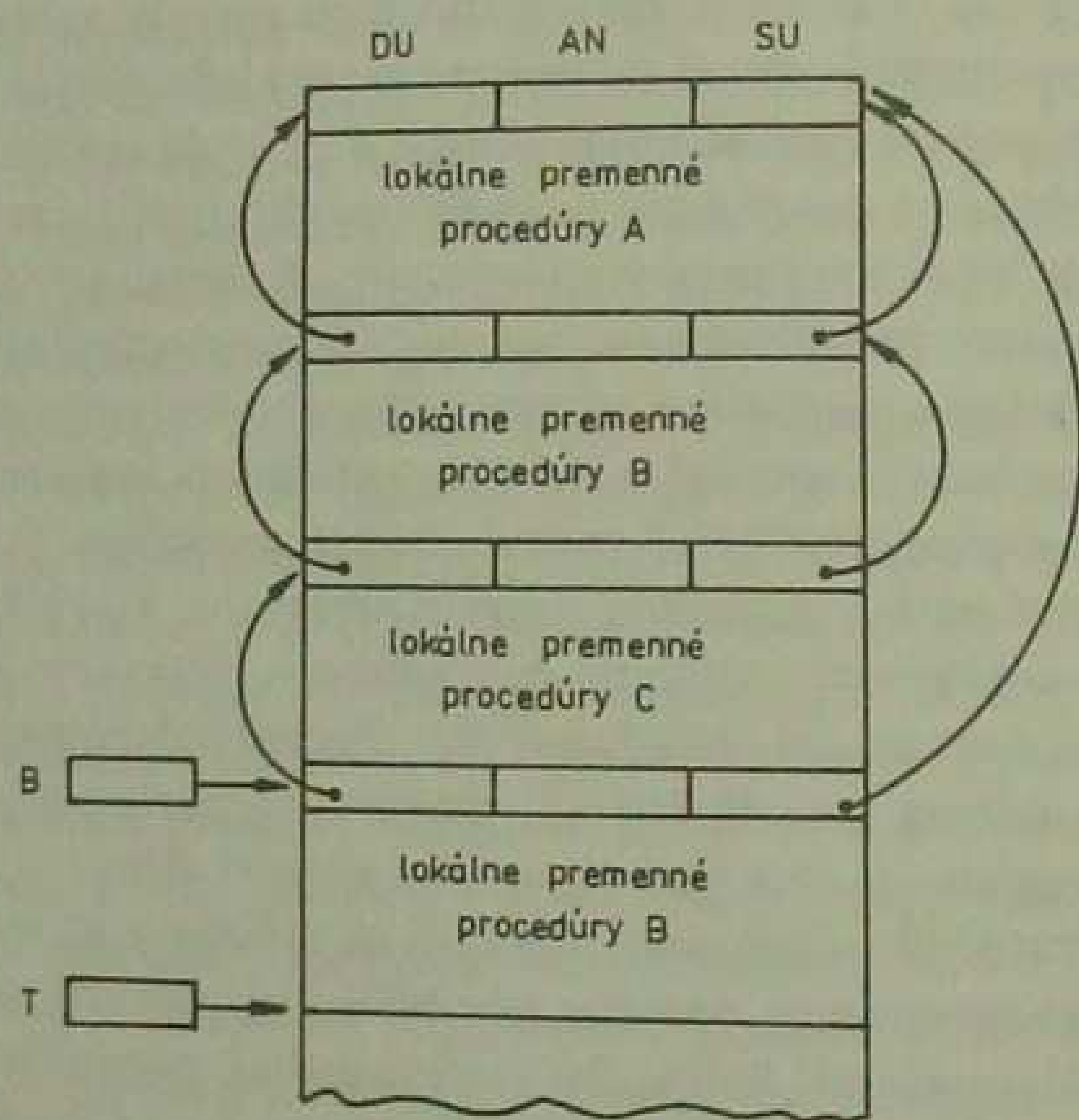
napĺňa kompilátor a v priebehu interpretovania kódu zostáva jej obsah nezmenený. Možno ju pokladať za pamäť, z ktorej sa dá iba čítať. Pamäť *S* pre údaje je organizovaná ako zásobník a všetky aritmetické operátory operujú s dvoma prvkami, ktoré sú na vrchu tohto zásobníka, pričom výsledok operácie nahradí ich operandy. Prvok na vrchu tohto zásobníka je prístupný (adresovateľný) prostredníctvom registra *T*, nazývaného *register vrchu zásobníka*. Práve interpretovaná inštrukcia sa nachádza v registri inštrukcií *I*. Register adresy programu, označený symbolom *P*, obsahuje adresu ďalšej inštrukcie, ktorá sa bude interpretovať.

Každá procedúra v jazyku PL/0 môže obsahovať lokálne premenné. Pretože procedúry môžu byť volané rekurzívne, nemôže byť pamäť pre lokálne premenné vyhradená pred skutočným volaním takejto procedúry. Preto je potrebné, aby sa údajové segmenty jednotlivých procedúr ukladali postupne do zásobníkovej pamäti *S*. Pretože sa volanie procedúr riadi výhradne stratégiou prvý-dnu-posledný-von, zásobník je vhodným prostriedkom prideľovania pamäti. Každá procedúra vlastní niektoré svoje interné informácie, menovite adresu v programe, odkiaľ bola volaná (adresu návratu) a adresu údajového segmentu procedúry, z ktorej bola volaná. Tieto dve adresy sú potrebné na úspešné pokračovanie realizácie programu po ukončení činnosti procedúry. Považujeme ich za vnútorné alebo implicitné lokálne premenné, ktorých pamäť je vyhradená v údajovom segmente procedúry, a nazývame ich adresa návratu *AN* a dynamický ukazovateľ *DU*. Začiatok dynamického ukazovateľa, t.j. adresa posledného vytvoreného údajového segmentu, sa nachádza v registri *B* nazývanom *register bázovej adresy*.

Pretože skutočné prideľovanie pamäti sa vykonáva až v čase behu (interpretácie) programu, nemôže kompilátor generovať cieľový kód s absolútnymi adresami. Jediné, čo dokáže urobiť, je vypočítať umiestnenia premenných v rámci údajového segmentu, t.j. určiť ich relatívne adresy. Interpret musí potom k týmto adresám pripočítať posunutie príslušného údajového segmentu vzhľadom na bázovú adresu. Ak je niektorá premenná lokálna v práve interpretovanej procedúre, tak sa bázová adresa nachádza v registri *B*. V opačnom prípade je potrebné túto adresu zistiť zostupným prechodom reťaze údajových segmentov. Kompilátor však môže poznať iba statickú hĺbku prístupovej cesty,

zatiaľ čo reťaz dynamických väzieb udržiava dynamickú históriu volaní procedúry. Žiaľ, tieto dve prístupové cesty nie sú tie isté.

Predpokladajme napr., že procedúra *A* volá procedúru *B*, ktorá je deklarovaná lokálne v procedúre *A*, procedúra *B* volá procedúru *C* deklarovanú lokálne v procedúre *B* a procedúra *C* volá procedúru *B* (rekurzívne). Hovoríme, že procedúra *A* je deklarovaná na úrovni 1, procedúra *B* na úrovni 2 a procedúra *C* na úrovni 3 (*obr. 5.7*). Ak chceme v procedúre *B* adresovať premennú a deklarovanú v procedúre *A*, tak kompilátor vie, že medzi procedúrami *A* a *B* existuje úrovňový rozdiel 1. Zostup o jeden krok pozdĺž reťaze dynamických väzieb však spôsobí sprístupnenie lokálnych premenných procedúry *C*!



Obr. 5.7. Zásobník počítača PL/0

Z toho vyplýva, že budeme potrebovať ďalšiu reťaz väzieb, pomocou ktorej pospájame údajové segmenty tak, aby sa kompilátor pri generovaní kódu dokázal orientovať v každej situácii. Toto spojenie nazveme statický ukazovateľ a označíme ho *SU*.

Generované adresy potom budú pozostávať z dvojíc čísel, z ktorých prvé bude určovať rozdiel statických úrovní a druhé relatívne posunutie v rámci údajového segmentu. Predpokladáme, že každé pamäťové miesto môže obsahovať buď adresu, alebo celé číslo.

Množina inštrukcií počítača PL/0 je prispôsobená požiadavkám jazyka PL/0. Obsahuje tieto inštrukcie:

1. Inštrukcia na uloženie čísel (literálov) do zásobníka (LIT).
2. Inštrukcia na uloženie premenných na vrch zásobníka (LOD).
3. Inštrukcia na zápis objektu z vrchu zásobníka do pamäti, zodpovedajúca priraďovacím príkazom (STO).
4. Inštrukcia na vyvolanie procedúry, zodpovedajúca príkazu **call** (CAL).
5. Inštrukcia na vyhradenie pamäti v zásobníku zväčšením hodnoty smerníka, ukazujúceho na vrch zásobníka *T* (INT).
6. Inštrukcie na nepodmienené a podmienené odovzdanie riadenia (skokové inštrukcie), používané v príkazoch **if** a **while** (JMP, JPC).
7. Množina aritmetických a relačných operátorov (OPR).

Formát inštrukcií (pozri *obr. 5.8*) pozostáva z troch prvkov: kódu operácie *f* a parametrov (jedného alebo dvoch). V prípade operátorov určuje parameter a identitu operátora, v ostatných prípadoch buď číslo (ide o inštrukcie LIT a INT), alebo adresu v programe (JMP, JPC, CAL), alebo adresu údajov (LOD, STO).

Obr. 5.8. Formát inštrukcie

Podrobnosti o činnosti počítača PL/0 možno vidieť v procedúre, nazvanej interpret, ktorá je časťou programu 5.6. Tento program je kombináciou úplného kompilátora s interpretom a slúži na preklad a výpočet programov napísaných v jazyku PL/0. Čitateľ sa môže v rámci cvičenia pokúsiť o modifikáciu tohto programu s cieľom generovať cieľový kód pre existujúci počítač. Výsledné rozšírenie kompilátora môžeme považovať za mieru vhodnosti vybraného počítača pre uvedenú úlohu.

Nepochybujeme o tom, že by sa uvedený počítač PL/0 dal vylepšiť

dômyselnejšou organizáciou s efektívnejšími operáciami. Jedným z možných kandidátov na vylepšenie je mechanizmus adresovania. Uvedené riešenie sme zvolili pre jeho prirodzenú jednoduchosť a preto, že z neho sú odvodené všetky možné vylepšenia.

5.11 GENEROVANIE CIEĽOVÉHO KÓDU

Keď chce kompilátor zostaviť (vygenerovať) inštrukciu, musí poznať jej operačný kód a parameter, ktorým môže byť buď literál (číslo), alebo adresa. Počas spracúvania deklarácií konštánt, premenných a procedúr spája kompilátor tieto hodnoty s príslušnými identifikátormi. Na tieto účely je tabuľka identifikátorov rozšírená o atribúty, ktoré sú spojené s každým identifikátorom. Ak príslušný identifikátor označuje konštantu, jeho atribútom je hodnota tejto konštanty; ak identifikátor označuje premennú, jeho atribútom je jej adresa, pozostávajúca z posunutia a úrovne; ak identifikátor označuje procedúru, tak je jeho atribútom adresa vstupného bodu procedúry a jej úroveň. Zodpovedajúce rozšírenie deklarácie premennej *tab* je uvedené v programe 5.6. Je to pozoruhodný príklad postupného zjemňovania (alebo obohacovania) deklarácie údajov, postupujúceho súčasne so zjemňovaním príkazovej časti programu.

Zatiaľ čo hodnoty konštánt poskytuje samotný text programu, adresy premenných musí vypočítať kompilátor. Pretože jazyk PL/0 je dostatočne jednoduchý, zvolila sa stratégia sekvenčného prideľovania pamäti pre premenné a kód. Spracovaním deklarácie každej premennej sa zvýši hodnota indexu prideľovania pamäti pre údaje o jednotku (pretože každá premenná zaberá podľa definície počítača PL/0 presne jedno pamäťové miesto). Uvedený index, označíme ho *dx*, je inicializovaný v okamihu zahájenia kompilácie procedúry, čím je súčasne dokumentovaná skutočnosť, že údajový segment procedúry je na začiatku jej kompilácie prázdny. (V skutočnosti je indexu *dx* priradená hodnota 3, pretože každý údajový segment obsahuje aspoň tri interné premenné *AN*, *DU*, *SU* (pozri predchádzajúci článok).) Príslušné výpočty atribútov identifikátorov sú zahrnuté v procedúre vstup, ktorá pridáva každý nový identifikátor do tabuľky.

Generovanie skutočného kódu pomocou uvedených informácií

o operandoch je už jednoduché. Vzhľadom na primeranú zásobníkovú organizáciu počítača PL/0 je vzťah medzi operandmi a operátormi zdrojového jazyka a inštrukciami cieľového kódu v pomere jedna ku jednej. Úlohou kompilátora je v podstate iba vhodné preskupenie do postfixovej formy. Pod postfixovou formou rozumieme také usporiadanie operandov a operátorov, v rámci ktorého operátory vždy nasledujú za svojimi operandmi. Zatiaľ čo pri bežnej, *infixovej forme* sa operátory nachádzajú medzi operandmi. *Postfixová forma* sa niekedy nazýva aj *Poľská forma* (podľa svojho autora LUKASCIEWICZA) alebo bezzátvorková forma, pretože zátvorky sú zbytočné. Niektoré vzťahy medzi infixovou a postfixovou formou výrazov ukazuje *tab. 5.4* (pozri aj odsek 4.4.2).

Tabuľka 5.4

Výrazy v infixovej a postfixovej forme

Infixová forma	Postfixová forma
$x + y$	$xy +$
$(x - y) + z$	$xy - z +$
$x - (y + z)$	$xyz +$
$x * (y + z) * w$	$xyz + * w *$

Veľmi jednoduchá technika tejto transformácie je realizovaná v procedúrach výraz a term v programe 5.6. Je to v podstate iba záležitosť pozdržania momentu generovania aritmetického operátora. Čitateľ by si mal overiť, či príslušné procedúry syntaktického analyzátora dbajú aj o patričnú interpretáciu všeobecných pravidiel priority medzi rôznymi operátormi.

Preklad podmienených príkazov a príkazov cyklu je o niečo zložitejšia záležitosť. V týchto prípadoch je potrebné vygenerovať inštrukcie skokov, ktorých cieľová adresa niekedy nie je známa. Ak trváme na dôslednom sekvenčnom generovaní inštrukcií v tvare výstupného súboru, tak potrebujeme schému dvojprechodového kompilátora. Cieľom činnosti druhého prechodu je doplniť neúplné inštrukcie skokov o ich cieľové adresy. Alternatívne riešenie, aplikované v našom kompilátore, spočíva v umiestnení inštrukcií do nejakého poľa, t. j. do priamo prístupnej pamäti. V tomto poli sa potom príslušné inštrukcie dopĺňajú

chýbajúcimi adresami (len čo sú známe). Táto operácia sa nazýva *fixovanie inštrukcie* (z anglického termínu fixup — pozn. prekl.).

Jedinou prídavnou operáciou pri generovaní takéhoto skoku na neznámu adresu (skok dopredu) je uchovanie jej adresy, t. j. jej indexu do pamäti programu. Táto adresa sa potom pri fixovaní použije na sprístupnenie príslušnej neúplnej inštrukcie. Podrobnosti možno nájsť opäť v programe 5.6 (pozri procedúry spracúvajúce príkazy **if** a **while**). Schémy generovania kódu pre príkazy **if** a **while** sú ($L1$ a $L2$ sú adresy inštrukcií):

if C **then** S	**while** C **do** S
kód pre podmienku C	$L1$: kód pre C
JPC $L1$	JPC $L2$
	kód pre S
kód pre príkaz S	JMP $L1$ pre S
$L1$: ...	$L2$: ...

Pre jednoduchosť programu zavedieme pomocnú premennú *gen*. Jej úlohou je vyprodukovať hotovú inštrukciu podľa jej troch parametrov. Automaticky zvyšuje hodnotu indexu *cx* (nazvaného počítadlo adries), ktorý určuje adresu najbližšie vygenerovanej inštrukcie. Nasledujúci príklad dokumentuje v mnemonickom tvare vygenerovaný kód pre procedúru násobenia (procedúra 5.14). Poznámky na pravej strane inštrukcií sú uvedené iba kvôli zrozumiteľnosti a dokumentácii.

Vygenerovaný kód pre procedúru 5.14 napísanú v jazyku PL/0

```
 2  INT  0,5   vyhradenie miesta pre ukazovatele a lokálne pre-
               menné
 3  LOD  1,3   x
 4  STO  0,3   a
 5  LOD  1,4   y
 6  STO  0,4   b
 7  LIT  0,0   0
 8  STO  1,5   z
 9  LOD  0,4   b
10  LIT  0,0   0
11  OPR  0,12  >
12  JPC  0,29
13  LOD  0,4   b
14  OPR  0,7   odd
15  JPC  0,20
16  LOD  1,5   z
17  LOD  0,3   a
18  OPR  0,2   +
19  STO  1,5   z
20  LIT  0,2   2
21  LOD  0,3   a
22  OPR  0,4   *
23  STO  0,3   a
24  LOD  0,4   b
25  LIT  0,2   2
26  OPR  0,5   /
27  STO  0,4   b
28  JMP  0,9
29  OPR  0,0   návrat
```


Mnohé úlohy súvisiace s kompiláciou programovacích jazykov sú oveľa zložitejšie ako tie, ktoré sme uviedli pri kompilátore jazyka PL/0 pre počítač PL/0 [5-4]. Väčšina z nich ťažko dospeje k elegantnej organizácii. Čitateľ, ktorý by sa pokúsil rozšíriť uvedený kompilátor jazyka buď za účelom zvýšenia výrazových možností jazyka, alebo pre konvenčnejší počítač, čoskoro spozná pravdivosť tohto tvrdenia. Záverom môžeme konštatovať, že základný prístup k tvorbe zložitého programu, uvedeného v tejto knihe, zostáva v platnosti, ba dokonca ešte i získava na svojej hodnote, ak bude riešená úloha zložitejšia a intelektuálne náročnejšia. Dôkazom úspešnosti použitia spomínaného prístupu môžu byť konštrukcie veľkých kompilátorov [5-1] a [5-9].

PROGRAM 5.6. *Kompilátor jazyka PL/0*

```
program PL/0 (input, output);
{kompilátor jazyka PL/0 s generátorom cieľového kódu}
label 99;
const norw = 11;        {počet kľúčových slov}
```

```
    txmax = 100;     {veľkosť tabuľky identifikátorov}
    nmax = 14;       {maximálny počet číslic v čísle}
    al = 10;         {dĺžka identifikátorov}
    amax = 2047;     {najvyššia adresa}
    levmax = 200;    {maximálna hĺbka vložených blokov}
    cxmax = 200;     {veľkosť priestoru pre kód}
type symbol =
  (nul, identifikátor, číslo, plus, mínus, krát, deleno, oddsym, eql, neq,
   lss, leq, gtr, geq, ľzátvorka, pzátvorka, čiarka, bodkočiarka, bodka,
   nadobudne, beginsym, endsym, ifsym, thensym, whilesym, dosym,
   callsym, constsym, varsym, procsym);
  alfa = packed array [1 .. al] of char;
  objekt = (konštanta, premenná, procedúra);
  symset = set of symbol;
  fct = (lit, opr, lod, sto, cal, int, jmp, jpc); {inštrukcie}
  inštrukcia = packed record
              f: fct;           {kód inštrukcie}
              l: 0 .. levmax;   {úroveň}
              a: 0 .. amax;     {adresa posunutia}
            end;
{LIT  0, a: ulož konštantu a do zásobníka
 OPR  0, a: vykonaj inštrukciu a
 LOD  l, a: ulož premenné l, a na vrchol zásobníka
 STO  l, a: zapíš premennú l z vrchu zásobníka do pamäti
 CAL  l, a: volaj procedúru a z úrovne l
 INT  0, a: zvýš obsah registra t o hodnotu a
 JMP  0, a: vykonaj skok na adresu a
 JPC  0, a: vykonaj podmienený skok na adresu a}
var ch: char;          {posledný prečítaný znak}
    sym: symbol;       {posledný prečítaný symbol}
    id: alfa;          {posledný prečítaný identifikátor}
    num: integer;      {posledné prečítané číslo}
    cc: integer;       {počet znakov}
    ll: integer;       {dĺžka riadku}
    kk, err: integer;
    cx: integer;       {počitadlo adries}
```



```
    riadok: array [1 .. 81] of char;
    a: alfa;
    kód: array [0 .. cxmax] of inštrukcia;
    slovo: array [1 .. norw] of symbol;
    ssym: array [char] of symbol;
    wsym: array [1 .. norw] of symbol;
    mnemo: array [fct] of
                    packed array [1 .. 5] of char;
    declbegsys, statbegsys, facbegsys: symset;
            tab: array [0 .. txmax] of
                  record meno: alfa;
                    case druh: objekt of
                      konštanta: (val: integer);
                      premenná, procedúra: (úroveň, adr: integer)
                  end;
procedure error (n: integer);
begin
  writeln ('****', ' ': cc − 1, '↑', n: 2); err: = err + 1
end {error};
procedure getsym;
  var i, j, k: integer;
  procedure getch;
  begin if cc = ll then
    begin if eof (input) then
            begin write ('NEÚPLNÝ PROGRAM'); goto 99
            end;
      ll: = 0; cc: = 0; write (cx: 5, ' ');
      while ¬ eoln (input) do
        begin ll: = ll + 1; read (ch); write (ch); riadok [ll]: = ch
        end;
        writeln; ll: = ll + 1; read (riadok [ll])
    end;
    cc: = cc + 1; ch: = riadok [cc]
  end {getch};
begin {getsym}
  while ch = ' ' do getch;
```

```
if ch in ['A'..'Z'] then
begin {identifikátor alebo kľúčové slovo}
k := 0;
  repeat if k < al then
    begin k := k + 1; a[k] := ch
    end;
    getch
  until ¬(ch in ['A'..'Z', '0'..'9']);
  if k ≧ kk then kk := k else
    repeat a[kk] := ' '; kk := kk − 1
    until kk = k;
  id := a; i := 1; j := norw;
  repeat k := (i + j) div 2;
    if id ≦ slovo[k] then j := k − 1;
    if id ≧ slovo[k] then i := k + 1
  until i > j;
  if i − 1 > j then sym := wsym[k] else sym := identifikátor
end else
if ch in ['0'..'9'] then
begin {číslo} k := 0; num := 0; sym := číslo;
  repeat num := 10 * num + (ord(ch) − ord('0'));
    k := k + 1; getch
  until ¬(ch in ['0'..'9']);
  if k > nmax then error (30)
end else
if ch = ':' then
begin getch;
  if ch = '=' then
  begin sym := nadobudne; getch
  end else sym := nul;
end else
begin sym := ssym[ch]; getch
end
end {getsym};
procedure gen (x: fct; y, z: integer);
begin if cx > cxmax then
```



```
    begin write ('PROGRAM JE PRÍLIŠ VEĽKÝ'); goto 99
    end;
    with kód [cx] do
      begin f := x; l := y; a := z
      end;
    cx := cx + 1
end {gen};
procedure test (s1, s2: symset; n: integer);
begin if ¬(sym in s1) then
    begin error (n); s1 := s1 + s2;
        while ¬(sym in s1) do getsym
    end
end {test};
procedure blok (lev, tx: integer; fsys: symset);
  var dx: integer;     {index prideľovania pamäti pre údaje}
      tx0: integer;    {začiatočný index do tabuľky identifikátorov}
      cx0: integer;    {začiatočná hodnota počitadla adries}
  procedure vstup (k: objekt);
  begin {zaradenie objektu do tabuľky}
    tx := tx + 1;
    with tab[tx] do
    begin meno := id; druh := k;
      case k of
        konštanta: begin if num > amax then
                           begin error (31); num := 0 end;
                         val := num
                   end;
        premenná: begin úroveň := lev; adr := dx; dx := dx + 1;
                  end;
        procedúra: úroveň := lev
        end
      end
    end {vstup};
    function pozícia (id: alfa): integer;
      var i: integer;
    begin {vyhľadanie identifikátora id v tabuľke}
```

```
        tab[0].meno: = id; i: = tx;
        while tab[i].meno ≠ id do i: = i − 1;
        pozicia: = i
      end {pozicia};
      procedure constdeklarácia;
      begin if sym = identifikátor then
        begin getsym;
        if sym in [eql, nadobudne] then
        begin if sym = nadobudne then error (1);
          getsym;
            if sym = číslo then
            begin vstup (konštanta); getsym
            end
          else error (2)
        end else error (3)
      end else error (4)
    end {constdeklarácia};
  procedure vardeklarácia;
  begin if sym = identifikátor then
      begin vstup (premenná); getsym
      end else error (4)
  end {vardeklarácia};
  procedure zobrazkód;
    var i: integer;
  begin {vytlačenie vygenerovaného kódu pre tento blok}
    for i: = cx0 to cx − 1 do
      with kód[i] do
        writeln (i, mnemo [f]: 5, l:3, a:5)
  end {zobrazkód};
  procedure prikaz (fsys: symset);
    var i, cx1, cx2: integer;
    procedure výraz (fsys: symset);
      var addop: symbol;
      procedure term (fsys: symset);
        var mulop: symbol;
        procedure factor (fsys: symset);
          var i: integer;
        begin test (facbegsys, fsys, 24);
          while sym in facbegsys do
          begin
            if sym = identifikátor then
            begin i: = pozícia (id);
              if i = 0 then error (11) else
              with tab[i] do
              case druh of
              konštanta: gen (lit, 0, val);
              premenná: gen (lod, lev-úroveň, adr);
              procedúra: error (21)
              end;
              getsym
            end else
            if sym = číslo then
            begin if num > amax then
                begin error (30); num: = 0
                end;
              gen (lit, 0, num); getsym
            end else
            if sym = ľzátvorka then
            begin getsym; výraz ([pzátvorka] + fsys);
              if sym = pzátvorka then getsym else error (22)
            end;
            test (fsys, [ľzátvorka], 23)
          end
        end {factor};
      begin {term} factor (fsys + [krát, deleno]);
        while sym in [krát, deleno] do
          begin mulop: = sym, getsym; factor (fsys + [krát, deleno]);
          if mulop = krát then gen (opr, 0, 4) else gen (opr, 0, 5)
          end
      end {term};
    begin {výraz}
      if sym in [plus, minus] then
```

```
    begin addop: = sym; getsym; term (fsys + [plus, minus]);
      if addop = minus then gen (opr, 0, 1)
    end else term (fsys + [plus, minus]);
  while sym in [plus, minus] do
    begin addop: = sym; getsym; term (fsys + [plus, minus]);
      if addop = plus then gen (opr, 0, 2) else gen (opr, 0, 3)
    end
end {výraz};
procedure podmienka (fsys: symset);
  var relop: symbol;
begin
  if sym = oddsym then
    begin getsym; výraz (fsys); gen (opr, 0, 6)
    end else
    begin výraz ([eql, neq, lss, gtr, leq, geq] + fsys);
      if ¬ (sym in [eql, neq, lss, leq, gtr, geq]) then
        error (20) else
      begin relop: = sym; getsym; výraz (fsys);
        case relop of
        eql: gen (opr, 0, 8);
        neq: gen (opr, 0, 9);
        lss: gen (opr, 0, 10);
        geq: gen (opr, 0, 11);
        gtr: gen (opr, 0, 12);
        leq: gen (opr, 0, 13);
        end
      end
  end
end {podmienka};
begin {príkaz}
  if sym = identifikátor then
  begin i: = pozícia (id);
    if i = 0 then error (11) else
    if tab[i].druh ≠ premenná then
      begin {priradenie hodnoty objektu, ktorý nie je premennou}
            error (12); i: = 0
```



```
      end;
    getsym; if sym = nadobudne then getsym else error (13);
    výraz (fsys);
    if i ≠ 0 then
      with tab[i] do gen (sto, lev-úroveň, adr)
  end else
  if sym = callsym then
  begin getsym;
    if sym ≠ identifikátor then error (14) else
      begin i: = pozícia (id);
        if i: = 0 then error (11) else
        with tab[i] do
          if druh = procedúra then gen (cal, lev-úroveň, adr)
          else error (15);
        getsym
      end
  end else
  if sym = ifsym then
  begin getsym; podmienka ([thensym, dosym] + fsys);
    if sym = thensym then getsym else error (16);
    cx1: = cx; gen (jpc, 0, 0);
    príkaz (fsys); kód [cx1].a: = cx
  end else
  if sym = beginsym then
  begin getsym; príkaz ([bodkočiarka, endsym] + fsys);
    while sym in [bodkočiarka] + statbegsys do
    begin
      if sym = bodkočiarka then getsym else error (10);
      príkaz ([bodkočiarka, endsym] + fsys)
    end;
    if sym = endsym then getsym else error (17)
  end else
  if sym = whilesym then
  begin cx1: = cx; getsym; podmienka ([dosym] + fsys);
    cx2: = cx; gen (jpc, 0, 0);
    if sym = dosym then getsym else error (18);
```

```
    príkaz (fsys); gen (jmp, 0, cx1); kód [cx2].a: = cx
  end;
  test (fsys, [ ], 19)
  end {príkaz};
begin {blok}
  dx: = 3; tx0: = tx; tab[tx].adr: = cx; gen (jmp, 0, 0);
  if lev > levmax then error (32);
  repeat
    if sym = constsym then
    begin getsym;
      repeat constdeklarácia;
        while sym = čiarka do
          begin getsym; constdeklarácia
          end;
      if sym = bodkočiarka then getsym else error (5)
      until sym ≠ identifikátor
    end;
    if sym = varsym then
    begin getsym;
      repeat vardeklarácia;
        while sym = čiarka do
          begin getsym; vardeklarácia
          end;
        if sym = bodkočiarka then getsym else error (5)
      until sym ≠ identifikátor;
    end;
    while sym = procsym do
    begin getsym;
      if sym = identifikátor then
        begin vstup (procedúra); getsym
        end
      else error (4);
      if sym = bodkočiarka then getsym else error (5);
      blok (lev + 1, tx, [bodkočiarka] + fsys);
      if sym = bodkočiarka then
```



```
        begin getsym; test (statbegsys + [identifikátor,
                                 procsym], fsys, 6)
        end
      else error (5)
    end;
    test (statbegsys + [identifikátor], declbegsys, 7)
  until ¬ (sym in declbegsys);
  kód [tab[tx0]. adr].a: = cx;
  with tab[tx0] do
    begin adr: = cx; {začiatočná adresa kódu}
    end;
  cx0: = cx; gen (int, 0, dx);
  príkaz ([bodkočiarka, endsym] + fsys);
  gen (opr, 0, 0); {návrat}
  test (fsys, [ ], 8);
  zobrazkód;
end {blok};
procedure interpret;
  const maxzásobník = 500;
  var p, b, t: integer; {programový, bázový a zásobníkový register}
      i: inštrukcia; {inštrukčný register}
      s: array [1..maxzásobník] of integer; {pamäť pre údaje}
  function báza (l: integer): integer;
    var b1: integer;
  begin b1: = b; {vyhľadanie bázy na l-tej úrovni}
    while l > 0 do
      begin b1: = s[b1]; l: = l − 1
      end;
    báza: = b1
  end {báza};
begin writeln ('START PL/0');
  t: = 0; b: = 1; p: = 0;
  s[1]: = 0; s[2]: = 0; s[3]: = 0;
  repeat i: = kód [p]; p: = p + 1;
    with i do
```

```
        case f of
        lit:  begin t: = t + 1; s[t]: = a
              end;
        opr:  case a of {operátor}
                0: begin {návrat}
                     t: = b − 1; p: = s[t + 3]; b: = s[t + 2];
                   end;
                1: s[t]: = −s[t];
                2: begin t: = t − 1; s[t]: = s[t] + s[t + 1]
                   end;
                3: begin t: = t − 1; s[t]: = s[t] − s[t + 1]
                4: begin t: = t − 1; s[t]: = s[t] * s[t + 1]
                   end;
                5: begin t: = t − 1; s[t]: ≐ s[t] div s[t + 1]
                   end;
                6: s[t]: = ord (odd s[t]));
                8: begin t: = t − 1; s[t]: = ord (s[t] = s[t + 1])
                   end;
                9: begin t: = t − 1; s[t]: = ord (s[t] ≠ s[t + 1])
                   end;
               10: begin t: = t − 1; s[t]: = ord (s[t] s[t + 1])
                   end;
               11: begin t: = t − 1; s[t]: = ord (s[t] = s[t + 1])
                   end;
               12: begin t: = t − 1; s[t]: = ord (s[t] s[t + 1])
                   end;
               13: begin t: = t − 1; s[t]: = ord (s[t] = s[t + 1])
                   end;
              end;
        lod:  begin t: = t + 1; s[t]: = s[báza (l) + a]
              end;
        sto:  begin s[báza (l) + a]: = s[t]; writeln (s[t]; t: = t − 1)
              end;
        cal:  begin {vytvorenie nového bloku}
                s[t + 1]: = báza (l); s[t + 2]: = : = b; s[t + 3]: = p;
                b: = t + 1; p: = a
              end;
        int:  t: = t + a;
        jmp:  p: = a;
        jpo:  begin if s[t] = 0 then p: = a; t: = t − 1
              end
      end {with, case}
    until p = 0;
    write ('END PL/0');
  end {interpret};
  begin {hlavný program}
    for ch: = 'A' to ';' do ssym [ch]: = nul;
    slovo [1]: = 'BEGIN';           slovo [2]: = 'CALL';
    slovo [3]: = 'CONST';           slovo [4]: = 'DO';
    slovo [5]: = 'END';             slovo [6]: = 'IF';
    slovo [7]: = 'ODD';             slovo [8]: = 'PROCEDURE';
    slovo [9]: = 'THEN';            slovo [10]: = 'VAR';
    slovo [11]: = 'WHILE';
    wsym [1]: = beginsym;           wsym [2]: = callsym;
    wsym [3]: = constsym;           wsym [4]: = dosym;
    wsym [5]: = endsym;             wsym [6]: = ifsym;
    wsym [7]: = oddsym;             wsym [8]: = procsym;
    wsym [9]: = thensym;            wsym [10]: = varsym;
    wsym [11]: = whilesym;
    ssym ['+']: = plus;             ssym ['−']: = minus;
    ssym ['*']: = krát;             ssym ['/']: = deleno;
    ssym ['(']: = ľzátvorka;        ssym [')']: = pzátvorka;
    ssym ['=']: = eql;              ssym [',']: = čiarka;
    ssym ['.']: = bodka;            ssym ['≠']: = neq;
    ssym ['<']: = lss;              ssym ['>']: = gtr;
    ssym ['≦']: = leq;              ssym ['≧']: = geq;
    ssym [';']: = bodkočiarka;
    mnemo [lit]: = 'LIT';           mnemo [opr]: = 'OPR';
    mnemo [lod]: = 'LOD';           mnemo [sto]: = 'STO';
    mnemo [cal]: = 'CAL';           mnemo [int]: = 'INT';
    mnemo [jmp]: = 'JMP';           mnemo [jpc]: = 'JPC';
    declbegsys: = [constsym, varsym, procsym];
```

```
statbegsys: = [beginsym, callsym, ifsym, whilesym];
facbegsys: = [identifikátor, číslo, ľzátvorka];
page (výstup); err: = 0;
cc: = 0; cx: = 0; ll: = 0; ch: = ' '; kk: = al; getsym;
blok (0, 0, [bodka] + declbegsys + statbegsys);
if sym ≠ bodka then error (9);
if err = 0 then interpret else write ('CHYBY V PL/0 PROGRAME');
99: writeln
end.
```

Cvičenia

5.1. Nech je daná nasledujúca syntax:

$$\begin{aligned} S &::= A \\ A &::= B \mid \textbf{if } A \textbf{ then } A \textbf{ else } A \\ B &::= C \mid B + C \mid + C \\ C &::= D \mid C * D \mid * D \\ D &::= x \mid (A) \mid - D \end{aligned}$$

Ktoré z uvedených symbolov sú terminálne a ktoré neterminálne? Určte množiny najľavejších a nasledujúcich symbolov $L(X)$ a $F(X)$ pre každý neterminálny symbol X. Zostrojte postupnosť krokov syntaktickej analýzy pre nasledujúce vety:

$x + x$
$(x + x) * (+ - x)$
$(x * - + x)$
if $x + x$ **then** $x * x$ **else** $- x$
if x **then if** $- x$ **then** x **else** $x + x$ **else** $x * x$
if $- x$ **then** x **else if** x **then** $x + x$ **else** x

5.2. Rozhodnite, či gramatika z cvičenia 5.1 spĺňa obmedzujúce tvrdenia 1 a 2 pre syntaktickú analýzu zhora nadol s predsnímaním jedného symbolu dopredu. Ak nie, nájdite ekvivalentnú syntax, ktorá uvedeným tvrdeniam vyhovuje. Vyjadrite túto syntax prostredníctvom syntaktického grafu a štruktúry údajov použitých v programe 5.3.

5.3. Zopakujte cvičenie 5.2 pre nasledujúcu syntax:

$$\begin{aligned} S &::= A \\ A &::= B \mid \textbf{if } C \textbf{ then } A \mid \textbf{if } C \textbf{ then } A \textbf{ else } A \\ B &::= D = C \\ C &::= \textbf{if } C \textbf{ then } C \textbf{ else } C \mid D \end{aligned}$$

Návod: Zistite, ktoré konštrukcie treba vylúčiť alebo nahradiť, aby sa dala aplikovať syntaktická analýza zhora nadol s predsnímaním jedného symbolu dopredu.

5.4. Uvažujte o probléme syntaktickej analýzy zhora nadol pre túto syntax:

$$\begin{aligned} S &::= A \\ A &::= B + A \mid DC \\ B &::= D \mid D * B \\ D &::= x \mid (C) \\ C &::= + x \mid - x \end{aligned}$$

Zistite maximálny počet symbolov, ktoré treba prezrieť dopredu, aby bolo možné analyzovať vety podľa tejto syntaxe.

5.5. Vykonajte transformáciu definície jazyka PL/0 vyjadrenej syntaktickými grafmi (*obr. 5.4*) do ekvivalentnej množiny prepisovacích pravidiel BNF.

5.6. Napíšte program, ktorý pre každý neterminálny symbol S z danej množiny prepisovacích pravidiel určí množiny začiatočných a nasledujúcich symbolov $L(S)$ a $F(S)$.
Návod: Použite časť programu 5.3 na vytvorenie vnútornej reprezentácie syntaxe v tvare štruktúry údajov. Potom vhodne manipulujte s takto pospájanou štruktúrou.

5.7. Rozšírte jazyk PL/0 a jeho kompilátor o nasledujúce konštrukcie:

a) Podmienený príkaz v tvare:

<príkaz> :: = **if** <podmienka> **then** <príkaz>
else <príkaz>

b) Príkaz cyklu v tvare:

<príkaz> :: = **repeat** <príkaz> {; <príkaz>}
until <podmienka>

Zistite, či uvedené rozšírenie jazyka spôsobí nejaké ťažkosti, ktoré by mohli viesť k zmene formy alebo interpretácie daných konštrukcií jazyka PL/0. Množinu inštrukcií počítača PL/0 však nesmiete rozšíriť o žiadne nové inštrukcie.

5.8. Rozšírte jazyk PL/0 a jeho kompilátor o možnosť používania parametrov procedúr. Zvážte dve možné alternatívy riešenia a na vašu realizáciu vyberte jednu z nich.

a) Parametre posielané hodnotou. Skutočnými parametrami pri vyvolaní procedúr sú výrazy, ktorých hodnoty sa priradia lokálnym premenným procedúr. Tieto lokálne premenné sú reprezentované formálnymi parametrami uvedenými v záhlaví deklarácií procedúr.

b) Parametre posielané referenciou. V tomto prípade sú skutočnými parametrami premenné, ktoré pri vyvolaní procedúr nahradia formálne parametre. Hodnotou parametra je teda adresa skutočného parametra, a nie jeho hodnota, ktorá sa zapíše na miesto formálneho parametra. Skutočné parametre sú potom prístupné nepriamo cez tieto adresy. Vidíme, že pomocou takýchto parametrov môžeme pristupovať k premenným mimo procedúr, v dôsledku čoho môžeme nasledujúcim spôsobom zmeniť pravidlo o rozsahu platnosti objektu: v každej procedúre sú priamo prístupné iba lokálne premenné; nelokálne premenné sú prístupné výlučne prostredníctvom parametrov.

5.9. Rozšírte jazyk PL/0 a jeho kompilátor o štruktúru pole. Predpokladajte, že rozsah indexov poľa *a* je určený v rámci jeho deklarácie nasledujúcim spôsobom:

var *a* (dolná : horná)

5.10. Upravte kompilátor jazyka PL/0 takým spôsobom, aby generoval kód pre váš počítač.
Návod: Generujte program v jazyku symbolických inštrukcií, aby ste sa vyhli problémom, ktoré by mohli nastať pri nedodržaní zásad spojovacieho a zavádzacieho programu vášho počítača. V prvom kroku sa vyhnite pokusom o optimalizáciu kódu. (Jedným z kandidátov na optimalizáciu je napr. mechanizmus prideľovania registrov.) Možné optimalizácie by sa mohli zaradiť do kompilátora až v priebehu štvrtého kroku jeho zjemňovania.

5.11. Rozšírte program 5.5 na program nazývaný „zúhľadňovač tlače". Cieľom tohto programu je prečítať text programu v jazyku PL/0 a vytlačiť ho v tvare, ktorý pekne zobrazuje textovú štruktúru vhodným oddelením riadkov a viacúrovňovým zarovnávaním. Na základe syntaktickej štruktúry jazyka PL/0 najprv definujte presné pravidlá oddeľovania riadkov a viacúrovňového zarovnávania, a potom ich implementujte prostredníctvom upravených príkazov pre zápis, ktoré vhodne zaraďte do programu 5.5. (Príkazy pre zápis musia byť potom, pochopiteľne, vyňaté z lexikálneho analyzátora jazyka PL/0.)

Zoznam použitej literatúry

5-1. Ammann, U.: The Method of Structured Programming Applied to the Development of a Compiler. International Computing Symposium 1973, A. Günther a kol eds. Amsterdam: North-Holland Publishing Co. (1974), s. 93—99.

5-2. Cohen, D. J. — Gotlieb, C. C.: A List Structure From of Grammars for Syntactic Analisis. Comp. Surveys, 2, No. 1 (1970), s. 65—82.

5-3. Floyd, R. W.: The Syntax of Programming Languages — A Survey. IEEE Trans., EC-13 (1964), s. 346—353.

5-4. Gries, D.: Compiler Construction for Digital Computers New York: Wiley (1971). (Slovenský preklad: Kompilátory číslicových počítačov. Bratislava, Alfa/SNTL 1981.)

5-5. Knuth, D. E.: Top-down Syntax Analisis. Acta Informatica, 1, No. 2 (1971), s. 79—110.

5-6. Lewis, P. M. — Stearns, R. E.: Syntax-directed Transduction. J. ACM, 15, No. 3 (1968), s. 465—488.

5-7. Naur, P. ed.: Report on the Algorithmic Language ALGOL 60. ACM, 6, No. 1 (1963), s. 1—17.

5-8. Schoore, D. V.: META II, A Syntax-oriented Compiler Writing Language. Proc ACM Natl. Conf., 19 (1964), D 1.3.1 — 11.

5-9. Wirth, N.: The Design of a PASCAL Compiler. Software-Practice and Experience, 1, No. 4 1971, s. 309—333.

A. MNOŽINA ZNAKOV ASCII

y \ x	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	|
13	cr	gs	−	=	M	]	m	}
14	so	rs	.	>	N	↑	n	~
15	si	us	/	?	O	—	o	del

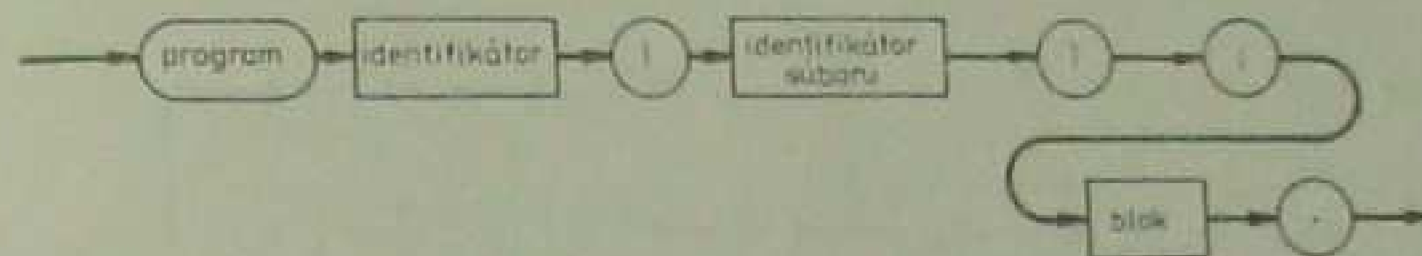
Poradové číslo znaku *ch* v uvedenej množine sa vypočíta pomocou jeho súradníc v tabuľke nasledujúcim spôsobom:

$$\text{ord}(ch) = 16 * x + y$$

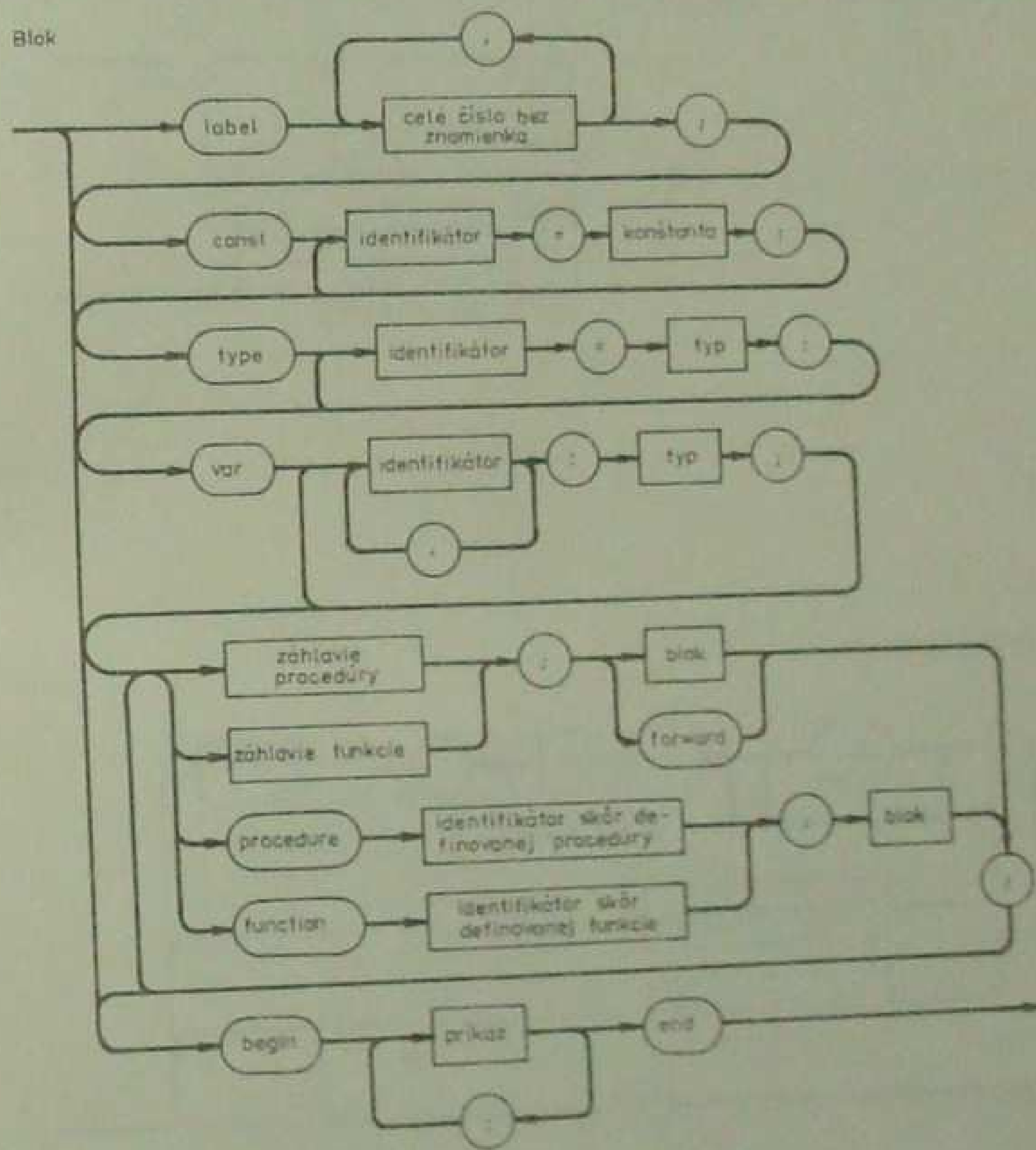
Znaky s poradovými číslami 0 až 31 a 127 sú riadiace znaky. Používajú sa pri prenose údajov a ovládaní periférnych zariadení počítača. Znak s poradovým číslom 32 je medzera.

B. SYNTAKTICKÉ DIAGRAMY JAZYKA PASCAL

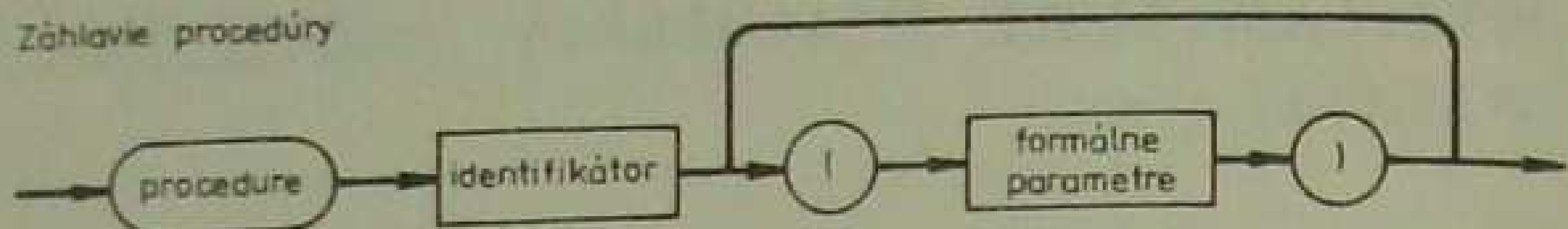
Program



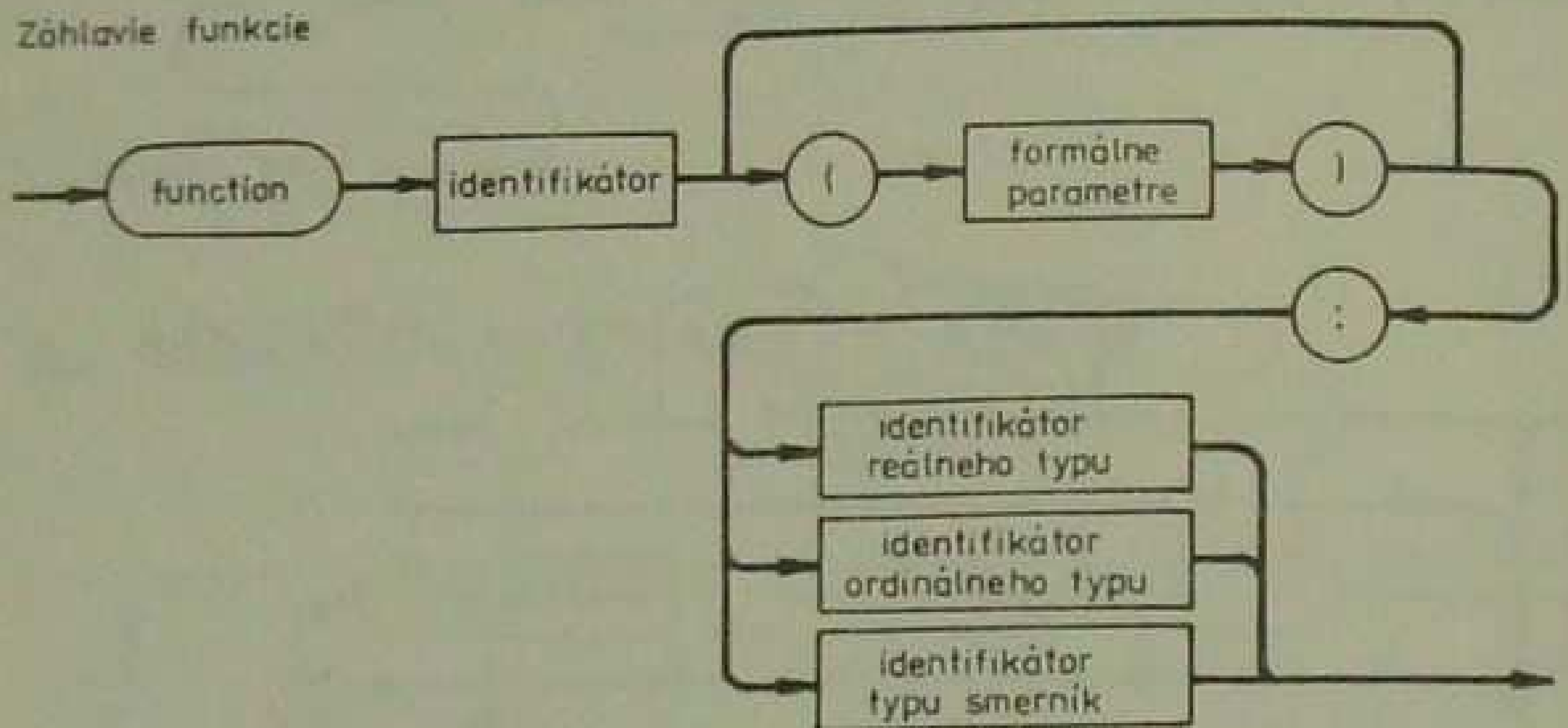
Blok



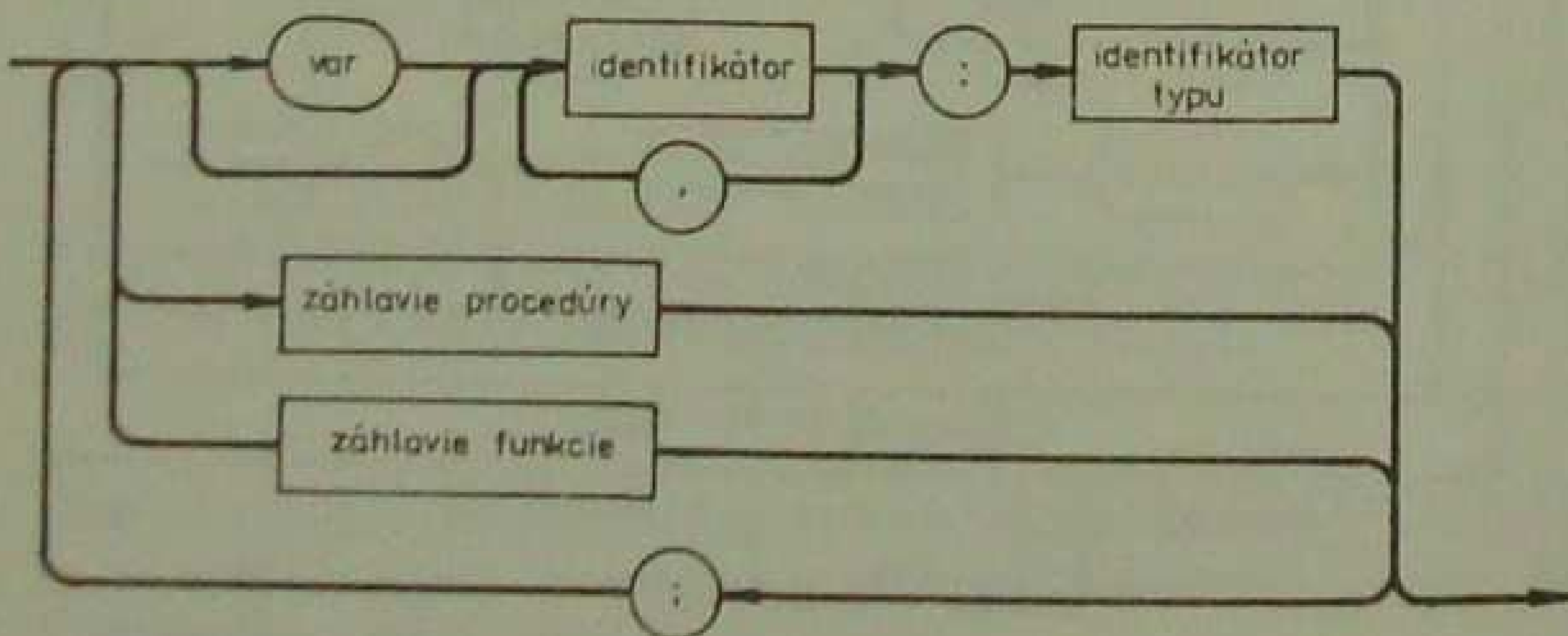
Záhlavie procedúry



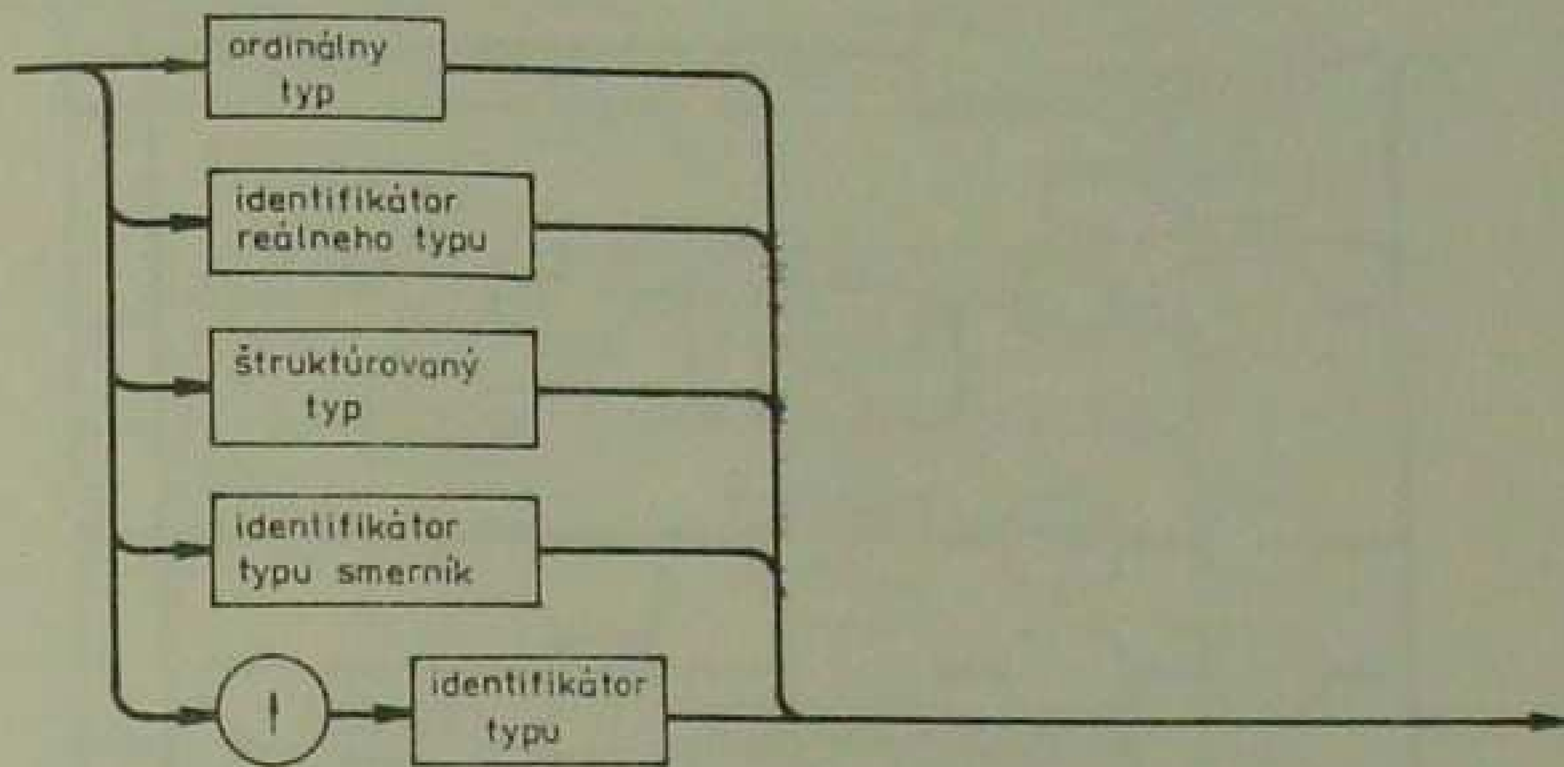
Záhlavie funkcie



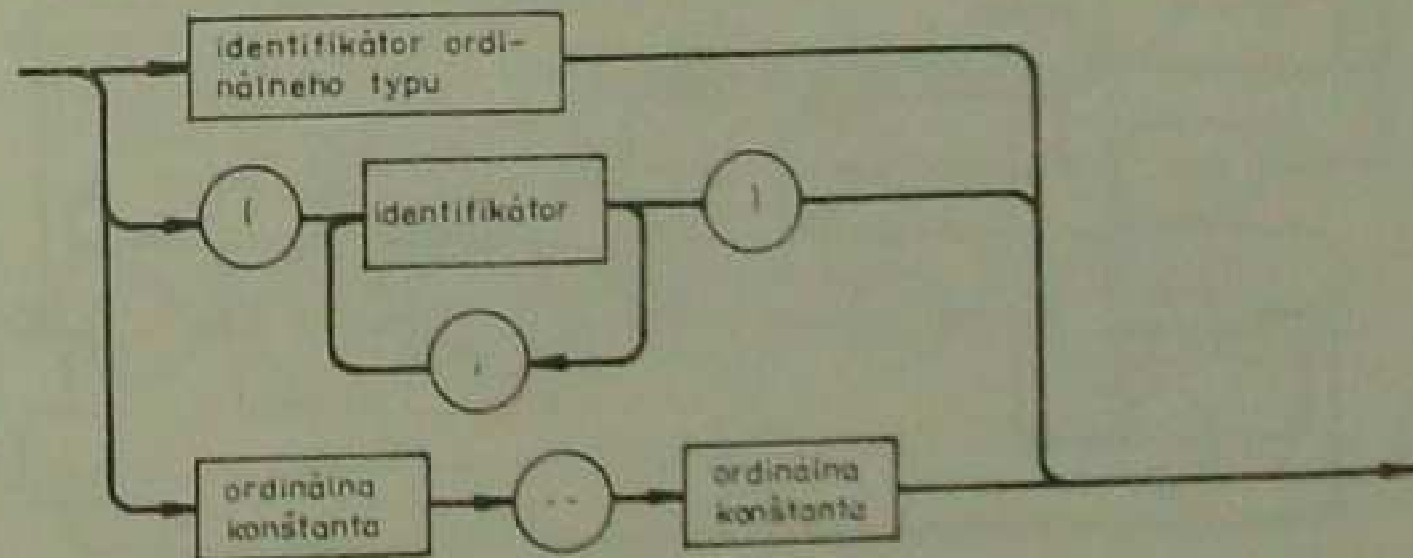
Formálne parametre



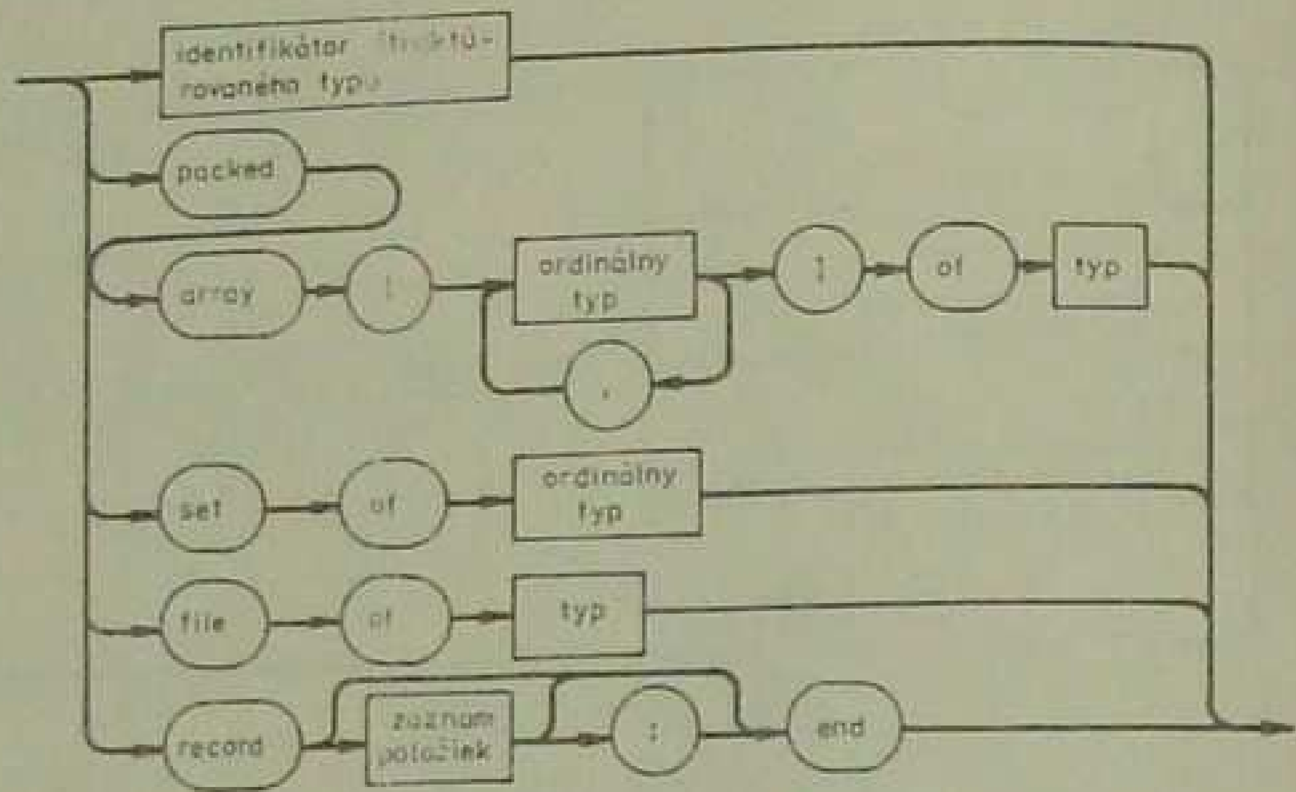
Typ



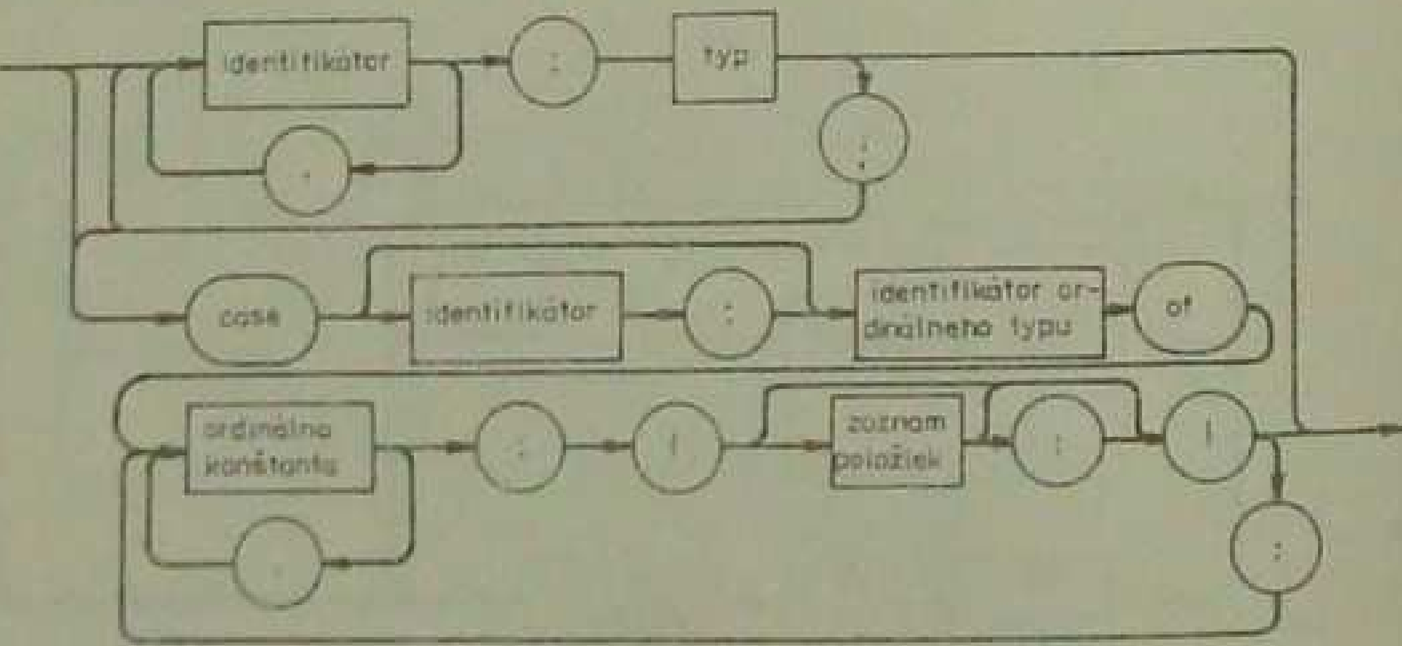
Ordinálny typ



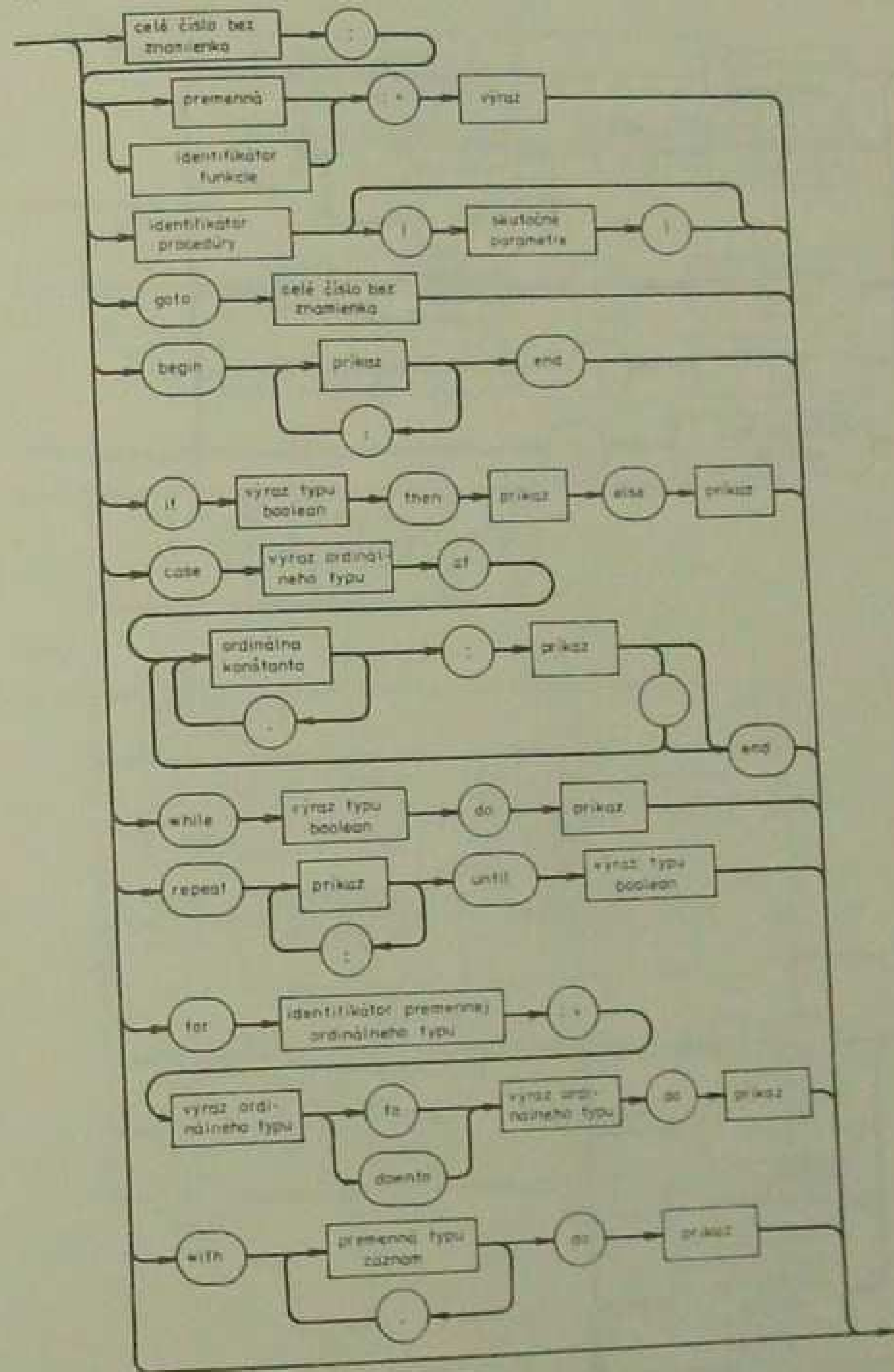
Štruktúrovaný typ



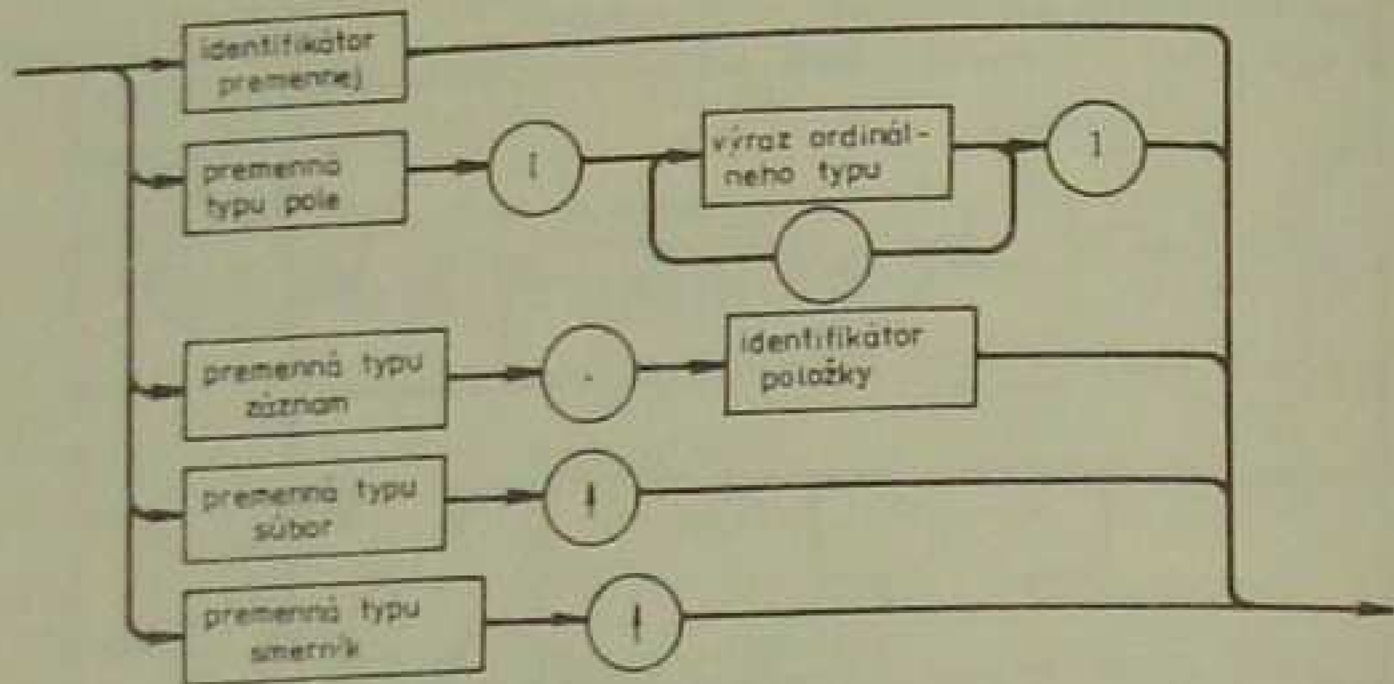
Zoznam položiek



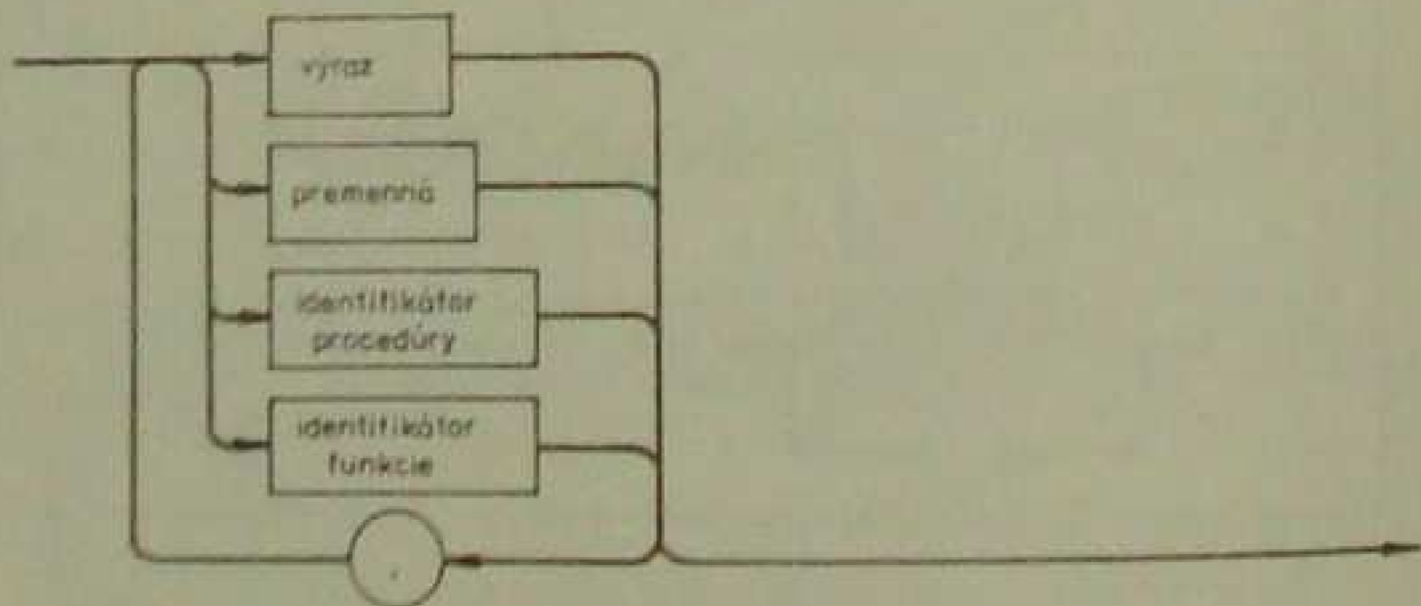
Príkaz



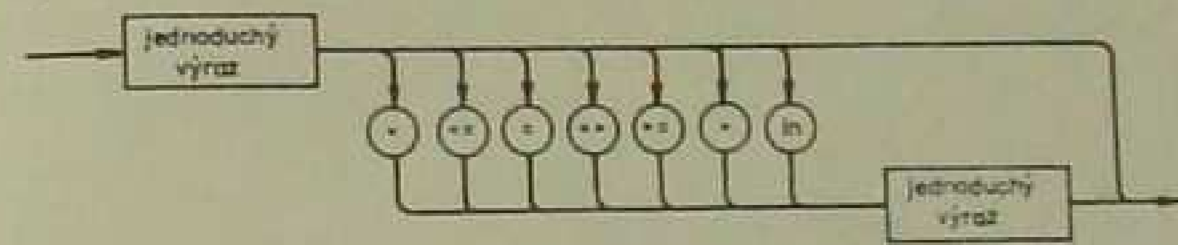
Premenná



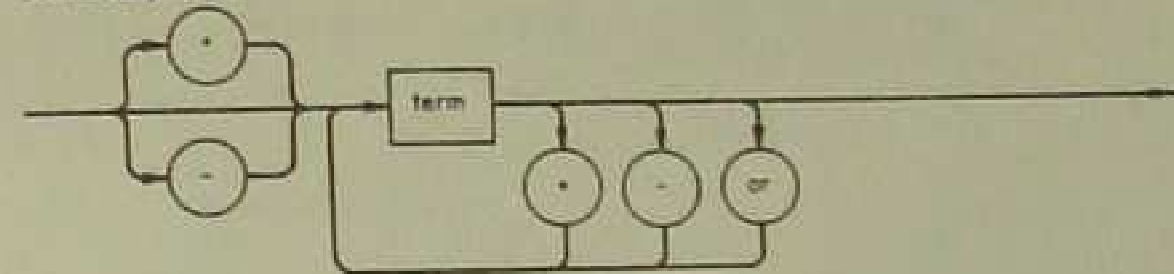
Skutočné parametre



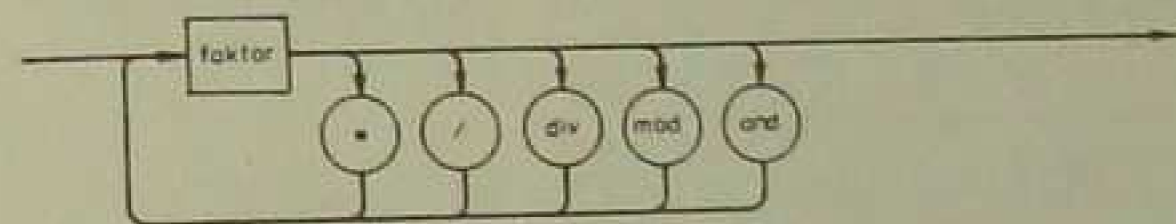
Výraz



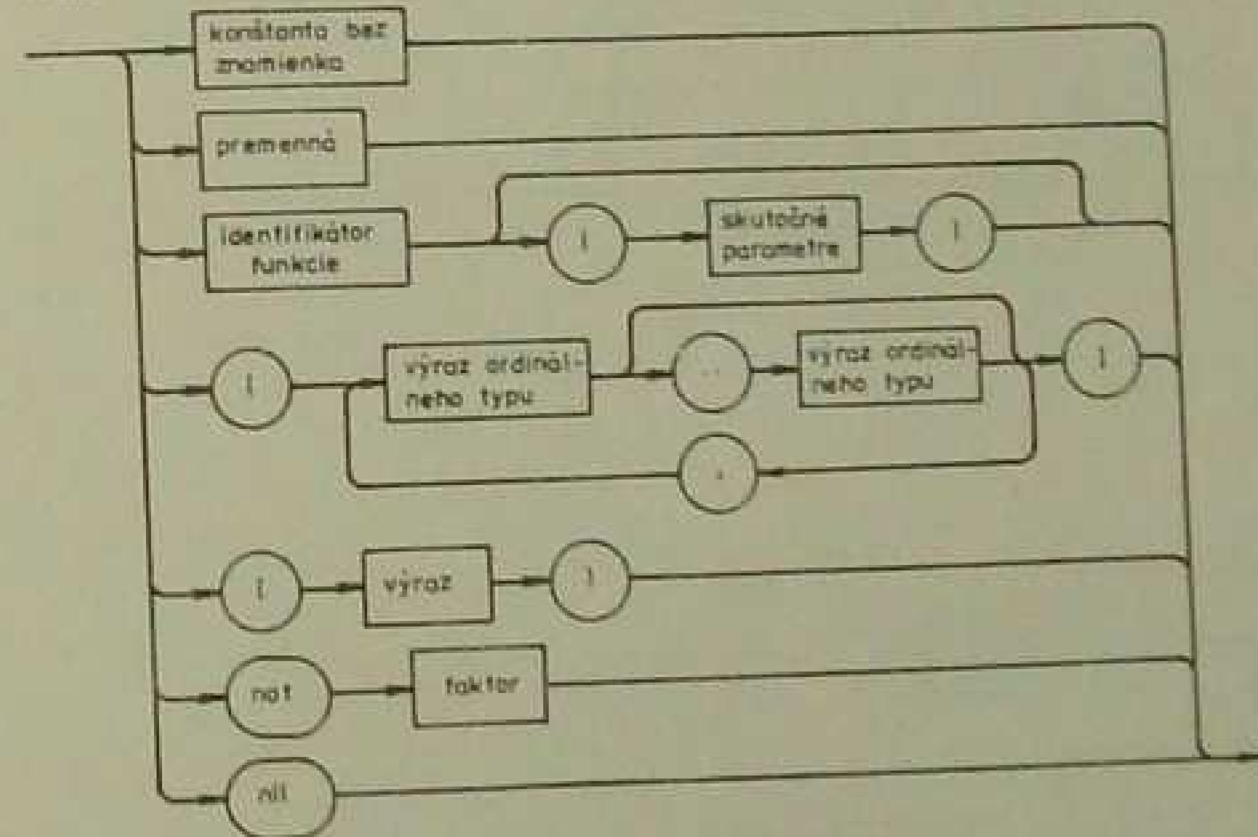
Jednoduchý výraz



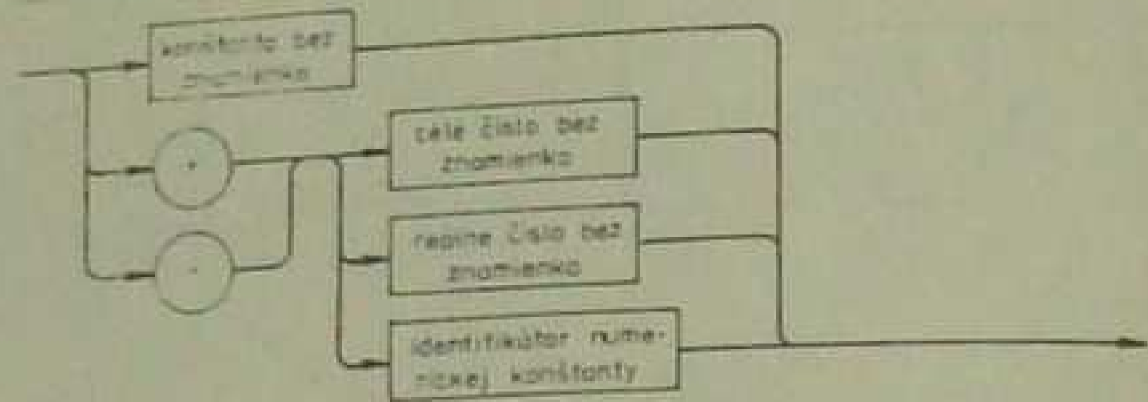
Term



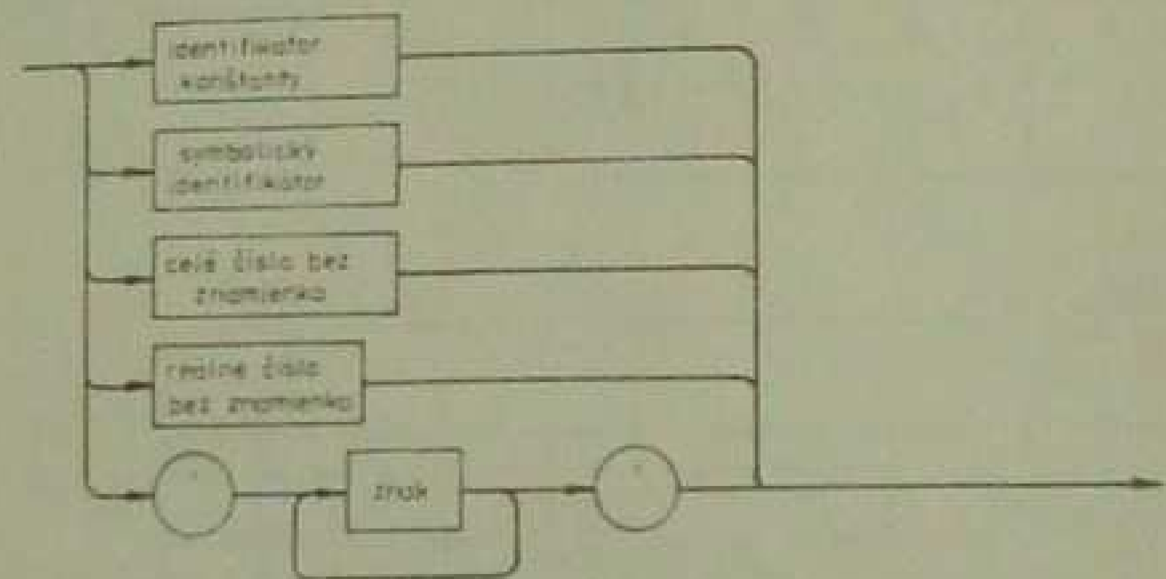
Faktor



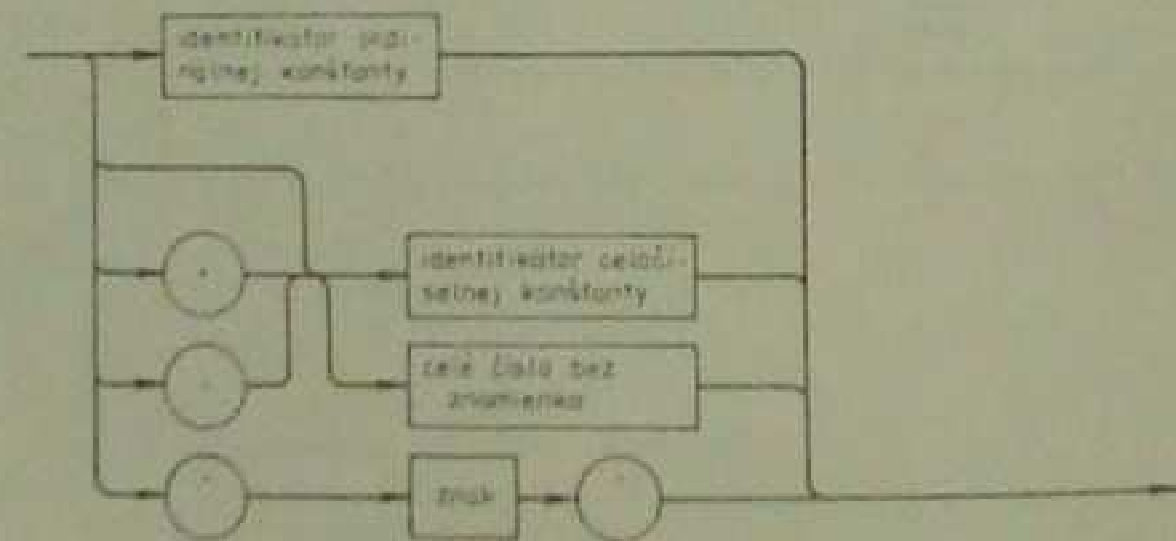
Konštanta



Konštanta bez znamienka

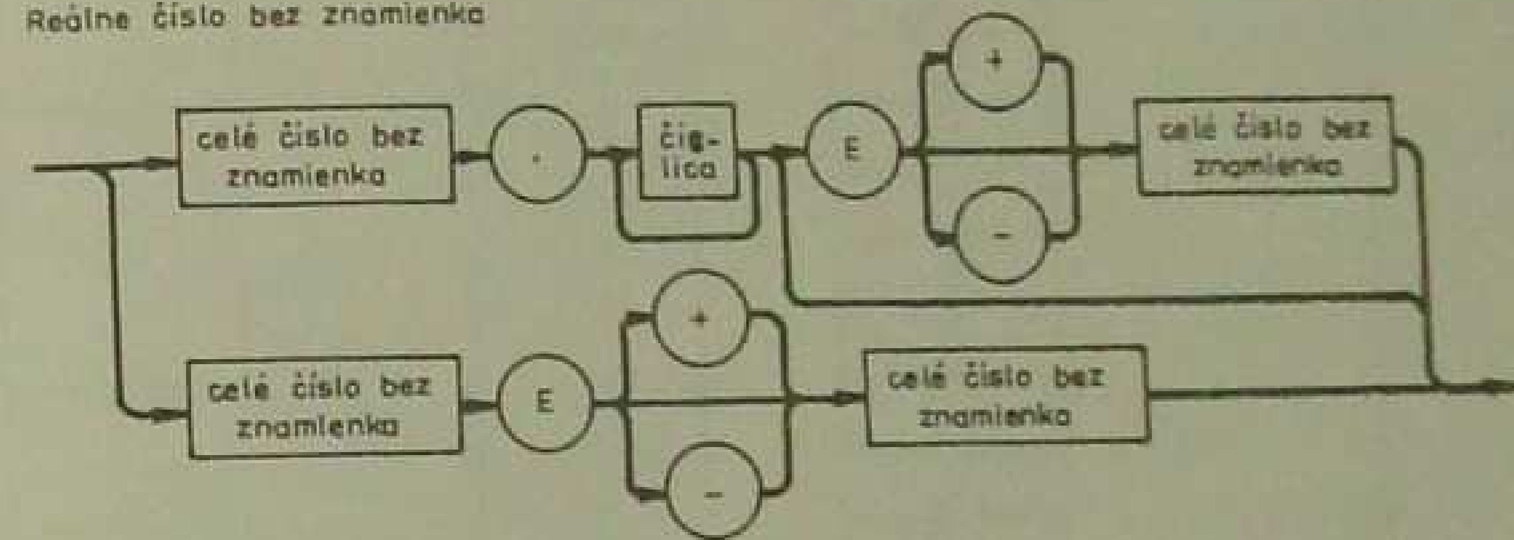


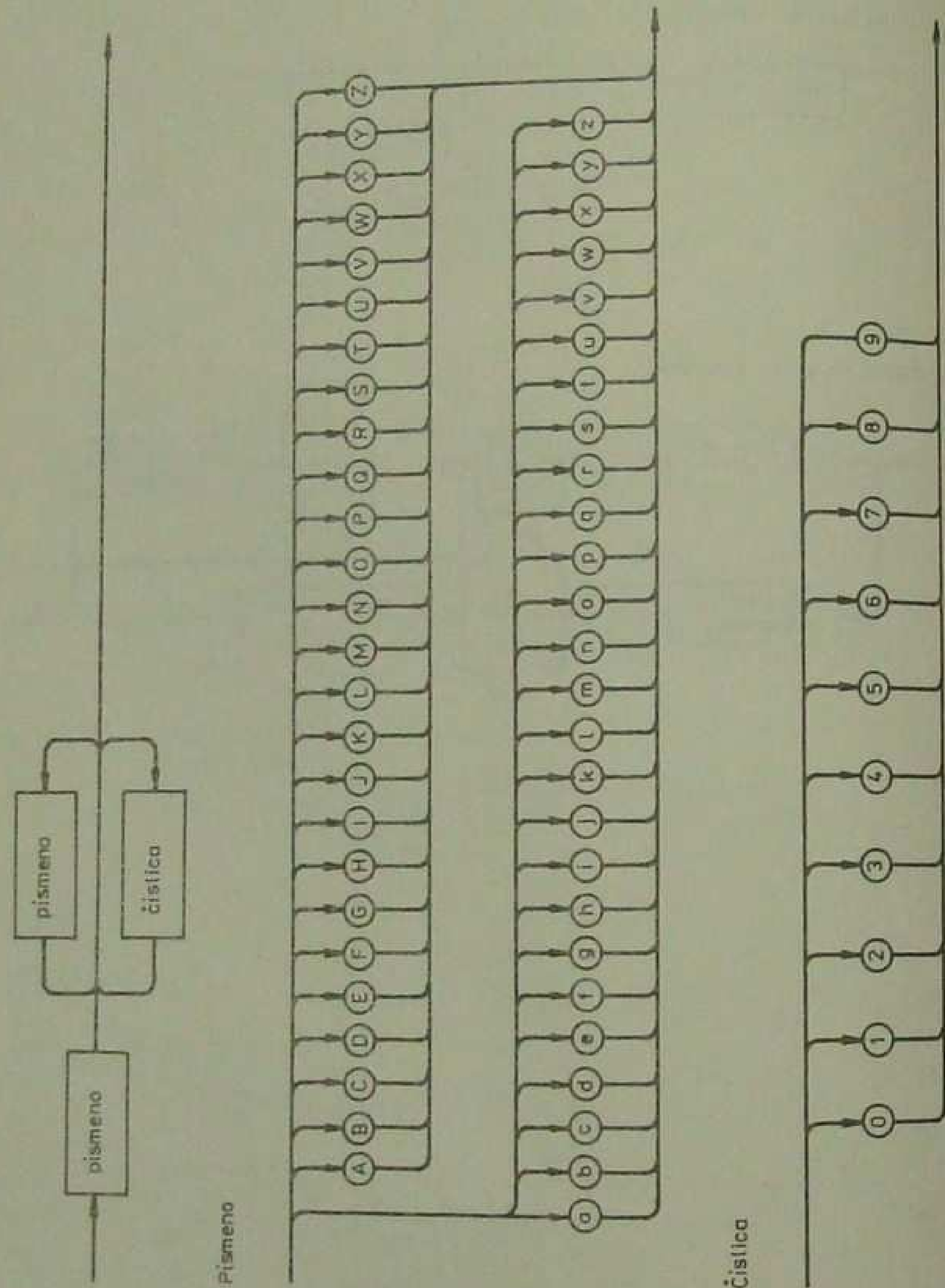
Ordinálna konštanta



Celé číslo bez znamienka

Reálne číslo bez znamienka





ZOZNAM PROGRAMOV

Vyhľadávanie v poli (1.15), (1.16)
Binárne vyhľadávanie (1.17)
Výpočet mocnín dvojky (Program 1.1)
Lexikálny analyzátor (Program 1.2)
Rozvrh hodín (1.28) až (1.30)
Zápis do súboru (1.52)
Čítanie súboru (1.53)
Zápis textového súboru (1.54)
Čítanie textového súboru (1.55)
Prečítanie reálneho čísla (Program 1.3)
Zápis reálneho čísla (Program 1.4)
Úprava textu (1.57) až (1.61)
Triedenie priamym vkladaním (Program 2.1)
Triedenie binárnym vkladaním (Program 2.2)
Triedenie priamym výberom (Program 2.3)
Bublinové triedenie (Program 2.4)
Shakersort (Program 2.5)
Shellov triediaci algoritmus (Program 2.6)
Vytvorenie haldy (Program 2.7)
Algoritmus triedenia haldou (Program 2.8)
Rozdelenie poľa na úseky (Program 2.9)
Quicksort (rekurzívna verzia) (Program 2.10)
Quicksort (nerekurzívna verzia) (Program 2.11)
Nájdenie *k*-tého prvku (Program 2.12)
Triedenie priamym zlučovaním (Program 2.13)
Triedenie prirodzeným zlučovaním (Program 2.14)
Triedenie vyváženým zlučovaním (Program 2.15)
Polyfázové triedenie (Program 2.16)
Distribúcia začiatočných behov (Program 2.17)
Hilbertove krivky (Program 3.1)
Sierpinského krivky (Program 3.2)

Pohyb šachového koňa (Program 3.3)
Osem dám (Program 3.4, 3.5)
Stabilné manželstvá (Program 3.6)
Optimálny výber (Program 3.7)
Generovanie zoznamu (4.13)
Prechod zoznamom (4.17)
Priame pridávanie prvkov do zoznamu (Program 4.1), (4.21) až (4.26)
Topologické triedenie (Program 4.2)
Vytvorenie dokonale vyváženého stromu (Program 4.3)
Prechod stromu (4.43) až (4.45)
Vyhľadávanie v strome (4.46), (4.47)
Stromové vyhľadávanie a pridávanie (Program 4.4)
Stromové vyhľadávanie a pridávanie (nerekurzívna verzia) (4.51)
Generátor krížových odkazov (Program 4.5)
Rušenie vrcholov v strome (4.52)
Vyhľadávanie a pridávanie do vyvážených stromov (4.63)
Rušenie vrcholov vo vyvážených stromoch (4.64)
Nájdenie optimálneho vyhľadávacieho stromu (Program 4.6)
Vyhľadávanie, pridávanie a rušenie v B-strome (Program 4.7)
Vyhľadávanie a pridávanie do zarovnaných stromov (4.87)
Generátor krížových odkazov (využívajúci transformačný tabuľku) (Program 4.8)
Syntaktický analyzátor (Program 5.1)
Tabuľkou riadený syntaktický analyzátor (5.11)
Syntaktický analyzátor jazyka (5.12), (Program 5.2)
Prekladač jazyka PL/0 (5.12), (Program 5.3)
Príklady programov v jazyku PL/0 (5.14) až (5.16)
Syntaktický analyzátor jazyka PL/0 (Program 5.4)
Syntaktický analyzátor jazyka PL/0 so systémom zotavenia sa z chýb (Program 5.5)
Kompilátor jazyka PL/0 (Program 5.6)

REGISTER

abstrakcia 19
Adelson-Velskii, G. M. 296
adresa 54
adresácia, otvorená 361
aktualizácia, výberová 35, 40
alfa (typ) 58
algol 16, 379
analýza, syntaktická 382
—, syntaktická cieľovo orientovaná 389
—, tabuľkou riadená 397
— zhora nadol 382
analyzátor, lexikálny 49, 419
ASCII 30, 72, 464
AVL-strom 296

B-strom 335
BB-strom 349
— —, binárny 349
— —, symetrický binárny 352
Backusova-Naurova forma (BNF) 379
Bayer, R. 335, 349, 350
beh 138
—, fiktívny 158
boolean 29

case (príkaz) 46

číslo, Fibonacciho 157, 185
—, harmonické 101

deklarácia typu 23
diagram závislosti 418
Dijkstra, E. W. 11
diskriminátor záznamu 44
distribúcia behov 138, 160, 167
div 28
dĺžka cesty, váhovaná 311
— slova 54
— vnútornej cesty 266
— vonkajšej cesty 267
doba, čakacia 70

eof (*f*) 67
eoln (*f*) 73
Euler, L. 18, 101, 295, 369

faktor, vyvažovací 300
— zaplnenia tabuľky 369
faktoriál 181
fáza 131
first (*x*) 63, 384
fixovanie inštrukcie 446
Floyd, R. W. 112
follow (množina nasledujúcich symbolov) 386
for (príkaz) 37
fortran 25, 183
front 238
funkcia, charakteristická 60
—, transformačná 360

Gauss, C. F. 203

generovanie zoznamu 239
get(*f*) 66
Gilstad, R. L. 154
Gotlieb, C. C. 318
graf, syntaktický 388

halda 111
Hilbert, D. 187
Hoare, C. A. R. 11, 116, 124
Hu, T. C. 317

char 29
chr(*x*) 31

ISO 30, 72
input 71
integer 28
interpret 440
invariant cyklu 35

jazyk 15, 21
—, bezkontextový 381
— PL/0, programovací 15, 411
—, programovací 21
—, rozšíriteľný 407

kardinalita 24, 26, 35, 39, 48
Knuth, D. E. 39, 101, 105, 108
kolízia 360
konkordancia 244
konštruktor 33, 39
korutina 171
krivka, Hilbertova 187
—, Sierpinskeho 191
krok 131

Landis, E. M. 296
list stromu 265
lokalita 411

matica 36
McCreight, E. 349
McVitie, D. G. 216

medián 124
metajazyk 402
metasymbol 379
metóda 24, 25
metóda štruktúrovania 25
množina, jednoprvková 48
—, potenčná 47
mod 28

new(*p*) 233
nil 234

oblasť preplnenia 361
obohacovanie, postupné 406
operácia, množinová 49
ord(*x*) 31
output 71

pack 58
pascal 12, 16, 23, 74, 127, 242
počítač PL/0 440
pokusy, kvadratické 363
—, lineárne 362
položka, rozlišovacia 44
postupnosť 63
—, jednoprvková 63
posunutie 59
potomok 331, 333
pravidlo kľúčového slova 428
—, kontextové prepisovacie 381
— obmedzenia 384, 385
—, prepisovacie 379
— úniku z chybového stavu 428
predchodca 265
predsnímanie 382
prehľadávanie s návratom 195, 384
prechod 131, 139
prechod stromu, priamy 275
— —, spätný 275
— —, vnútorný 275
prideľovanie pamäti, dynamické 232
prienik, množinový 49
prístup, náhodný 32

problém prázdneho reťazca 386
programy:
binárne vyhľadávanie 35
bublinové triedenie 102
čítanie súboru 68
čítanie textového súboru 73
distribúcia začiatočných behov 172
generátor krížových odkazov 286
generátor krížových odkazov (využívajúci transformačnú tabuľku) 364
generovanie zoznamu 240
Hilbertove krivky 190
kompilátor jazyka PL/0 447
lexikálny analyzátor 50
nájdenie a zobrazenie optimálneho vyhľadávacieho stromu 325
nájdenie *k*-tého najmenšieho prvku postupnosti 125
optimálny výber 222
osem dám 205, 208
Quicksort (nerekurzívna verzia) 120
Quicksort (rekurzívna verzia) 119
pohyb šachového koňa 199
polyfázové triedenie 164
prečítanie reálneho čísla 75
prechod stromu 276
prechod zoznamom 243
prekladač jazyka PL/0 407
priame pridávanie prvkov do zoznamu 245
príklady programov v jazyku PL/0 415
rozdelenie poľa na úseky 117
rozvrh hodín 52, 53
rušenie vrcholov v strome 290
rušenie vrcholov vo vyvážených stromoch 306

Shakersort 104
Shellov triediaci algoritmus 107
Sierpinskeho krivky 193
stabilné manželstvá 214
stromové vyhľadávanie a pridávanie 281
stromové vyhľadávanie a pridávanie (nerekurzívna verzia) 283
syntaktický analyzátor 395
syntaktický analyzátor jazyka 403
syntaktický analyzátor jazyka PL/0 419
syntaktický analyzátor jazyka PL/0 so systémom zotavenia sa z chýb 434
tabuľkou riadený syntaktický analyzátor 400
topologické triedenie 261
triedenie binárnym vkladaním 97
triedenie haldou 114
triedenie metódou priameho zlučovania 135
triedenie metódou vkladania 96
triedenie metódou výberu 100
triedenie prirodzeným zlučovaním 144
triedenie vyváženým zlučovaním 151
úprava textu 81—85
vyhľadávanie a pridávanie do vyvážených stromov 303
vyhľadávanie a pridávanie prvkov do zarovnaných stromov 356
vyhľadávanie, pridávanie a rušenie v B-strome 343
vyhľadávanie v poli 35
vyhľadávanie v strome 277

výpočet mocnín dvojky 37
vytvorenie dokonale vyváženého stromu 272
vytvorenie haldy 112
zápis do súboru 68
zápis reálneho čísla 78
zápis textového súboru 73
put (*f*) 66

Quicksort 116

read 67
readln 72
real 29
rekurzia 13, 180
reprezentácia údajov 20
reset (*f*) 66
rest (*x*) 63
reťazec 138
rewrite (*f*) 65
rozdelenie stránky 337
rozdiel, množinový 49

selektor 33
— poľa 33
— záznamu 40
sémantika 379
Shakersort 103
Shell, D. L. 106
schéma, programová 68, 73, 182, 201
—, rekurzívna 188
skalár (typ) 24
slovník 378
slovo 54
smerník (typ) 231
stránky stromu 334
strom 267
—, binárny 267
—, 2—3 349
—, Fibonacciho 297
— hĺbka 265
—, lexikografický 285
—, optimálny 313
strom prechodu 276
— SBB 352
— stupeň 266
—, viacestný 331
—, vyhľadávací 281
— vyhľadávanie 279
—, zarovnaný 354
súbor, indexový 32
—, segmentovaný 69
— s priamym prístupom 70
súčin, karteziánsky 38
symbol, neterminálny 379
—, terminálny 379
—, začiatočný 380

štruktúra, homogénna 32
— množina 47
— s variantmi, záznamová 43
štruktúrovanie súborov 68
štruktúry, spoločne používané 233

transformácia kľúča 358
— — so sekundárnou tabuľkou 361
triedenie, binárnym vkladaním 97
—, bublinové 102
— kaskádovým zlučovaním 178
—, polyfázové 153
— priamou výmenou 101
— priamym vkladaním 94
— priamym výberom 99
—, Shellove 106
—, stabilné 93
—, topologické 254
—, vnútorné 91
—, vonkajšie 91
— zlučovaním 129
Tucker, A. C. 317
typ, bázový 32
— indexu 32
—, rekurzívny 227
—, štandardný 24
— údajov 23

ťažisko 318

ukazovateľ, dynamický 441
—, statický 442
ukončenie výpočtu 182
unpack 58
úprava súborov 80
úroveň stromu 205
úsek poľa 117
usporiadanie, čiastočné 254, 256

variant záznamu 43
vetvenie s obmedzením 223
vyhľadávanie, binárne 35
vymenovanie 24
vypĺňovanie 56
výška stromu 265
vyváženie, dokonalé 271
vyváženosť (stromov) 296
vyvažovanie stránok 341

Walker, W. A. 318
Williams, J. 110
Wilson, L. B. 216
write 67
writeln 72

zápis, infixový 275
—, postfixový 275
—, prefixový 275
zarážka 35, 95, 246
závislosť jazyka, kontextová 418
zhusťovanie 57
zjednotenie, množinové 49
zjemňovanie, postupné 80
zlučovanie, jednofázové 131
—, priame 129
—, prirodzené 137
— stránok 343
—, vyvážené 146
znak, riadiaci 72, 464
zoznam 238
—, reorganizovaný 244
—, usporiadaný 244
— krížových odkazov 385
zreťazenie 63
—, priame 361