1. Typy a štruktúry údajov

1.1 ÚVOD

Základnými stavebnými prvkami každého programu sú algoritmus a údaje. Snáď preto, že údaje tvoria "pasívny" prvok, ktorý sa má transformovať "aktívnym" prvkom - algoritmom, venovala sa v programovaní podstatne väčšia pozornosť algoritmom ako údajom. V ostatnom období sa však situácia zmenila. Ukázalo sa, že aj "údajová" zložka programu môže mať veľký vplyv na jeho kvalitu.

Údaje, s ktorými program pracuje, netvoria izolované objekty, ale existujú medzi nimi väzby, alebo ich možno organizovať tak, aby medzi nimi vhodné väzby existovali. Hovoríme, že údaje majú určitú štruktúru. Je len prirodzené, že údaje organizujeme tak, aby vytvorená štruktúra mala vhodné vlastnosti.

Pojem štruktúra údajov nie je v programovaní nový. Naopak, stretávame sa s ním hneď od vzniku počítačov. Boli to práve počítače, a to ich pamäť, ktorá bola určujúcim faktorom pri organizovaní údajov. Štruktúry údajov sa obmedzovali iba na bit, slovo, prípadne pole slov. Stačili však na reprezentáciu toho najpodstatnejšieho - čísla a logickej hodnoty. Túto skutočnosť odrážali aj ďalšie časti počítača, ktoré zabezpečovali realizáciu programov prostredníctvom aritmeticko-logických a riadiacich operácií.

Veľmi skoro sa ukázalo, že tieto štruktúry sú nedostačujúce. Podnetmi pre vznik nových štruktúr boli ďalšie a ďalšie aplikácie výpočtovej techniky, jej rozvoj, ako aj pokroky v programovacích technikách. Zavedenie vonkajších pamätí si vyžadovalo novú zodpovedajúcu organizáciu údajov - súbory. Programovacie techniky dali podnet pre vznik štruktúr, ako sú viacrozmerné polia. Aplikácie počítačov v nenumerickej oblasti si vyžiadali vznik zoznamových štruktúr.

V pokroku v organizovaní údajov vidieť isté "vzďaľovanie sa" od štruktúr, ktoré poskytuje technické vybavenie počítačov a naopak "približovanie sa" k problémom, ktoré chceme na počítači riešiť. Tento "pohyb" prináša celý rad výhod z hľadiska riešenia úloh na počítači, no nie je "zadarmo". K novým organizáciam údajov už nemáme zodpovedajúce technické vybavenie, ktoré by nám realizovalo potrebné operácie nad týmito štruktúrami. Použitie takýchto štruktúr si vyžaduje:

a. reprezentovať objekty štruktúry v pamäti počítača, teda vyjadriť ich štruktúrami, ktoré poskytuje počítač,
b. vyjadriť operácie nad štruktúrami pomocou operácií, ktoré nám poskytuje počítač.

Je len prirodzené, že nás nezaujímajú iba samotné štruktúry údajov, ale aj operácie nad štruktúrami údajov. Realizáciou operácií nad štruktúrou dochádza k "zmene" štruktúry, ale táto zmena nie je ľubovoľná. Zvyčajne sa nemenia podstatné vlastnosti štruktúry - organizácia sa nemení. Napríklad operáciou sčítanie vektorov dostaneme opäť vektor. Pripúšťame teda iba také operácie, ktoré nemenia typ organizácie, typ štruktúry. Dostaneme sa tak k ďalšiemu dôležitému pojmu k typu údajov alebo údajovému typu. V praxi sa veľmi často používa okrem údajov synonymný termín dáta. Preto sa aj hovorí o typoch dát alebo dátových typoch. Norma ČSN pripúšťa obidva pojmy.

Pod typom údajov alebo údajovým typom budeme rozumieť množinu objektov nazývaných štruktúry údajov spolu s množinou operácií - funkcií, ktorých argumenty, resp. hodnoty sú štruktúry údajov.

Vlastnosti typov údajov sú charakterizované vlastnosťami operácií, ktoré sa vykonávajú s údajovými štruktúrami. Druhy operácií závisia od aplikačnej oblasti, v ktorej daný typ údajov používame. Vždy sú tam však operácie umožňujúce štruktúru vytvárať, meniť ju a používať.

Intuitívne možno v typoch údajov vyčleniť dve "úrovne":

a. úroveň odrážajúcu vlastnosti technických prostriedkov počítačov,
b. úroveň odrážajúcu vlastnosti problému spolu s použitými programovacími technikami.

Bezprostredná realizácia programov na počítači si vyžaduje použitie prvej úrovne, odrážajúcej vlastnosti technických prostriedkov. Ich priame použitie pri riešení reálnych problémov je však ťažkopádne a málo efektívne. V praxi volíme preto iný postup. Používame úroveň odrážajúcu vlastnosti problému, spolu s použitými programovacími technikami, a potom ju transformujeme do prostredia počítača. Reprezentujeme štruktúry údajov pomocou štruktúr údajov odrážajúcich technické prostriedky počítača a operácie vyjadríme pomocou operácií počítača. Tomuto procesu hovoríme implementácia.

Pod implementáciou typu údajov v inom, resp. pomocou iného typu údajov, budeme rozumieť spôsob reprezentácie štruktúr údajov a spôsob realizácie operácií typu údajov na takto reprezentovaných údajoch.

Typ údajov, pomocou ktorého reprezentujeme štruktúru údajov a spôsob realizácie operácií typu údajov na takto reprezentovaných údajoch, nazývame implementujúcim typom údajov a typ, ktorý implementujeme, nazývame implementovaným.

V bežnom programovaní sa s typmi údajov stretávame najčastejšie v programovacích jazykoch. Vyššie programovacie jazyky poskytujú celý rad štruktúr, resp. typov údajov. Aj keď vyššie programovacie jazyky neposkytujú rovnaké štruktúry resp. typy údajov, jedno majú v podstate všetky spoločné: implementácia prípustných údajových typov sa realizuje predkladačom. Používateľ ich preto môže používať bez toho, aby sa musel zaoberať ich implementáciou. Prináša to so sebou ďalšie výhody: údajové typy vyšších programovacích jazykov možno použiť ako implementujúce údajové typy. Takáto implementácia údajových typov je jasnejšia, čitateľnejšia. Takúto formu implementácie budeme ďalej používať aj my.

Praktické používanie typov údajov si vyžaduje, aby boli presne opísané - - špecifikované. Nestačí teda iba vymenovať prípustné operácie, ale musíme opísať aj ich sémantiku. Aj v tomto prípade je výhodné použiť niektorý vyšší programovací jazyk a v ňom realizovať údajový typ. Potom operácie implementovaného typu údajov sú opísané formou procedúr, programových modulov daného programovacieho jazyka. Takejto špecifikácii hovoríme operačná špecifikácia.

Výhodou operačnej špecifikácie je priama realizácia štruktúr údajov a prípustných operácií prekladačom použitého programovacieho jazyka. Na druhej strane má operačná špecifikácia nevýhodu v tom, že udáva iba jeden spôsob implementácie daného typu údajov: jednu reprezentáciu údajovej štruktúry, jeden spôsob realizácie každej operácie. Z hľadiska vlastnej špecifikácie údajového typu je v takejto špecifikácii veľa nepodstatných podrobností a údajový typ sa javí ako "prešpecifikovaný".

Druhý spôsob opisu typu údajov je axiomatická špecifikácia. V axiomatickej špecifikácii je sémantika operácii opísaná logickými formulami - axiómami. Z hľadiska typu údajov vyjadrujú axiómy podstatné vlastnosti operácií, a to nezávisle od toho, ako typ údajov sa bude implementovať. V axiomatickej špecifikácii sa teda sústreďujeme na to, čo jednotlivé operácie robia a robí sa abstrakcia od toho ako to robia, ako sú reprezentované štruktúry údajov a realizované operácie. Hovoríme preto o abstraktných typoch údajov alebo dát (ATD).

Základom teórie abstraktných typov údajov budeme venovať samostatnú časť. V súlade s touto teóriou budeme používať pojmový aparát pri špecifikácii typov, ktorými sa budeme zaoberať v ďalších častiach.

Aj v teórii ATD sa typ údajov chápe ako súhrn určitých objektov a operácií definovaných nad danými objektami. Pretože objekty v jednom type údajov môžu mať rôznu štruktúru, rozdeľujeme ich do niekoľkých množín - druhov. Operácie sú definované ako zobrazenia s presne určenými definičnými obormi a obormi hodnôt. Druhy, spolu s operáciami, tvoria tzv. viacdruhovú algebru a množiny mien jednotlivých druhov D a operácií R tvoria signatúru (D,R) tejto algebry. Signatúra nám umožňuje opísať syntax ľubovoľného typu dát, no chýba tu sémantika operácií. Ak sme už povedali, sémantiku operácií možno definovať zodpovedajúcou množinou axióm. Pridaním množiny axióm E k signatúre dostaneme špecifikáciu (D,R,E). V axiomatickej špecifikácii, ktorou sa v teórii ATD budeme zaoberať, majú axiómy tvar rovností dvoch výrazov, vytvorených z mien operácií a premenných. Pritom každá premenná je spojená s jedným, pevným druhom.

Pre zápis signatúry sa používajú dva formalizmy:

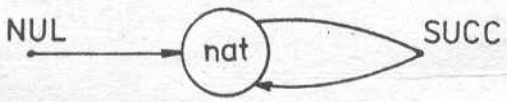
a. množinové zobrazenie, ako ho poznáme z teórie množín,
b. grafický zápis, kde
   druhy sa zobrazujú oválmi (kružnicami),
   operácie - názov sa zobrazuje vrcholom grafu
   - zobrazenie orientovanými hranami z druhu do druhu cez názov operácie, kde druh, z ktorého hrana vychádza tvorí definičný obor, do ktorého vstupuje je obor hodnôt.

Príklad 1.1

Ako typ údajov možno definovať napr. množinu prirodzených čísiel. Špecifikáciu typu údajov "prirodzené číslo" možno urobiť týmto spôsobom

a/ druhy : nat
   operácie: NUL : $\longrightarrow$ nat konštanta "nula"
   SUCC: nat $\longrightarrow$ nat operácia nasledovníka

b/



Obr. 1.1
Signatúra typu údajov "prirodzené číslo"

Z uvedeného príkladu vidieť, že každému prvku špecifikovaného typu údajov (každému prirodzenému číslu) možno priradiť aspoň jeden správne vytvorený výraz pozostávajúci z mien operácií tohto typu, a naopak, každému správne vytvorenému výrazu zodpovedá jedno prirodzené číslo. Napríklad prirodzené-

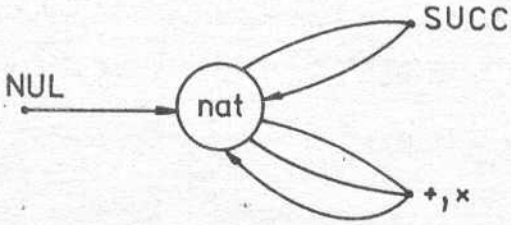
mu číslu 4 zodpovedá výraz SUCC(SUCC(SUCC(SUCC(0)))). Výrazu SUCC(SUCC(0)) zodpovedá číslo 2. Každý výraz je vlastne sprístupnením určitého prvku daného typu údajov. Prirodzene, že sprístupnenie musí existovať pre každý prvok typu údajov.

K operáciam typu údajov "prirodzené číslo" špecifikovaného v predchádzajúcom príklade sa zvyčajne pridávajú aj aritmetické operácie.

Príklad 1.2

Prirodzené čísla, ako typ údajov, možno špecifikovať aj týmto spôsobom:

druhy	: nat		
operácie:	NUL :	$\longrightarrow$ nat	konštanta "nula"
	SUCC:	nat $\longrightarrow$ nat	operácia nasledovníka
	+ :	nat,nat $\longrightarrow$ nat	sčítanie
	x :	nat,nat $\longrightarrow$ nat	násobenie



Obr. 1.2
Signatúra typu údajov "prirodzené číslo"
s aritmetickými operáciami

V druhom príklade treba venovať pozornosť dvom skutočnostiam: operácie + a x sú binárne, a teda v zápise zobrazenia vystupuje kartézsky súčin, v ktorom činitele oddeľujeme čiarkou. Druhou stránkou je spôsob vytvárania tohto typu údajov. Z hľadiska prvého príkladu možno tento typ údajov chápať ako obohatenie prvého typu údajov. Zložitejší typ údajov možno vytvoriť z už definovaného jednoduchšieho typu údajov obohatením množiny operácií. Takýchto spôsobov existuje viac:

a. faktorizácia - daný typ údajov doplníme novými axiómami,
b. obohatenie - daný typ údajov doplníme - novými druhmi,
- novými operáciami,
- novými axiómami
c. vytvorenie nových operácií z už definovaných operácií.

Príklad 1.3

Ako typ údajov možno definovať aj celé čísla. Signatúra tohto typu je takáto:

druhy : int
operácie: NUL : $\longrightarrow$ int — konštanta "nula"
PRED: int $\longrightarrow$ int — operácia predchodcu
SUCC: int $\longrightarrow$ int — operácia nasledovníka

Faktorizáciou možno definovať nový typ údajov, v ktorom budú okrem uvedených druhov (druhu) a operácií aj axiómy:

a1: PRED(SUCC(N))=N
a2: SUCC(PRED(N))=N

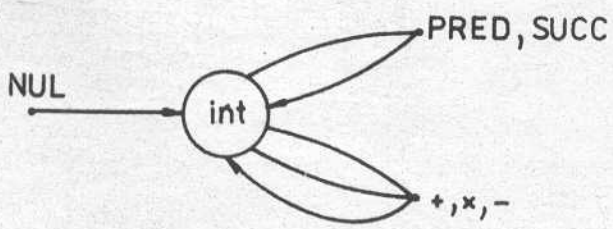
Zo vzniknutého typu údajov možno definovať nový obohatením, a to pridaním aritmetických operácií a axióm pre ich sémantiku. Ak budeme konštantu nula písať ako Ø, dostaneme:

a3 : +(N,Ø)=N
a4 : +(N,SUCC(M))=SUCC(+(N,M))
a5 : +(N,PRED(M))=PRED(+(N,M))
a6 : -(N,Ø)=Ø
a7 : -(N,SUCC(M))=PRED(-(N,M))
a8 : -(N,PRED(M))=SUCC(-(N,M))
a9 : x(N,Ø)=Ø
a10: x(N,SUCC(M))=+(x(N,M),N)
a11: x(N,PRED(M))=-(x(N,M),N)

pričom vlastná signatúra je:

druhy : int
operácie: NUL : $\longrightarrow$ int
PRED: int $\longrightarrow$ int
SUCC: int $\longrightarrow$ int
+ : int,int $\longrightarrow$ int
- : int,int $\longrightarrow$ int
x : int,int $\longrightarrow$ int

alebo graficky:



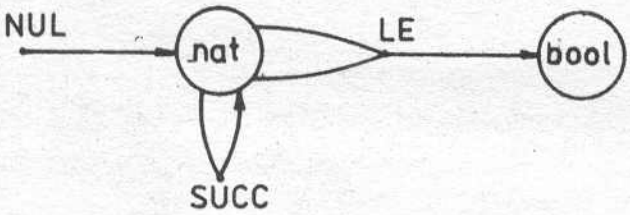
Obr. 1.3
Signatúra typu údajov "celé číslo"

Príklad 1.4

Uvažujme typ údajov "prirodzené číslo" definovaný v príklade 1.1. Nový typ údajov možno z neho získať pridaním relácie usporiadania LE, ktorá si "vyžaduje" rozšíriť aj druhy o bool, ako aj axiómy pre sémantiku pridanej operácie. Dostaneme:

druhy : nat
bool

operácie: NUL : $\longrightarrow$ nat
SUCC: nat $\longrightarrow$ nat
LE : nat,nat $\longrightarrow$ bool



Obr. 1.4
Signatúra typu údajov "prirodzené číslo"
s reláciou usporiadania

Zavedený pojmový aparát a formalizmus použijeme teraz na špecifikáciu a implementáciu najčastejšie používaných typov údajov. Pri implementácii budeme používať jazyk Pascal.

1.2 ŠPECIFIKÁCIA VYBRANÝCH TYPOV ÚDAJOV

1.2.1 Základné typy údajov

Základnými, alebo tiež primitívnymi typmi údajov rozumieme také typy údajov, ktorých špecifikácia nepoužíva iný typ údajov, s výnimkou typu "logická resp. boolovská hodnota", ktorý umožňuje definovať predikáty ako operácie špecifikovaných základných typov. Ide o veľmi jednoduché typy údajov, ktoré sú implementované v podstate v každom programovacom jazyku.

1.2.1.1 Boolovská hodnota

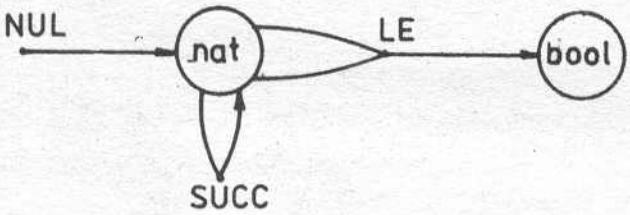
Typ údajov "boolovská hodnota" patrí k najjednoduchejším, ale veľmi používaným typom údajov.

Príklad 1.4

Uvažujme typ údajov "prirodzené číslo" definovaný v príklade 1.1. Nový typ údajov možno z neho získať pridaním relácie usporiadania LE, ktorá si "vyžaduje" rozšíriť aj druhy o bool, ako aj axiómy pre sémantiku pridanej operácie. Dostaneme:

druhy : nat
bool

operácie: NUL : $\longrightarrow$ nat
SUCC: nat $\longrightarrow$ nat
LE : nat,nat $\longrightarrow$ bool



Obr. 1.4
Signatúra typu údajov "prirodzené číslo"
s reláciou usporiadania

Zavedený pojmový aparát a formalizmus použijeme teraz na špecifikáciu a implementáciu najčastejšie používaných typov údajov. Pri implementácii budeme používať jazyk Pascal.

1.2 ŠPECIFIKÁCIA VYBRANÝCH TYPOV ÚDAJOV

1.2.1 Základné typy údajov

Základnými, alebo tiež primitívnymi typmi údajov rozumieme také typy údajov, ktorých špecifikácia nepoužíva iný typ údajov, s výnimkou typu "logická resp. boolovská hodnota", ktorý umožňuje definovať predikáty ako operácie špecifikovaných základných typov. Ide o veľmi jednoduché typy údajov, ktoré sú implementované v podstate v každom programovacom jazyku.

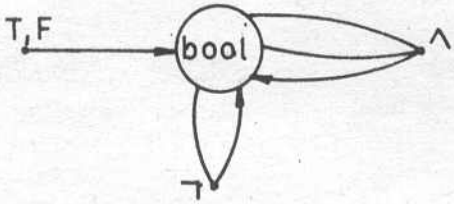
1.2.1.1 Boolovská hodnota

Typ údajov "boolovská hodnota" patrí k najjednoduchejším, ale veľmi používaným typom údajov.

Špecifikácia:

druhy : bool

operácie:	T : $\longrightarrow$ bool	konštanta - true
	F : $\longrightarrow$ bool	konštanta - false
	$\wedge$: bool,bool $\longrightarrow$ bool	logický súčin
	$\neg$: bool $\longrightarrow$ bool	negácia



Obr. 1.5
Signatúra typu údajov "boolovská hodnota"

Počet operácií možno podľa potreby rozšíriť.

1.2.1.2 Konečné typy údajov

Pod konečným typom údajov rozumieme ľubovoľnú konečnú množinu objektov a operácií. Každý objekt je pomenovaný a na množine objektov je definované usporiadanie. Okrem relácie usporiadania sa zvyčajne pridávajú ďalšie operácie.

Špecifikácia:

druhy : set

operácie:	s_i : $\longrightarrow$ set	i=1,2,...,n	pomenovanie konštanty jednoznačné
	$\leq$: set,set $\longrightarrow$ bool		usporiadanie
	SUCC: set $\longrightarrow$ set		
	PRED: set $\longrightarrow$ set		

Vzhľadom na to, že množina objektov je konečná, operácie SUCC, PRED sú čiastočné! Túto skutočnosť si treba pamätať pri implementácii. Ak operácia nie je pre nejaký argument definovaná, situácia sa ošetruje špeciálnym spôsobom.

Konečným typom je aj typ "logická hodnota". Ako konečné typy údajov možno definovať napr. "desiatková číslica", "abecedný znak", "alfanumerický znak" a pod.

1.2.2 Zložené typy údajov

Zložený typ údajov je taký, ktorý pri svojej konštrukcii používa iný typ (iné typy) údajov.

1.2.2.1 Podtyp

Najjednoduchšou konštrukciou zloženého typu údajov je podtyp. Ide o typ údajov, ktorý konštruujeme z daného typu T tak, že zoberieme ľubovoľnú podmnožinu objektov typu T spolu so všetkými operáciami zúženými na túto podmnožinu.

Špecifikácia:

druhy : P		$P \subseteq T$
operácie: I	: $P \longrightarrow T$	vnorenie P do T
	I^{-1} : $T \longrightarrow P$	identická operácia na prvkoch z P, ináč indikuje chybu
	isin P: $T \longrightarrow$ bool	test na príslušnosť objektu v P. Výsledok je "true", ak je objekt v P, ináč je výsledok "false".

Zúženie operácie f na podtyp P typu T možno urobiť takto:

Ak $f: T,\bar{X} \longrightarrow T$ je operácia typu údajov T, kde $\bar{X}$ je kartézsky súčin množín rôznych od T, potom jej zúženie f/P na množinu P je definované takto:

$$f/P(p,\bar{x}) = I^{-1}(f(I(p),\bar{x})) \quad \text{pre všetky } p \in P \text{ a } \bar{x} \in \bar{X}.$$

Príkladom podtypu je napr. interval prirodzených čísel ako podtyp typu celých čísel.

1.2.2.2 Kartézsky súčin

Kartézsky súčin typov $T_1,T_2,\ldots,T_n$ je množina $T = T_1 \times T_2 \times \ldots \times T_n$ spolu s operáciami

cons : $T_1,T_2,\ldots,T_n \longrightarrow T$		vytvorenie n-tice,
$proj_i$: $T \longrightarrow T_i$	$i=1,2,\ldots,n$	i-ta projekcia, pre n-ticu vyberie jej i-tu zložku.

1.2.2.3 Zjednotenie typov

Zjednotenie typov údajov $T_1, T_2, \ldots, T_n$ je množina $T = T_1 \cup T_2 \cup \ldots \cup T_n$ spolu s operáciami:

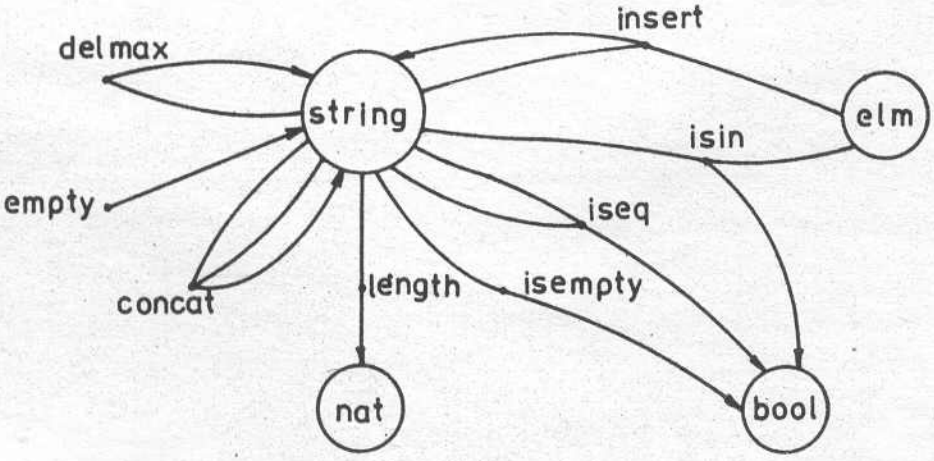
I_i	$: T_i \longrightarrow T$	$i=1,2,\ldots,n$	Vnorenie množiny T_i do T
I_i^{-1}	$: T \longrightarrow T_i$	$i=1,2,\ldots,n$	Operácia totožná s identitou na prvkoch z T_i, ináč chyba
isT_i	$: T \longrightarrow bool$	$i=1,2,\ldots,n$	Ak je prvok typu T_i, potom "true", ináč "false".

1.2.3 R e ť a z e c

Pod reťazcom rozumieme usporiadanú n-ticu prvkov danej množiny. Počet prvkov v reťazci určuje jeho dĺžku. Reťazec s nulovou dĺžkou nazývame prázdny reťazec.

Špecifikácia:

druhy	: string			
	elm			
	bool			
	nat			
operácie:	concat	: string,string	$\longrightarrow$ string	zreťazenie reťazcov
	empty	:	$\longrightarrow$ string	vytvorenie prázdneho reťazca
	length	: string	$\longrightarrow$ nat	určenie dĺžky reťazca
	insert	: string,elm	$\longrightarrow$ string	pridanie prvku k reťazcu
	delmax	: string	$\longrightarrow$ string	vylúčenie maximálneho prvku
	isin	: string,elm	$\longrightarrow$ bool	test, či je prvok v reťazci



Obr. 1.6
Signatúra typu údajov "reťazec"

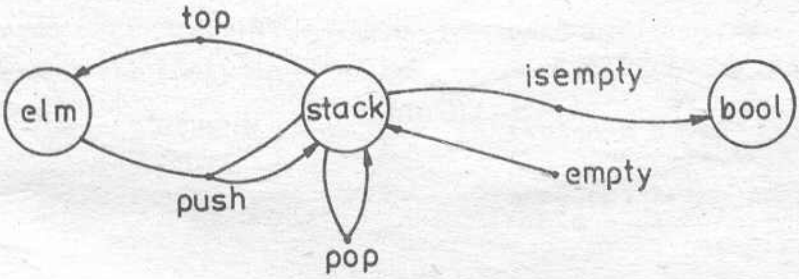
iseq	: string,string ⟶ bool		test, či sa dva reťazce rovnajú
isempty:	string ⟶ bool		test, či je reťazec prázdny

1.2.4 Z á s o b n í k

Zásobník je typ údajov, ktorý možno zo statického hľadiska stotožniť s reťazcom. Ide tiež o postupnosť prvkov, ktorú však možno meniť iba na jednom konci. Prvky možno pridávať alebo vyberať iba na jednom konci, ktorý nazývame vrchom zásobníka. Spôsob práce so zásobníkom najlepšie charakterizuje veta "posledný dnu, prvý von". Často sa preto označuje názvom LIFO (Last In First Out - posledný dnu, prvý von).

Špecifikácia:

druhy	: stack		
	elm		
	bool		
operácie:	push	: stack,elm ⟶ stack	pridanie prvku do zásobníka
	top	: stack ⟶ elm	výber prvku zo zásobníka
	pop	: stack ⟶ stack	zrušenie prvku z vrchu v zásobníku
	empty	: ⟶ stack	konštrukcia zásobníka (prázdneho)
	isempty:	stack ⟶ bool	test, či je zásobník prázdny



Obr. 1.7
Signatúra typu údajov "zásobník"

axiómy: a1: top(push(s,e))=e
a2: pop(push(s,e))=s
a3: isempty(empty)=true
a4: isempty(push(s,e))=false
a5: pop(empty)=~~empty~~ error
a6: top(empty)=error

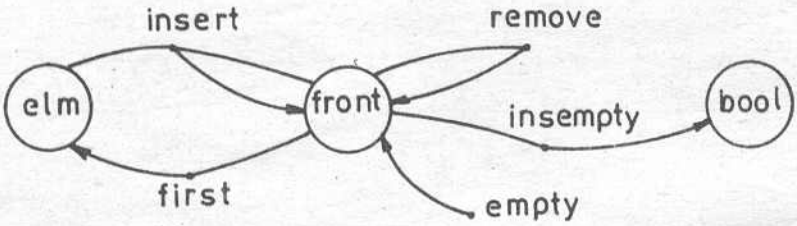
Axiómy určujú sémantiku operácií a ľahko z nich vyčítame spôsob manipulácie so zásobníkom. Operácia push pridáva prvok na vrch zásobníka. Operácia pop "ruší" prvok na vrchu zásobníka a novým vrchom sa stáva prvok, ktorý bol vložený do zásobníka pred prvkom, ktorý rušíme. Operácia top vyberá prvok z vrchu zásobníka a novým vrchom sa stáva prvok vložený pred ním. Operácia empty vytvára prázdny zásobník. Operácia isempty je test, či je zásobník prázdny. Ak je prázdny, výsledkom je true, ináč false.

1.2.5 F r o n t

Typ údajov front je príbuzný typu údajov zásobník. Spôsob práce s frontom možno charakterizovať vetou "prvý dnu, prvý von". Označuje sa názvom FIFO (First In First Out - prvý dnu, prvý von).

<u>Špecifikácia:</u>

```
druhy    : front
           elm
           bool
operácie: insert : front,elm ---> front     pridanie prvku do frontu
          remove : front     ---> front     zrušenie prvku vo fronte
          first  : front     ---> elm       výber prvku z frontu
          empty  :           ---> front     konštrukcia frontu - prázdneho
          isempty: front     ---> bool      test, či je front prázdny
```



Obr. 1.8
Signatúra typu údajov "front"

```
axiómy: a1: isempty(empty)=true
        a2: isempty(insert(f,e))=false
        a3: first(empty)=error
        a4: first(insert(f,e))=if isempty(f) then e else first(f)
        a5: remove(empty)=error
        a6: remove(insert(f,e))=if isempty(f) then empty else
                                insert(remove(f),e)
```

V praxi sa stretávame aj s typom údajov, ktorý je istou modifikáciou frontu v tom, že pridávanie a výber prvkov možno robiť na obidvoch koncoch. Niekedy sa takémuto typu údajov hovorí balík alebo obojsmerný front.

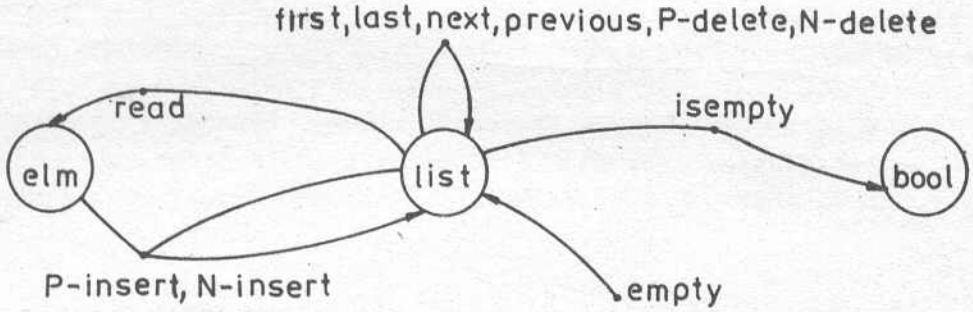
1.2.6 Zoznam

Zoznam patrí do tej istej skupiny typov údajov ako front a zásobník a možno ho považovať za určité zovšeobecnenie obidvoch týchto typov údajov. Zoznam je postupnosť prvkov, ktorú možno meniť na ľubovoľnom mieste. Zoznam je sprístupniteľný alebo "viditeľný" cez vyhradený prvok. Cez tento prvok je sprístupniteľné ľubovoľné miesto v zozname, čo umožňuje urobiť na ľubovoľnom mieste požadovanú zmenu. Sprístupnenie sa robí smerníkmi.

Špecifikácia:

druhy	: list		
	elm		
	bool		
operácie:	empty :	⟶ list	vytvorenie zoznamu prázdneho
	first : list	⟶ list	nastavenie smerníka na začiatok zoznamu
	last : list	⟶ list	nastavenie smerníka na koniec zoznamu
	next : list	⟶ list	presun smerníka na nasledujúci prvok
	previous: list	⟶ list	presun smerníka na predchádzajúci prvok
	read : list	⟶ elm	prečítanie prvku
	P-insert: list,elm	⟶ list	vloženie nového prvku pred prvok označený smerníkom
	N-insert: list,elm	⟶ list	vloženie nového prvku za prvok označený smerníkom
	P-delete: list	⟶ list	zrušenie prvku označeného smerníkom a označenie predchádzajúceho prvku smerníkom
	isempty : list	⟶ bool	test, či je zoznam prázdny
	N-delete: list	⟶ list	zrušenie prvku označeného smerníkom a označenie nasledujúceho prvku smerníkom

V praxi sa často stretávame so zoznamami, ktoré majú zložitejšiu štruktúru, ako sme si špecifikovali. Signatúra z obr. 1.9 umožňuje vytvárať iba špeciálny typ zoznamu tzv. lineárny zoznam, v ktorom sú všetky prvky z hľadiska zoznamu na jednej úrovni a z hľadiska operácii nad zoznamom sú nedeliteľné. Hovorí sa im atómy.



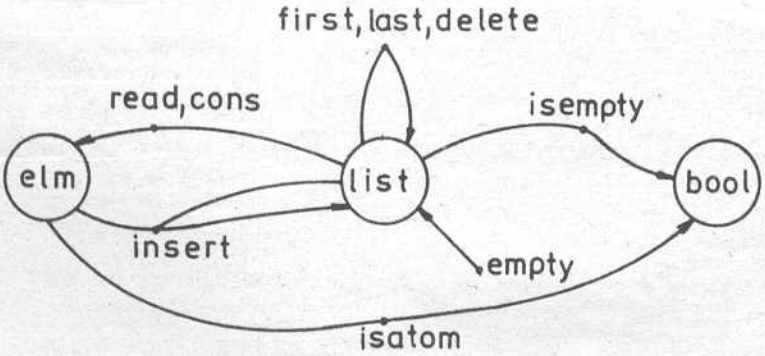
Obr. 1.9
Signatúra typu údajov "zoznam"

Vytvorenie hierarchickej zoznamovej štruktúry, t.j. zoznamu, v ktorom môžu byť prvky opäť zoznamy, vyžaduje si rozšíriť operácie tak, aby odrážali túto skutočnosť. Pochopiteľne, že to bude mať vplyv aj na implementáciu.

Signatúra typu údajov zoznam s hierarchickou štruktúrou bude vychádzať zo signatúry pre lineárny zoznam, do ktorej pridáme operáciu umožňujúcu testovať, či prvok je zoznam alebo atóm a operáciu na vytvorenie prvku zoznamu, ktorým bude samotný zoznam. Operácie zo signatúry pre lineárny zoznam zostávajú nezmenené. Treba si však uvedomiť, že vkladanie, vyberanie, rušenie a manipulovanie s prvkom zoznamu sa môže týkať opäť zoznamu. Preto napr. operácie vkladania a rušenia prvkov sa "zužujú" na prvok označený smerníkom.

Signatúra typu údajov zoznam s hierarchickou štruktúrou je teda takáto: (Obr. 1.10), kde operácia cons a isatom sú definované:

cons : list ⟶ elm
isatom: elm ⟶ bool



Obr. 1.10
Signatúra typu údajov "zoznam s hierarchickou štruktúrou"

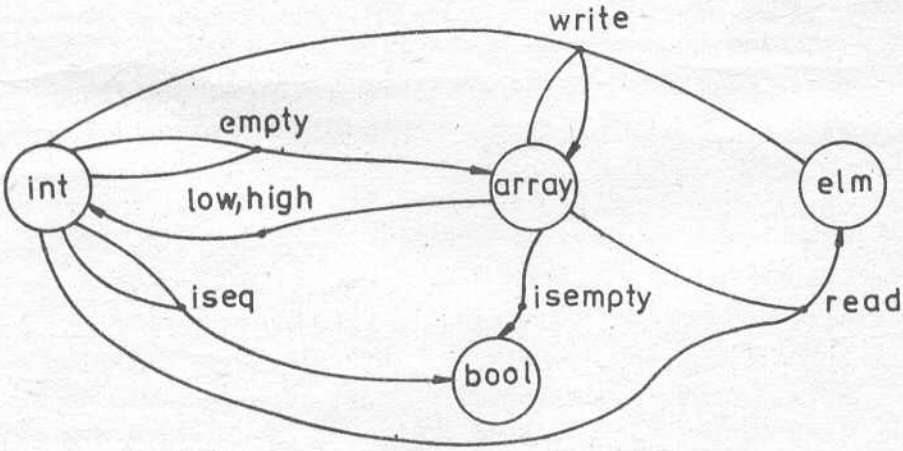
1.2.7 P o l e

S typom údajov pole sa v praxi stretávame najčastejšie, pretože nám ho poskytuje v určitej forme v podstate každý vyšší programovací jazyk. Naviac, v zjednodušenom tvare, ako jednorozmerné pole má na druhej strane svoj ekvivalent v organizácii pamäti počítača, čo vedie k veľmi jednoduchej implementácii.

Poľom rozumieme množinu prvkov identifikovateľných pomocou n-tice iných prvkov, ktoré nazývame indexy prvkov poľa. Číslo n sa nazýva rozmer poľa. Ako indexy sa zvyčajne používajú celé čísla. Každý index môže nadobúdať hodnoty iba z určitej množiny hodnôt. Je ohraničený dolnou a hornou hranicou. Keďže existuje jednoznačné zobrazenie n-rozmerného poľa na jednorozmerné, uvedieme si špecifikáciu jednorozmerného poľa.

Špecifikácia:

druhy	: array			
	int			
	elm			
	bool			
operácie:	empty	: int,int	⟶ array	konštrukcia poľa - prázdneho
	read	: array,int	⟶ elm	čítanie daného prvku poľa
	write	: array,int,elm	⟶ array	zápis daného prvku poľa na dané miesto
	low	: array	⟶ int	dolná hranica indexu poľa
	high	: array	⟶ int	horná hranica indexu poľa
	isempty	: array	⟶ bool	test, či je pole prázdne
	iseq	: int,int	⟶ bool	test, či sa dva indexy rovnajú



Obr. 1.11
Signatúra typu údajov "pole"

axiómy: a1: read(empty(i,j),k)=error 1
a2: read(write(a,i,e),j)=<u>if</u> iseq(i,j) <u>then</u> e <u>else</u> read(a,j)
a3: write(a,i,e)=<u>if</u> i < low(a) ∨ i > high(a) <u>then</u> error 2
<u>else</u> write(a,i,e)
a4: low(empty(i,j))=i
a5: high(empty(i,j))=j
a6: low(write(a,i,e))=low(a)
a7: high(write(a,i,e))=high(a)
a8: isempty(empty(m,n))=<u>true</u>
a9: isempty(write(empty(i,j),i,e))=<u>false</u>

K viacrozmernému poľu možno zo špecifikovaného poľa prejsť aj tak, že ako prvky budeme uvažovať jednorozmerné polia, v čom môžeme ísť do ľubovoľnej hĺbky.

V uvedenej špecifikácii pracujeme v podstate s dvoma typmi údajov: indexovým typom údajov a typom údajov určujúcim množinu hodnôt poľa. Zatiaľ čo pri indexoch skutočnosť, že všetky indexy nadobúdajú hodnoty iba <u>jedného typu</u>, považuje sa za samozrejmosť, pri hodnotách prvkov poľa to také samozrejmé nie je. Ak prvky poľa nie sú rovnakého typu, hovoríme o <u>heterogénnych poliach</u>. Heterogénnym poľom je napr. záznam v jazyku Pascal.

1.2.8 T a b u ľ k a

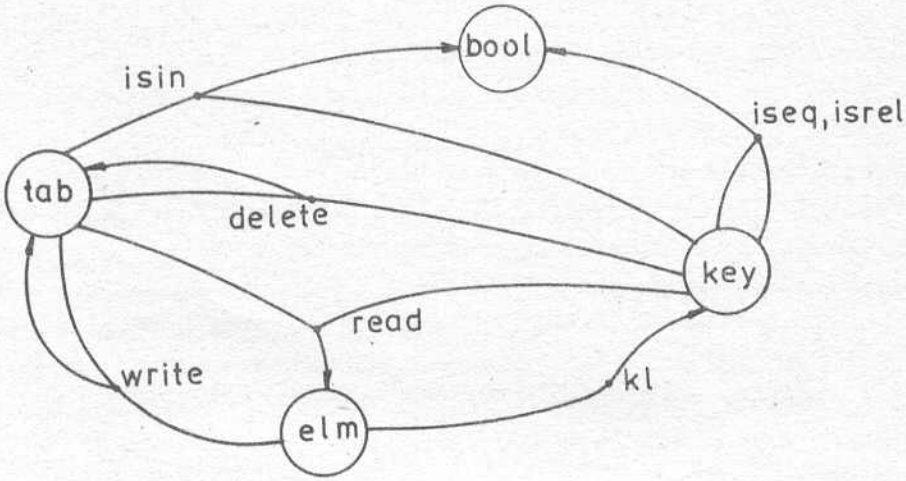
Tabuľka je množina prvkov identifikovateľných pomocou tzv. kľúča. Kľúč je súčasťou prvku a vyhradenou časťou prvku. V porovnaní s poľom je tabuľka určitým zovšeobecnením poľa. Pri poli je s prvkom spojený index, pri tabuľke kľúč. Na rozdiel od poľa je kľúč súčasťou prvku a môže to byť ľubovoľná časť prvku, ktorá nás z hľadiska aplikácie zaujíma. S prvkom môže byť spojených aj viac kľúčov.

<u>Špecifikácia:</u>

druhy : tab
elm
key
bool

operácie:	read : tab,key ⟶ elm		čítanie prvku určeného daným kľúčom
	write : tab,elm ⟶ tab		zápis daného prvku do tabuľky
	isin : tab,key ⟶ bool		test, či je prvok s daným kľúčom v tabuľke
	delete: tab,key ⟶ tab		zrušenie prvku v tabuľke

iseq : key,key ⟶ bool		test, či sa dva kľúče rovnajú
isrel: key,key ⟶ bool		test, či sú dva kľúče v relácii
kl : elm ⟶ key		určenie kľúča daného prvku



Obr. 1.12
Signatúra typu údajov "tabuľka"

1.2.9 G r a f

Pod grafom G rozumieme dvojicu (V,H), teda G=(V,H), kde V je množina vrcholov, H ⊆ V x V je množina hrán. Hovoríme, že dvojica vrcholov z H, ktoré definujú príslušnú hranu, s touto hranou incidujú. Ak dvojice z H považujeme za usporiadané, hovoríme o orientovanom grafe, ak ich považujeme za neusporiadané, hovoríme o neorientovanom grafe.

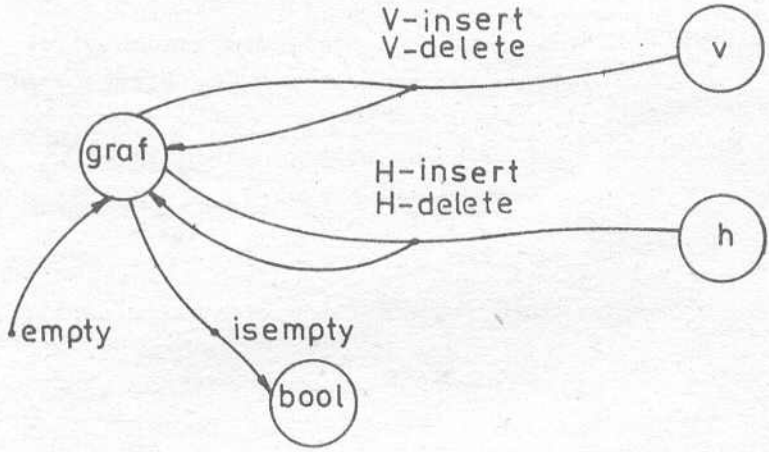
V aplikáciach sa najčastejšie stretávame s ohodnotenými grafmi, keď s množinou vrcholov, hrán alebo obidvoma množinami je prostredníctvom určitého zobrazenia spojená iná množina. Hovoríme potom o vrcholovo, hranovo alebo vrcholovo i hranovo ohodnotenom grafe.

Vzhľadom na rozmanitosť aplikácií grafov je aj špecifikácia grafu, ako údajového typu, závislá od prostredia, v ktorom sa graf bude používať. Z tohto dôvodu uvedieme v signatúre iba základné operácie a poukážeme na ďalšie možnosti.

Špecifikácia:

druhy: graf
bool
v
h

operácie:	empty :	⟶ graf	vytvorenie grafu
	V-insert:	graf,v ⟶ graf	pridanie vrcholu do grafu
	H-insert:	graf,h ⟶ graf	pridanie hrany do grafu
	V-delete:	graf,v ⟶ graf	zrušenie vrchola v grafe
	H-delete:	graf,h ⟶ graf	zrušenie hrany v grafe
	isempty :	graf ⟶ bool	test, či je graf prázdny



Obr. 1.13
Signatúra typu údajov "graf"

Ako ďalšie operácie možno uvažovať napr. prienik grafov, zjednotenie grafov, výber špeciálneho grafu, ako je kostra, cesta, cyklus, ap.

1.2.10 S t r o m

V praktických aplikáciach grafu sa najčastejšie používa jeho špeciálny tvar - strom. Neformálne možno strom definovať ako orientovaný graf bez cyklov, v ktorom existuje jeden vyhradený vrchol - koreň stromu, do ktorého nevstupuje žiadna hrana a do každého iného vrchola vstupuje práve jedna hrana. Vrcholy, z ktorých nevystupuje žiadna hrana sa nazývajú koncové vrcholy alebo listy.

Ak (u,v) je hrana stromu, potom vrchol u nazývame predchodca vrchola v a vrchol v nazývame nasledovník vrchola u. Ak niektorý nasledovník má ďalej nasledovníkov, nazývame ich vzhľadom na predchodcu nepriamymi nasledovníkmi. Vrcholy, ktoré majú nasledovníkov nazývame vnútornými.

Vrchol stromu, spolu so všetkými svojimi priamymi i nepriamymi nasledovníkmi, nazývame podstrom daného stromu. Množina stromov sa nazýva les.

Ku každému vrcholu stromu existuje cesta z jeho koreňa. Počet hrán na ceste z koreňa stromu k danému vrcholu nazývame hĺbkou daného vrchola. Dĺžka najdlhšej cesty z vrchola u k ľubovoľnému koncovému vrcholu je jeho výška. Výška stromu je určená výškou jeho koreňa.

Strom, v ktorom priami nasledovníci každého vrchola tvoria usporiadanú množinu, nazývame usporiadaným stromom.

V mnohých aplikáciach je výhodné zúžiť ešte aj pojem stromu a používať špeciálne typy stromov. Najpoužívanejší je binárny strom. Binárny strom je taký, v ktorom má každý vrchol najviac dvoch nasledovníkov s označením ľavý a pravý. V prípade, že vrchol má iba jedného nasledovníka, musí byť určené, či je to ľavý alebo pravý nasledovník.

Problémy so špecifikáciou stromu ako štruktúry údajov sú analogické ako s grafom. Zostaneme preto pri signatúre uvedenej pre grafy.

1.2.11 M n o ž i n a

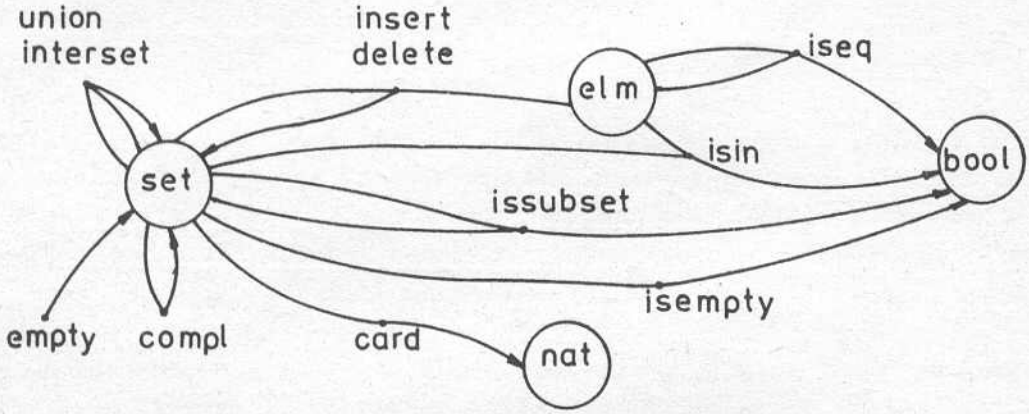
Typ údajov množina je taký, ktorého prvkami sú konečné množiny a operáciami sú množinové operácie. Možno ju považovať za základný typ údajov, ktorý používame na špecifikáciu iných typov údajov. Uvidíme, že niektoré typy údajov možno považovať za špeciálny prípad typu údajov množina.

Pri špecifikácii typu údajov množina budeme predpokladať, že prvky množiny sú konečné množiny rovnakého typu.

Špecifikácia neusporiadanej množiny:

druhy	: set		
	elm		
	nat		
	bool		
operácie:	empty	: ⟶ set	vytvorenie prázdnej množiny
	insert	: set,elm ⟶ set	pridanie prvku do množiny
	delete	: set,elm ⟶ set	zrušenie prvku v množine
	union	: set,set ⟶ set	zjednotenie množín
	intersect	: set,set ⟶ set	prienik množín
	compl	: set ⟶ set	doplnok k danej množine samozrejme, že sa používa vtedy, ak je univerzum konečné
	card	: set ⟶ nat	kardinalita množiny
	isin	: set,elm ⟶ bool	test, či sa v množine nachádza daný prvok

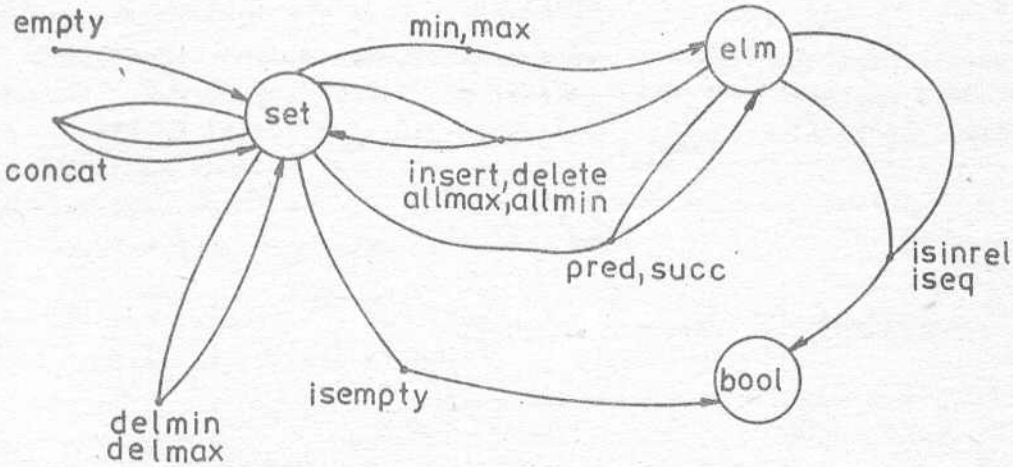
isempty :	set ⟶ bool	test, či je množina prázdna
iseq :	elm,elm ⟶ bool	test, či sa dva prvky rovnajú
issubset:	set,set ⟶ bool	test, či je jedna množina podmnožinou inej množiny



Obr. 1.14
Signatúra typu údajov "neusporiadaná množina"

Špecifikácia usporiadanej množiny:

druhy :	set		
	elm		
	bool		
operácie:	pred :	set,elm ⟶ elm	Predchodca daného prvku - najväčší prvok množiny nie väčší ako daný prvok
	succ :	set,elm ⟶ elm	nasledovník daného prvku - najmenší prvok množiny nie menší ako daný prvok
	min :	set ⟶ elm	minimálny prvok množiny
	max :	set ⟶ elm	maximálny prvok množiny
	delmin :	set ⟶ set	vylúčenie minimálneho prvku z množiny
	delmax :	set ⟶ set	vylúčenie maximálneho prvku z množiny
	allmin :	set,elm ⟶ set	vylúčenie všetkých prvkov z množiny, ktoré sú menšie ako daný prvok
	allmax :	set,elm ⟶ set	vylúčenie všetkých prvkov z množiny, ktoré sú väčšie ako daný prvok
	insert :	set,elm ⟶ set	pridanie prvku
	delete :	set,elm ⟶ set	vylúčenie daného prvku
	isinrel:	elm,elm ⟶ bool	test, či sú dva prvky v relácii ($\leq$)
	iseq :	elm,elm ⟶ bool	test, či sa dva prvky rovnajú
	isempty:	set ⟶ bool	test, či je množina prázdna
	concat :	set,set ⟶ set	zjednotenie, ale predpokladá sa, že všetky prvky prvej množiny sú menšie ako všetky prvky druhej množiny
	empty :	⟶ set	vytvorenie prázdnej množiny



Obr. 1.15
Signatúra typu údajov "usporiadaná množina"

Ďalším prípadom typu údajov množina je "systém dizjunktných množín". Používame ho vtedy, keď potrebujeme pracovať s viacerými navzájom dizjuntnými množinami súčasne. Možno ho definovať cez dvojicu množín sysd a name, kde sysd je množina systémov po dvoch dizjunktných množín s prvkami z daného typu a name je množina mien alebo názvov množín, pričom každému menu z name je v každom systéme zo sysd priradená nejaká množina. Keďže každá neprázdna množina je priradená jedinému menu, väčšine mien je priradená prázdna množina.

Okrem operácii uvedených pre typ údajov množina používajú sa ešte operácie:

find : sysd,elm	⟶ name	určenie mena množiny, do ktorej patrí daný prvok
split: sysd,elm,name,name,name	⟶ sysd	rozdelenie množiny udanej prvým menom na množinu prvkov menších ako daný prvok - označí sa druhým menom a množinu zvyšných prvkov - označí sa tretím menom.

Bohatosť operácii poskytuje veľa možnosti pre použitie typu údajov množina. Na druhej strane existuje celý rad aplikácii, kde vystačíme s podstatne menším množstvom operácií. Takéto špeciálne prípady typu údajov množina sa už určitým spôsobom štandardizovali a dostali aj svoje pomenovanie. Patria sem tieto typy údajov:

meno:	operácie:
slovník	empty,insert,delete,isin
prioritný front	empty,insert,delete,min
halda	empty,insert,delmin
zlučovateľná halda	empty,insert,delete,min,union,find

meno:	operácie:
zreťaziteľný front	empty,insert,delete,find,concat,split
faktorová množina	union,find
univerzálny front	všetky operácie na jednej množine

1.3 ABSTRAKTNÉ TYPY ÚDAJOV

1.3.1 Špecifikácia

Praktické použitie typov údajov si vyžaduje ich presnú špecifikáciu. Pritom by bolo výhodné, aby špecifikácia nebola závislá od implementácie. Práve v tomto smere sa rozvíja teória abstraktných typov údajov. Najďalej v rozpracovaní tejto teórie prišla skupina autorov Gougen, Thatcher, Wagner a Wright, označovaná ako skupina ADJ. Rozpracovala dôslednú matematickú teóriu abstraktných typov údajov, založenú na viacdruhovej algebre. Tejto teórii sa budeme teraz venovať podrobnejšie.

Pod algebrou rozumieme množinu, ktorú nazývame nosič algebry, spolu s indexovanou triedou operácií (funkcií) definovaných na kartézskych súčinoch tejto množiny. Potom viacdruhová algebra pozostáva z indexovaného súboru množín, ktoré nazývame nosiče a indexovanej triedy operácií definovaných na kartézskych súčinoch týchto množín. Indexovú množinu nosičov nazývame množina druhov.

Ak napríklad zoberieme za indexovú množinu {int,real,bool}, algebra A bude mať nosiče $A_{int}, A_{real}, A_{bool}$ a ako operácie by sme mohli uvažovať:

$$exp : A_{real} \times A_{int} \longrightarrow A_{real}$$
$$kond: A_{bool} \times A_{real} \times A_{real} \longrightarrow A_{real}$$

kde operácie exp je exponenciálna funkcia - umocnenie reálneho základu na celočíselný exponent a operácia kond je podmienený výraz - podľa hodnoty booleovského výrazu je hodnota celého výrazu prvý alebo druhý argument.

Poznámka: V praxi sa často stretávame so zápismi, v ktorých druhy nepoužívame ako indexy nosičov, ale priamo ako nosiče a ako symbol kartézskeho súčinu sa používa čiarka.

K abstraktným typom údajov sa dostaneme cez formálnejšie definície.

meno:	operácie:
zreťaziteľný front	empty,insert,delete,find,concat,split
faktorová množina	union,find
univerzálny front	všetky operácie na jednej množine

1.3 ABSTRAKTNÉ TYPY ÚDAJOV

1.3.1 Špecifikácia

Praktické použitie typov údajov si vyžaduje ich presnú špecifikáciu. Pritom by bolo výhodné, aby špecifikácia nebola závislá od implementácie. Práve v tomto smere sa rozvíja teória abstraktných typov údajov. Najďalej v rozpracovaní tejto teórie prišla skupina autorov Gougen, Thatcher, Wagner a Wright, označovaná ako skupina ADJ. Rozpracovala dôslednú matematickú teóriu abstraktných typov údajov, založenú na viacdruhovej algebre. Tejto teórii sa budeme teraz venovať podrobnejšie.

Pod algebrou rozumieme množinu, ktorú nazývame nosič algebry, spolu s indexovanou triedou operácií (funkcií) definovaných na kartézskych súčinoch tejto množiny. Potom viacdruhová algebra pozostáva z indexovaného súboru množín, ktoré nazývame nosiče a indexovanej triedy operácií definovaných na kartézskych súčinoch týchto množín. Indexovú množinu nosičov nazývame množina druhov.

Ak napríklad zoberieme za indexovú množinu {int,real,bool}, algebra A bude mať nosiče $A_{int}, A_{real}, A_{bool}$ a ako operácie by sme mohli uvažovať:

$$exp : A_{real} \times A_{int} \longrightarrow A_{real}$$
$$kond: A_{bool} \times A_{real} \times A_{real} \longrightarrow A_{real}$$

kde operácie exp je exponenciálna funkcia - umocnenie reálneho základu na celočíselný exponent a operácia kond je podmienený výraz - podľa hodnoty booleovského výrazu je hodnota celého výrazu prvý alebo druhý argument.

Poznámka: V praxi sa často stretávame so zápismi, v ktorých druhy nepoužívame ako indexy nosičov, ale priamo ako nosiče a ako symbol kartézskeho súčinu sa používa čiarka.

K abstraktným typom údajov sa dostaneme cez formálnejšie definície.

Definícia 3.1

Nech $D=\{d_1,\dots,d_n\}$ je množina, ktorej prvky nazývame druhy. Potom signatúra Σ je trieda $\langle\Sigma_{w,d}\rangle$ dizjunktných množín indexovaných kartézskym súčinom $D^* \times D$, kde D^* je množina všetkých reťazcov alebo slov, vytvorených na D, vrátane prázdneho reťazca. $\Sigma_{w,d}$ je množina symbolov operátorov $\Omega=\{\omega_1, \omega_2, \dots, \omega_m\}$ typu $\langle w,d\rangle$, árnosti w, druhu d a rádu určeného dĺžkou reťazca w.

Signatúru Σ budeme skrátene zapisovať $\Sigma=\langle D,\Omega\rangle$, pričom árnosť a druh budú dané implicitne zápisom operácií.

Definícia 3.2

Σ-algebra A je trieda množín $\langle A_d\rangle_{d\in D}$, ktoré nazývame nosiče algebry A, pričom A_d je nosič druhu $d\in D$, spolu s triedou operácií, v ktorej je pre každé $\langle w,d\rangle \in D^* \times D$ a $\omega\in\Omega$ operácia ω_A, typu $\langle w,d\rangle$, teda ω_A: $A_{d_1} \times A_{d_2} \times \dots \times A_{d_n} \longrightarrow A_d$, $d_i \in D$ pre $i=1,2,\dots,n$, $w=d_1\dots d_n$. Ak w je prázdny reťazec, kde $w=e$, potom operácia $\omega_A(e,d)$ určuje konštantu druhov d, t.j. $\omega_A \in A_d$.

Pri pevnom D a meniacom sa Σ hovoríme o D-druhovej algebre, ak sa mení aj D, hovoríme o viac (mnoho) druhovej algebre. K danej signatúre existuje nekonečne veľa Σ-algebier. Dôležitý vzťah medzi Σ-algebrami možno definovať pomocou homomorfizmu.

Definícia 3.3

Ak A a B sú Σ-algebry, potom Σ-homomorfizmus h: $A \rightarrow B$ je trieda funkcii (zobrazení) $\langle h_d: A_d \longrightarrow B_d\rangle_{d\in D}$, ktoré zachovávajú operácie, t.j.

a/ ak $\omega \in \Sigma_{e,d}$, potom $h_d(\omega_A) = \omega_B$

b/ ak $\omega \in \Sigma_{d_1\dots d_n,d}$ a $\langle a_1,\dots,a_n\rangle \in A_{d_1} \times \dots \times A_{d_n}$, potom

$$h_d(\omega_A(a_1,\dots,a_n)) = \omega_B(h_{d_1}(a_1),\dots h_{d_n}(a_n))$$

Ak je zobrazenie bijektívne, hovoríme o izomorfizme.

Každá signatúra určuje triedu všetkých Σ-algebier. Dá sa ľahko overiť, že táto trieda, spolu so všetkými Σ-homomorfizmami medzi nimi, tvorí kategóriu, t.j. triedu, kde platí kompozícia morfizmov, ktorá je asociatívna a existuje v nej izomorfizmus, ktorý je vzhľadom na kompozíciu morfizmov identitou. Kategóriu Σ-algebier budeme označovať K_Σ.

Z hľadiska abstraktných typov údajov má v kategórii K_Σ osobitné miesto jedna Σ-algebra, resp. trieda Σ-algebier, navzájom izomorfných, ktorá charakterizuje celú kategóriu K_Σ. Táto Σ-algebra sa nazýva iniciálna.

Definícia 3.4

Σ-algebra A je v kategórii K_Σ iniciálna, ak ku každej Σ-algebre B existuje jediný homomorfizmus $h: A \longrightarrow B$.

Definícia 3.5

Triedu izomorfných iniciálnych algebier v kategórii Σ-algebier budeme nazývať abstraktným typom údajov.

Tým, že uvažujeme triedu izomorfných algebier, dosiahneme nezávislosť typu údajov od implementácie, teda jeho abstraktnosť. Treba však ukázať, že v kategórii K_Σ iniciálna algebra vždy existuje a pojem iniciálnosti je až na izomorfizmus jednoznačný.

Definícia 3.6

Nech $\Sigma = \langle D, \Omega \rangle$ je signatúra, $\langle T_{\Sigma,d} \rangle_{d \in D}$ je najmenšia trieda množín obsiahnutých v $(\Omega \cup \{(,)\})^*$ spĺňajúcich tieto podmienky:

1. $\Sigma_{e,d} \subseteq T_{\Sigma,d}$
2. ak $\omega \in \Sigma_{w,d}$ $\quad w = d_1 \ldots d_n$, $\quad n > 0$ a $t_i \in T_{\Sigma,d_i}$, potom

$$\omega(t_1, t_2, \ldots, t_n) \in T_{\Sigma,d}$$

Potom na triede $\langle T_{\Sigma,d} \rangle$ možno definovať Σ-algebru T_Σ definovaním operácií

1. pre $\omega \in \Sigma_{e,d}$ $\quad \omega_{T_\Sigma} = \omega \in T_{\Sigma,d}$
2. pre $\omega \in \Sigma_{w,d}$ $\quad w = d_1 \ldots d_n$ a $t_i \in T_{\Sigma,d_i}$

$$\omega_{T_\Sigma}(t_1, \ldots, t_n) = \omega(t_1, \ldots, t_n) \in T_{\Sigma,d}$$

Σ-algebra T_Σ vytvorená na základe definície 3.6 je vlastne algebra slov vytvorených zo symbolov operácií a zátvoriek (,) vrátane triviálne reprezentovaných operácií. Nosičom druhu $d \in D$ je množina $T_{\Sigma,d}$.

Definícia 3.6 udávajúca spôsob vytvárania Σ-algebry T_Σ je základom riešenia problému existencie iniciálnej algebry v kategórii K_Σ.

Veta 3.1

Nech Σ je ľubovoľná signatúra. Potom Σ-algebra T_Σ je iniciálnou algebrou v kategórii všetkých Σ-algebier K_Σ.

Príklad 3.1

Uvažujme nasledujúcu signatúru $\Sigma = (D, \Omega)$:

$$D = \{nat\}$$
$$\Omega = \{nul, succ\}$$
$$nul: \longrightarrow nat$$
$$succ: nat \longrightarrow nat$$

potom algebra T_Σ obsahuje slová:

$$nul, succ(nul), succ(succ(nul)), \ldots$$

Vidíme, že T_Σ obsahuje všetky slová, ktoré dostaneme aplikáciou operácie succ na nul, resp. na predchádzajúcu aplikáciu succ. Túto množinu slov možno rekurzívne definovať takto (definícia 3.6):

1. nul patrí do T_Σ,
2. ak t patrí do T_Σ, potom aj succ(t) patrí do T_Σ,
3. do T_Σ patria iba slová vytvorené na základe 1. a 2.

Ak zavedieme označenie

$$succ^0(nul) = nul$$
$$succ^{i+1}(nul) = succ(succ^i(nul)),$$

potom do T_Σ patria všetky slová $succ^i(nul)$ pre $i \geq 0$.

Z príkladu 3.1 vidíme, že ak budeme operáciu nul reprezentovať číslom 0 a $succ^i(nul)$ prirodzeným číslom i, potom T_Σ je množina prirodzených čísiel. Nie je to však jediná reprezentácia. Ak budeme nul reprezentovať 0 a operáciu succ(nul) = 00, potom dostaneme Σ-algebru, v ktorej T_Σ reprezentuje množinu postupností núl. Medzi Σ-algebry so signatúrou Σ patrí aj jednoprvková množina $\{nul\}$, ak succ(nul) = nul.

Každá signatúra určuje triedu všetkých Σ-algebier. Umožňuje nám zapísať syntax ľubovoľného údajového typu, nie je v nej však zahrnutá sémantika operácií. Zo všetkých Σ-algebier nás budú zaujímať iba také, ktoré majú určité vlastnosti. Z kategórie K_Σ vyberieme podkategóriu algebier, v ktorej existuje iniciálny objekt, a ktoré spĺňajú určité podmienky reprezentované nejakou sústavou axióm. V prístupe ADJ majú všetky axiómy tvar rovnosti dvoch výrazov, vytvorených z mien operácií, voľných premenných a zátvoriek, pričom každá voľná premenná má explicitne priradený druh z D (voľné rozšírenie signatúry).

Sústava axióm E definuje v každej Σ-algebre reláciu ϱ, reláciu prvkov "vyhovujúcich" ľavej a pravej strane axióm. Relácia ϱ generuje v Σ-algebre Σ-kongruenciu $\equiv_E$, t.j. ekvivalenciu, ktorá sa zachováva pri použití operácií zo Σ. Σ-kongruencia $\equiv_E$ definuje faktorovú Σ-algebru $A/\equiv_E$ pôvodnej algebry A, kde prvky $A/\equiv_E$ sú triedy ekvivalencií $\equiv_E$ Σ-algebry A. Analogicky možno vytvoriť faktorovú algebru $T_\Sigma/\equiv_E$.

Veta 3.2

Faktorová algebra $T_\Sigma/\equiv_E$ je iniciálnou algebrou v kategórii $K_{\Sigma,E}$ všetkých Σ-algebier vyhovujúcich sústave axióm E.

Ak k syntaxi typu údajov vyjadrenej danou signatúrou, pridáme množinu axióm určujúcich sémantiku operácií, dostávame jeho špecifikáciu.

Definícia 3.7

Špecifikácia typu údajov je dvojica $\varphi = (\Sigma, E)$, kde $\Sigma = (D, \Omega)$ je signatúra daného typu údajov a E je množina axióm.

Σ-algebra A spĺňa špecifikáciu φ, ak každá z rovností z E, chápaná ako formula prediktátového počtu, je v A pravdivá. Ak Σ-algebra A spĺňa špecifikáciu φ, nazýva sa φ-algebra.

Príklad 3.2

Uvažujme signatúru z príkladu 3.1. Pridaním množiny axióm E, ktorá je prázdna, dostaneme špecifikáciu. Túto špecifikáciu spĺňa Σ-algebra s nosičom $T_{\Sigma,nat}$ obsahujúcim prvky nul, succ(nul), succ(succ(nul)),... Vidieť, že Σ-algebra T_Σ je izomorfná s algebrou prirodzených čísiel.

Príklad 3.3

Uvažujme signatúru $\Sigma = (D, \Omega)$

$$D = \{int\}$$
$$\Omega = \{nul, succ, pred\}$$
$$nul : \longrightarrow int$$
$$succ: int \longrightarrow int$$
$$pred: int \longrightarrow int$$

Špecifikáciu φ dostaneme pridaním množiny axióm E

1. $succ(pred(x)) = x$
2. $pred(succ(x)) = x$

Pre $d \in D$ sa nosič $(T_\Sigma/\equiv_E)_d$ skladá z tried ekvivalencie $\equiv_{E,d}$ na $T_{\Sigma,d}$, teda nosič $(T_\Sigma/\equiv_E)_{int}$ pozostáva z prvkov

$\{nul, succ(pred(nul)), pred(succ(nul)), succ(pred(succ(pred(nul)))), \ldots\}$
$\{succ(nul), succ(succ(pred(nul))), succ(pred(succ(nul))), \ldots\}$
$\{pred(nul, pred(succ(pred(nul))), pred(pred(succ(nul))), \ldots\}$

$\vdots$

Ak si pre jednotlivé triedy ekvivalencie zvolíme vhodné reprezentanty, možno nosič $(T_\Sigma/\equiv_E)_{int}$ písať v tvare

$$\ldots pred(pred(nul)), pred(nul), nul, succ(nul), succ(succ(nul)), \ldots$$

alebo skrátene

$$pred^n(nul), nul, succ^n(nul)$$

z čoho je už zrejmý izomorfizmus s algebrou celých čísiel.

Operácie typov údajov možno rozdeliť na:

a. konštruktory - ktorých aplikáciou vzniká nový prvok alebo nový stav niektorého z nosičov,

b. selektory - ktorých aplikáciou dostávame hodnoty prvkov nosičov reprezentujúcich druh výsledku, pričom hodnota ani stav prvku sa nemení,

c. predikáty - používame na zisťovanie vlastností. Výsledkom je hodnota true alebo false. Hodnota stavu nosičov zostáva nezmenená.

Už v úvode sme hovorili o rôznych spôsoboch špecifikácie nových typov údajov. Z hľadiska tvorby programov a abstrakcie sú zaujímavé tzv. parametrické typy údajov. Ide o špecifikáciu, v ktorej namiesto daného typu vystupuje parameter. Napríklad možno špecifikovať typ údajov "pole celých čísiel",

"pole prirodzených čísiel" atď. Vidíme, že ako súčasť typu údajov pole vystupujú typy "celé číslo" a "prirodzené číslo", a pochopiteľne, že tam môžu vystupovať aj ďalšie typy. Možno preto špecifikovať typ údajov "pole typu T" a za T potom dosadiť príslušný typ.

1.3.2 I m p l e m e n t á c i a

vyjadrenie operácií v danom prostredí

Implementáciou typu údajov intuitívne rozumieme spôsob reprezentácie údajových štruktúr a spôsob realizácie operácii daného typu údajov na takto reprezentovaných štruktúrach. Použitie špecifikovaných abstraktných typov údajov si vyžaduje implementovať ich v "známom prostredí", v konečnom dôsledku v prostredí daného počítača. Keďže aj tento počítač alebo jeho model možno opísať ako algebru, resp. "abstraktný" typ údajov, hovoríme o implementácii jedného abstraktného typu údajov iným abstraktným typom údajov.

Analogicky ako pri prekladačoch, kľúčovou otázkou implementácie je jej korektnosť. Nutnou podmienkou korektnosti implementácie je korektnosť samotnej špecifikácie typov údajov. Jeden prístup ku korektnosti vychádza z predpokladu, že máme definovanú vhodnú Σ-algebru A, ktorá je modelom uvažovaného abstraktného typu údajov špecifikovaného špecifikáciou $\varphi = (\Sigma, E)$. Špecifikácia φ určuje jednoznačne abstraktný typ údajov ako triedu algebier izomorfných s iniciálnou algebrou T_Σ a jej korektnosť vzhľadom na Σ-algebru A možno definovať nasledujúcim spôsobom:

Veta 3.3

Špecifikácia $\varphi = (\Sigma, E)$ je korektná vzhľadom na Σ-algebru A práve vtedy, keď $A \cong T_d$.

Takto chápaná korektnosť predpokladá vopred definovaný model, Σ-algebru A, ale nekladie žiadne požiadavky na spôsob jej špecifikácie. Možno ju špecifikovať pomocou iniciálnej algebry, ale sémantiku možno definovať inou sústavou axióm ako tie, ktoré sú súčasťou špecifikácie daného typu údajov. Dokonca nemusí byť definovaná axiomaticky.

Nie vždy máme k danej špecifikácii matematický model. V takomto prípade možno ku korektnosti špecifikácie pristupovať tak, že zoberieme niektoré základné špecifikácie, ktorých korektnosť sa už preukázala. Z nich vybudujeme našu špecifikáciu. Pôvodné typy údajov zostanú v novom type bez zmeny, a teda v definovanom type zostanú zachované všetky vlastnosti pôvodného typu. V tomto zmysle overujeme korektnosť špecifikácie vzhľadom na inú špecifikáciu.

Teraz možno prejsť k formálnejšiemu a presnejšiemu definovaniu pojmu implementácie. Existuje niekoľko prístupov k definovaniu implementácie. Prvý vychádza z našej intuitívnej predstavy. Implementáciu typu údajov možno chápať ako ľubovoľnú implementáciu jeho operácií, ktorá je modelom pôvodného typu, spolu s funkciou f, ktorá priraďuje prvkom v interpretujúcej algebre prvky interpretovanej algebry. Podmienkou korektnosti takto definovanej implementácie je, aby zobrazenie f zachovávalo operácie pôvodného typu údajov, alebo inými slovami, musí komutovať diagram (Obr. 1.16).

$$\begin{tikzcd}
x \arrow[r, "\omega"] & \omega(x) \\
x_{ref} \arrow[u, "f"] \arrow[r, "\omega_{impl}"'] & \omega_{impl}(x_{rep}) \arrow[u, "f"']
\end{tikzcd}$$

Obr. 1.16
Podmienka korektnosti implementácie

Druhý prístup je založený na tzv. vnorení dvoch špecifikácií. Ak $\varphi_1 = (D_1, \Omega_1, E_1)$ a $\varphi_2 = (D_2, \Omega_2, E_2)$ sú špecifikácie, potom ich vnorenie je dvojica zobrazení $f = (g,h)$, kde

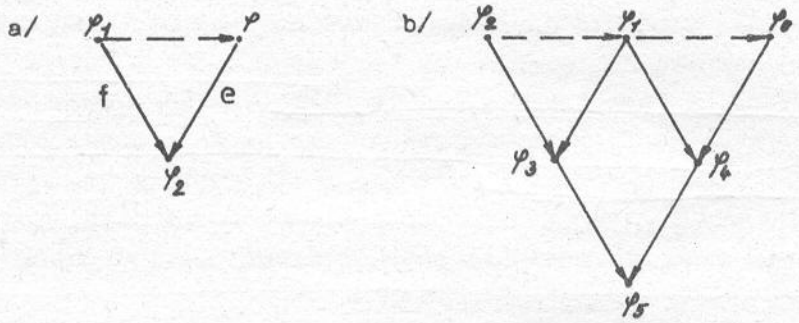
$g: \Omega_1 \longrightarrow \Omega_2,$
$h: D_1 \longrightarrow D_2,$

ktoré sú injektívne, a pre ktoré platí:

1. typy argumentov operácie $\omega_1 \in \Omega_1$ zobrazené h sú v zhode s typmi argumentov operácie $\omega_2 = g(\omega_1) \in \Omega_2$,
2. iniciálna algebra $T = (D_2, \Omega_2, E_2)$ vyhovuje axiómam E_1 transformovaným zobrazením g.

Implementácia špecifikácie φ v špecifikácii φ_1 je potom trojica (φ_2, e, f), kde φ_2 je nová špecifikácia
e je vnorenie $e: \varphi \longrightarrow \varphi_2$,
f je vnorenie $f: \varphi_1 \longrightarrow \varphi_2$.

Implementáciu vnorením možno zobraziť ako to dokumentuje obr. 1.17a/. Špecifikácia φ_2 hrá úlohu špecifikácie φ_1 obohatenej o nové operácie, pomocou ktorých sú potom implementované operácie špecifikácie φ. Špecifikáciu φ_1 môže hrať špecifikácia typov údajov programovacieho jazyka, ktorý používame na implementáciu a obohatili sme ho o nové operácie - procedúry. Pomocou nich potom vyjadríme operácie implementovaného typu údajov. Implementáciu vnorením možno dokonca skladať, ako to znázorňuje obr. 1.17 b/.



Obr. 1.17
Implementácia vnorením

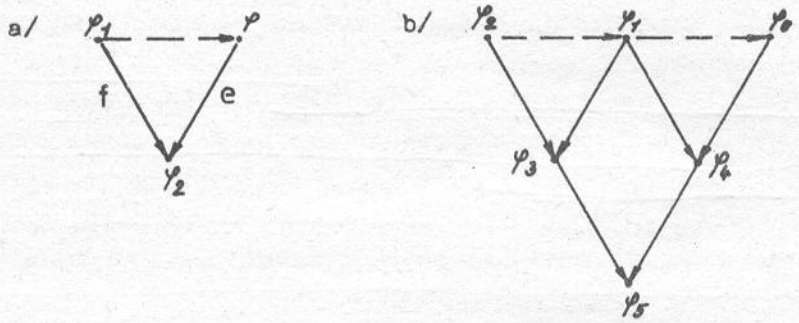
Ďalší prístup k definovaniu implementácie je založený na rozšírení zobrazenia algebier tak, že algebre priraďujeme algebru a morfizmu algebier morfizmus algebier. Pod implementáciou špecifikácie typu údajov $\varphi = (\Sigma, E)$ rozumieme trojicu $I = (A, F, f)$, kde A je implementujúca algebra $A = (\Sigma', E')$, F je funktor z kategórie Alg Σ', do kategórie Alg Σ, f je surjektívny morfizmus $F(A) \longrightarrow T: (\Sigma, E)$.

Implementácia sa opisuje morfizmom f. Operácie potrebné k implementácii sú určené algebrou A. Funktor F nám umožňuje tieto operácie podľa potreby rozšíriť alebo zúžiť. Aj v tomto prípade možno implementácie skladať.

Ako sme v úvode povedali, kľúčovou otázkou zostáva korektnosť implementácie. Dokázať korektnosť implementácie znamená dokázať, že implementácia vyhovuje definícii. Algebraická špecifikácia typov údajov umožňuje dokazovať korektnosť exaktne. Operačná špecifikácia je súčasne implementácia. Overovať jej korektnosť znamená overovať korektnosť algoritmu vlastnej operačnej špecifikácie.

1.4 IMPLEMENTÁCIA VYBRATÝCH TYPOV ÚDAJOV

V tejto kapitole si rozoberieme praktickú realizáciu implementácie vybratých typov údajov, ktoré sme špecifikovali v kapitole 1.2. Ako sme už spomenuli, použitie typov údajov si vyžaduje ich implementáciu v prostredí počítača. Napriek možnosti špecifikácie počítačových prostredí ako typu údajov, v ktorom množinu operácií tvorí operačný kód a množina druhov je určená typmi údajov zobraziteľnými v počítači, nepoužívame ho ako implementujúci typ údajov. Nevyhnutnosť implementácie v počítačovom prostredí "obchá-



Obr. 1.17
Implementácia vnorením

Ďalší prístup k definovaniu implementácie je založený na rozšírení zobrazenia algebier tak, že algebre priraďujeme algebru a morfizmu algebier morfizmus algebier. Pod implementáciou špecifikácie typu údajov $\varphi = (\Sigma, E)$ rozumieme trojicu $I = (A, F, f)$, kde A je implementujúca algebra $A = (\Sigma', E')$, F je funktor z kategórie Alg Σ', do kategórie Alg Σ, f je surjektívny morfizmus $F(A) \longrightarrow T: (\Sigma, E)$.

Implementácia sa opisuje morfizmom f. Operácie potrebné k implementácii sú určené algebrou A. Funktor F nám umožňuje tieto operácie podľa potreby rozšíriť alebo zúžiť. Aj v tomto prípade možno implementácie skladať.

Ako sme v úvode povedali, kľúčovou otázkou zostáva korektnosť implementácie. Dokázať korektnosť implementácie znamená dokázať, že implementácia vyhovuje definícii. Algebraická špecifikácia typov údajov umožňuje dokazovať korektnosť exaktne. Operačná špecifikácia je súčasne implementácia. Overovať jej korektnosť znamená overovať korektnosť algoritmu vlastnej operačnej špecifikácie.

1.4 IMPLEMENTÁCIA VYBRATÝCH TYPOV ÚDAJOV

V tejto kapitole si rozoberieme praktickú realizáciu implementácie vybratých typov údajov, ktoré sme špecifikovali v kapitole 1.2. Ako sme už spomenuli, použitie typov údajov si vyžaduje ich implementáciu v prostredí počítača. Napriek možnosti špecifikácie počítačových prostredí ako typu údajov, v ktorom množinu operácií tvorí operačný kód a množina druhov je určená typmi údajov zobraziteľnými v počítači, nepoužívame ho ako implementujúci typ údajov. Nevyhnutnosť implementácie v počítačovom prostredí "obchá-

dzame" skladaním implementácií. Využívame "implementujúci jazyk", resp. typy údajov prípustné v danom programovacom jazyku, ktorých implementáciu v "skladaní" zabezpečuje prekladač.

Použitie vyššieho programovacieho jazyka však neznamená, že problém implementácie typov údajov máme vyriešený. Ako už vieme, abstraktné typy údajov a ich špecifikácia sú založené na jazyku teórie množín. Spomínanému jazyku sú však vzdialené nielen súčasné počítače, ale aj programovacie jazyky. Okrem toho je škála typov údajov programovacích jazykov oveľa užšia ako škála, ktorú sme si v kapitole 1.2 špecifikovali.

Opustenie počítačového prostredia a využitie skladania implementácií si vyžaduje výber vhodných primitívnych implementujúcich typov údajov. Existuje viac prístupov k voľbe primitívnych implementujúcich typov údajov. V zásade však vychádzame zo skutočnej organizácie pamäti - lineárnej postupnosti pamäťových buniek alebo dynamickým zreťazením pamäťových oblastí. Pri lineárnej postupnosti pamäťových buniek predpokladáme, že všetky bunky sú "rovnaké" a fyzicky susedné. Preto si takýto typ údajov vyžaduje súvislý blok pamäti. V dôsledku toho sa na implementáciu používajú najmä také typy údajov, ktoré sú z hľadiska operácií statické, a teda nemenia počet prvkov štruktúry. Dynamické zreťazenie pamäťových oblastí je spôsob využívania nesúvislých pamäťových oblastí. Zreťazujeme tzv. voľnú pamäť, pričom zreťazované prvky môžu, ale nemusia byť rovnaké a môžu, ale nemusia byť fyzicky susedné. Typu údajov založenom na prvom prístupe hovoríme vektor, na druhom zreťazená voľná pamäť. Z hľadiska údajovej štruktúry má zreťazená voľná pamäť charakter zoznamu.

1.4.1 Implementujúce typy údajov

Vzhľadom na to, že počítačové prostredie je ako implementujúci typ veľmi zložité, využívame prípustnosť skladania implementácii a ako implementujúci typ údajov používame najčastejšie vektor a zreťazenú voľnú pamäť. Pravda, za implementujúce typy údajov možno považovať aj typy údajov, ktoré nám poskytujú vyššie programovacie jazyky.

A. Vektor

Vektor je usporiadaná n-tica prvkov, z ktorých každý je prístupný celým číslom - indexom. V podstate ide o jednorozmerné pole, ktorého signatúru možno použiť aj pre vektor. Niekedy sa signatúra dopĺňa o operáciu usporiadania na množine indexov.

Pri praktickom používaní vektoru si treba uvedomiť, že vektor je definovaný:

- bázou,
- veľkosťou prvku,
- počtom prvkov.

Typ údajov vektor je obsiahnutý priamo v strojovom kóde počítača. Prvky vektora sú pamäťové bunky (slová, bajty) a indexy sú adresy pamäťových buniek. Vektor ako typ údajov pripúšťa aj väčšina vyšších programovacích jazykov. Preto sa implementáciou vektora nebudeme zaoberať.

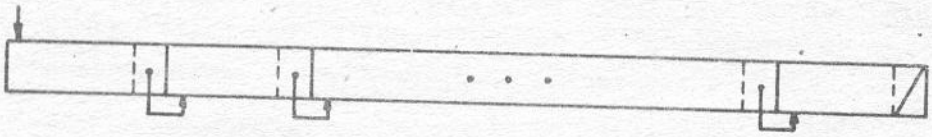
B. Zreťazená voľná pamäť

Zreťazená voľná pamäť je taká organizácia pamäti, v ktorej sú jednotlivé prvky pamäti "spojené" - zreťazené do jedného celku. Jednotlivé prvky pamäti môžu mať rovnakú alebo rôznu dĺžku. V prvom prípade budeme hovoriť o zreťazenej voľnej pamäti rovnakých prvkov, v druhom prípade o zreťazenej voľnej pamäti nerovnakých prvkov. Rozoberieme si ch samostatne.

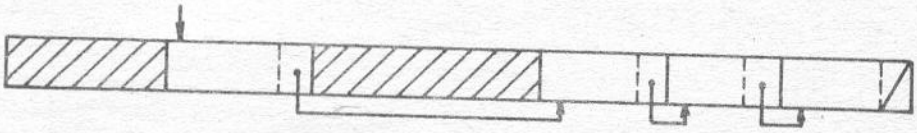
Zreťazenú voľnú pamäť rovnakých prvkov možno špecifikovať pomocou tejto signatúry:

druhy	: zvprp			
	elm			
	bool			
operácie:	init :		→ zvprp	začiatočné reťazenie voľnej pamäti
	dispose:	zvprp,elm	→ zvprp	vrátenie prvku do voľnej pamäti
	new :	zvprp	→ elm	poskytnutie prvku z voľnej pamäti
	isempty:	zvprp	→ bool	test, či je už voľná pamäť vyčerpaná

Implementovať budeme v jazyku Pascal. Ako implementujúci typ údajov použijeme jednorozmerné pole (vektor). Prispôsobíme sa však už požiadavkám jazyka Pascal a predpokladajme, že prvky implementovaných štruktúr budú mať veľkosť zodpovedajúcu d hodnotám typu integer, maximálny počet prvkov je max, d a max sú konštanty. Prvky reťazíme tak, že v prvku vydelíme miesto pre smerník na nasledujúci prvok. Štruktúra prvku bude teda taká, že d-1 celočíselných miest vyhradíme na vlastný obsah prvku implementovanej štruktúry a jedno celočíselné miesto použijeme na smerník. Na začiatku bude zreťazenú voľnú pamäť tvoriť súvislý úsek pamäti a smerníky tak budú ukazovať na prvok, ktorý je fyzicky susedný (Obr. 1.18), avšak po niekoľkých operáciách vrátenia prvku voľnej pamäti sa situácia môže zmeniť, dokumentuje obr. 1.19.



Obr. 1.18
Začiatočné zreťazenie voľnej pamäti rovnakých prvkov

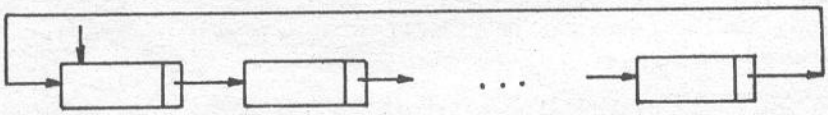


Obr. 1.19
Zreťazenie voľnej pamäti po niekoľkých operáciách vrátenia prvku. Vyšrafovaná časť je "obsadená"

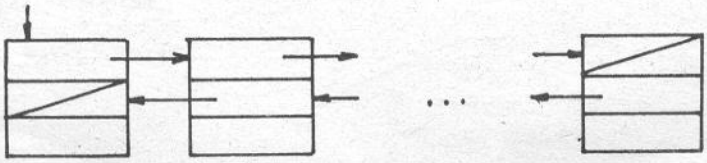
```
const d =  ;
type index  = 1...max;
     vektor = array [index] of integer;
var  i      : index;
     zvprp  : vektor;
procedure init (var zvprp : vektor; i : index);
          /* začiatočné zreťazenie voľnej pamäti */
          var j   : index;
          begin j := i+d;
          repeat
                 zvprp [j-1] := j;
                 j := j+d;
          until  j > max;
                 zvprp [j-d] := 0;
end;

procedure new (var zvprp : vektor; var i,j : index);
          /* poskytnutie voľného prvku zreťazenej voľnej pamäti zvprp.
             Prvý voľný prvok udáva parameter i. Voľný prvok sa poskytne
             cez parameter j */
          begin if isempty (zvprp,i) then error
                                     else begin j := i;
                                                i := zvprp [i+d]
                                          end
          end;
```

```
procedure dispose (var zvprp : vektor; var i,j : index);
          /* vrátenie prvku do zreťazenej voľnej pamäti zvprp. Prvok sa
             vracia cez parameter j. Prvý voľný prvok udáva parameter i */
          begin zvprp [j+d] := i;
                        i := j;
          end

function isempty (zvprp : vektor; i : index) : boolean;
          /* test, či je voľná pamäť vyčerpaná */
          begin isempty := zvprp [i] = 0
          end;
```

Prvky zreťazenej voľnej pamäti môžu vytvárať špeciálne štruktúry podľa spôsobu zreťazenia. Známe sú najmä cyklický zoznam prvkov a obojsmerne zreťazený zoznam prvkov. Graficky ich možno zobraziť týmto spôsobom (Obr. 1.20, obr. 1.21):

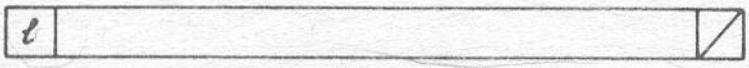


Obr. 1.20
Cyklický zoznam



Obr. 1.21
Obojsmerne zreťazený zoznam

Zreťazená voľná pamäť nerovnakých prvkov poskytuje ako implementujúci typ údajov väčšie možnosti. Pochopiteľne, že nie "zadarmo", pretože treba uvažovať aj veľkosť prvku. Prejaví sa to v špecifikácii, no najmä pri implementácii, kde napr. nestačí mať voľný prvok, ale musí to byť prvok s potrebnou veľkosťou. Ako dôsledok bude potrebné spájať niekoľko prvkov do jedného, ktorý bude spĺňať požiadavky na veľkosť. Štruktúra prvku teraz obsahuje: dĺžku prvku, hodnotovú časť prvku a nakoniec miesto pre smerník, ktorý používame na zreťazenie prvkov. Na začiatku je voľná pamäť zreťazenia tak, že celú voľnú pamäť tvorí jeden prvok (Obr. 1.22). Štruktúra sa stáva zložitejšou až po realizácii prípustných operácií určených signatúrou.



Obr. 1.22
Začiatočné zreťazenie voľnej pamäti

Signatúra zreťazenej voľnej pamäti nerovnakých prvkov zvpnp je:

druhy	: zvpnp			
	elm			
	nat			
	bool			
operácie:	init :		→ zvpnp	začiatočné reťazenie
	new :	zvpnp,nat	→ elm	poskytnutie prvku danej dĺžky
	dispose:	zvpnp,elm,nat	→ zvpnp	vrátenie prvku s danou dĺžkou
	connect:	zvpnp,elm,nat	→ zvpnp	pripojenie prvku s danou dĺžkou k niektorému prvku zreťazenej voľnej pamäti a vytvorenie jedného nového prvku
	collect:	zvpnp	→ zvpnp	spojenie prvkov
	isempty:	zvpnp	→ bool	test, či je voľná pamäť vyčerpaná

Pri implementácii jednotlivých operácií je dôležité vhodne vyriešiť otázku reprezentácie prvkov. Keďže prvky majú rôznu dĺžku, nevystačíme už s prvkami rovnakého typu. Ako implementujúcu štruktúru údajov možno uvažovať pole záznamov s premennými časťami. Každý záznam poľa bude obsahovať: ak je voľný, potom jeho dĺžku d a ukazovateľ next na ďalší voľný záznam - zreťazenie voľných prvkov, inak obsahuje nejaké pole znakov.

Aj v tomto prípade ide o pole, ktorého prvky sú sprístupniteľné pomocou indexov, no premenná dĺžka prvkov sprístupnenie skomplikuje. Možno použiť prístup, v ktorom sa na sprístupnenie prvkov použijú ukazovatele, čím sa dĺžke prvkov "vyhneme". Sprístupnenie prvku pri jeho poskytovaní je sprevádzané prehľadávaním, pretože musíme poskytnúť taký prvok, ktorý má zodpovedajúcu dĺžku. Prehľadávanie možno robiť od začiatku poľa a poskytnúť prvý vhodný prvok. Ak nemáme k dispozícii prvok s vyžadovanou dĺžkou, usilujeme sa situáciu riešiť spojením voľných prvkov do jedného - analógia začiatočnej situácie, a ak vyhovuje, poskytneme ho, inak nastáva situácia, že voľná pamäť je vyčerpaná (aspoň na riešenie danej situácie) a signalizuje chybu.

1.4.2 Implementácia reťazca

Ako implementujúci typ údajov pre reťazec možno použiť vektor a zoznam. Voľba závisí od operácií nad reťazcami. Každú implementáciu si opíšeme osobitne.

1.4.2.1 Implementácia reťazca pomocou vektora

Vektor používame ako implementujúci typ údajov pre reťazec vtedy, ak nepotrebujeme reťazce rozbíjať, ale manipulujeme s ním ako s celkom. Najbežnejšou operáciou je potom operácia porovnania reťazcov. Pochopiteľne, že to nemusí byť jediná operácia.

Reprezentácia prvku reťazca závisí od toho, či je jeho dĺžka pevná, alebo sa mení. Najjednoduchejšia situácia je pri pevnej dĺžke prvku. Potom možno výhodne použiť priamo počítačové prostredie a prvok reprezentovať bajtom, prípadne jeho násobkom, napr. polslovom, slovom atď. Takáto reprezentácia umožňuje na manipuláciu s reťazcom výhodne využiť strojový kód daného počítača.

Ak sa dĺžka prvku reťazca mení, potom možno pri reprezentácii prvku postupovať dvoma spôsobmi: zobrať ako základ dĺžku najdlhšieho prvku a uvažovať ju ako pevnú alebo zahrnúť dĺžku prvku do samotného prvku.

Z hľadiska účinnosti manipulácie s reťazcom je niekedy výhodné, ak vektor obsahuje aj informáciu o dĺžke reťazca.

Vektor, ako implementujúci typ údajov reťazca, sa stáva ťažkopádnym vtedy, ak potrebujeme pridávať alebo rušiť prvok. Tieto operácie si vyžadujú posúvať prvky niektorej časti reťazca. V takomto prípade je výhodnejšie použiť zoznam.

Implementovať možno týmto spôsobom:

```
const k =    ;
type index = 0.. max;
     prvok = packed array [1.. k] of char;
     vektor = array [index] of prvok;

procedure concat(var r1,r2 : vektor; var i,j : index);
          /* Zreťazenie reťazcov r1, r2 s dĺžkou udanou parametrami i,j.
             Výsledný reťazec bude v r1 s dĺžkou i */
          var m,n : index;
```

```
        begin n := 1;
              m := i+j;
              for i := i+1 to m do
              begin r1 [i] := r2 [n];
                    n := n+1
              end
        end

function isin (r : vektor; i : index; p : prvok): boolean;
        /* Test, či prvok p je vo vektore r */
        var j : index;
            b : boolean;
        begin j := 1;
              b := false;
              while not b and j < = i do
              begin b := r[j] = p;
                    j := j+1
              end;
              isin := b
        end

function iseq (r1,r2 : vektor; i1,i2 : index): boolean;
        /* Test, či sa reťazce r1 a r2 s dĺžkami i1 a i2 rovnajú. */
        var i : index;
            b : boolean;
        begin b := i1 = i2;
        i := 1;
        while b and i < = i1 do
              begin b := r1[i] = r2[i];
                    i := i+1
              end;
              iseq := b
        end

procedure insert(var r : vektor; var i : index; p : prvok);
        /* Vloženie prvku p do vektora r, pri zachovaní usporiadania
           prvkov vo vektore */
        var j,k : index;
        begin
              r[i+1] := p;
              j := 1;
              while r[j] < p do
                    j := j+1;
              k := i;
              while j < = k do
```

```
        begin r[k+1] := r[k];
              k := k-1
        end;
        r[j] := p;
        i := i+1;
    end
```

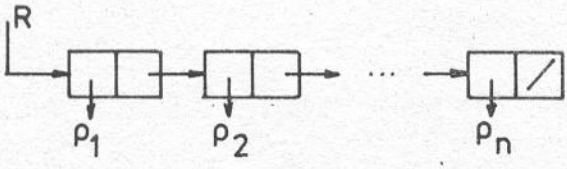
Ostávajúce operácie, a to empty, length, delmax a isempty sú triviálne. Dĺžku reťazca udávame aktuálnym indexom posledného prvku. Pomocou neho možno jednoducho implementovať operácie empty a isempty. Operácia delmax stráca pravý zmysel, pretože všetky prvky majú rovnakú dĺžku. Možno ju však modifikovať na vylúčenie daného prvku. Túto operáciu nazveme delete a je definovaná takto:

delete: string,elm ⟶ string.

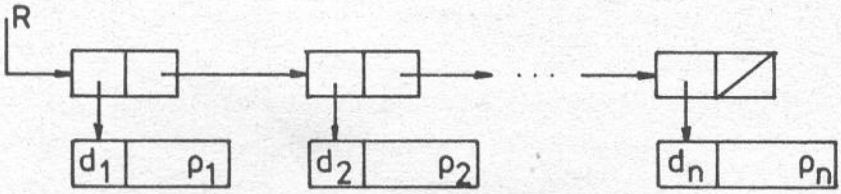
```
procedure delete(var r : vektor; var i : index; var p : prvok);
    /* Vylúčenie prvku p z vektora r */
    var j : index;

    begin
        j := 1;
        r[i+1] := p;
        while r[j]< >p do
              j := j+1;
        while j <i do
        begin r[j] := r[j+1];
              j := j+1
        end;
        i := i-1
    end
```

1.4.2.2 Implementácia reťazca pomocou zoznamu

Zoznam, ako implementujúci typ údajov pre reťazec, používame obyčajne vtedy, ak potrebujeme reťazec rozbíjať a manipulovať s jeho prvkami. Stačí nám použiť jednoúrovňový zoznam, ako dokumentuje obr. 1.23. Ak reťazec obsahuje prvky s premennou dĺžkou, možno postupovať rovnako ako pri použití vektora, teda zobrať maximálnu dĺžku a uvažovať ju ako pevnú, alebo dĺžku zahrnúť do prvku, ako znázorňuje obr. 1.24.



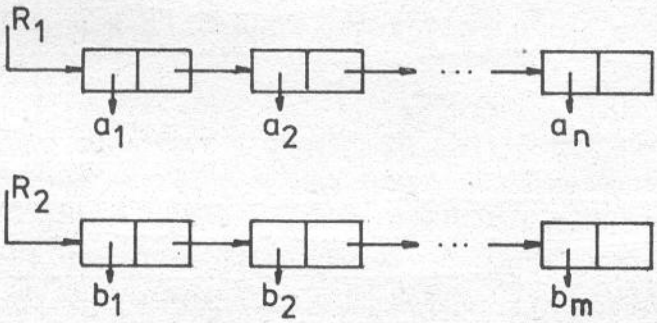
Obr. 1.23
Reprezentácia reťazca pomocou zoznamu



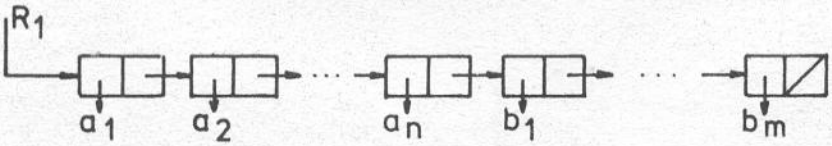
Obr. 1.24
Reprezentácia reťazca s premennou dĺžkou prvkov pomocou zoznamu

Implementáciu niektorých operácii si ukážeme graficky.

concat: string,string $\longrightarrow$ string

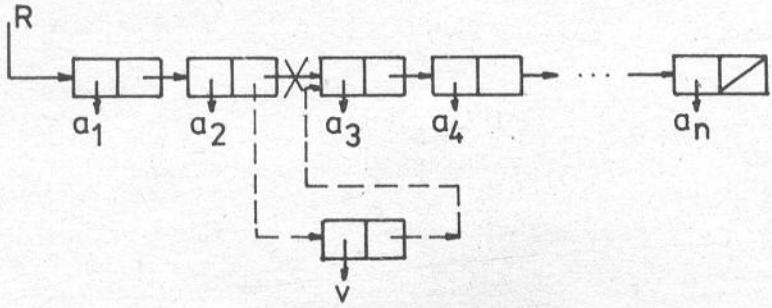


Namiesto NIL zoznamu R_1 zapíšeme R_2, čím dostaneme zoznam



Obr. 1.25
Implementácia operácie concat

insert: string,elm ⟶ string



Obr. 1.26
Implementácia operácie insert

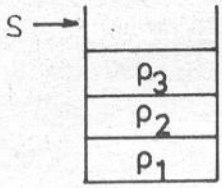
Namiesto smerníka vkladaného prvku vložíme smerník prvku, za ktorý daný prvok vkladáme a ten nahradíme smerníkom na vkladaný prvok.

Analogicky možno implementovať aj ostatné operácie. Niektoré sú však menej účinné ako pri použití vektora. Napríklad test na zhodu dvoch reťazcov - - operácia iseq si vyžaduje prechádzanie celým zoznamom.

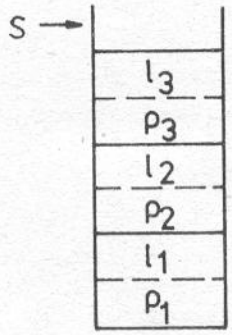
1.4.3 Implementácia zásobníka

Ako implementujúci typ údajov pre zásobník používame vektor. Vektor umožňuje jednoduchú reprezentáciu prvkov zásobníka i implementáciu operácií. Aj teraz môže mať prvok pevnú alebo premennú dĺžku. Ak je dĺžka premenná, možno uvažovať s maximálnou dĺžkou alebo dĺžku zahrnieme do prvku. Vzhľadom na charakter práce so zásobníkom dĺžka prvku sa udáva na konci prvku.

Dynamicky meniaci sa stav zásobníka ošetrujeme pomocou smerníka S, ktorý ukazuje na prvé voľné miesto v zásobníku. Dno zásobníka je určené prvým prvkom vektora (Obr. 1.27, obr. 1.28).

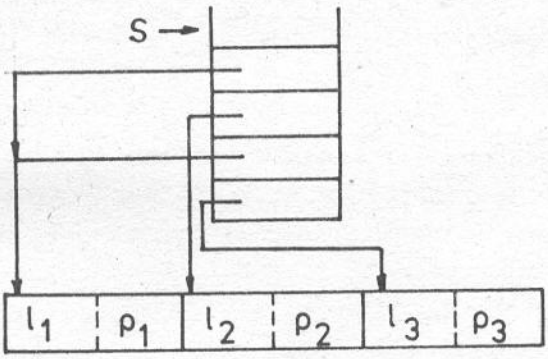


Obr. 1.27
Reprezentácia zásobníka pomocou vektora



Obr. 1.28
Reprezentácia zásobníka, ktorého prvky obsahujú aj informáciu o svojej dĺžke, pomocou vektora

Niekedy je výhodné držať prvky zásobníka mimo samotného zásobníka a v zásobníku držať iba smerníky na dané prvky. Tento spôsob nám môže zjednodušiť prácu, pretože sa pracuje iba so smerníkmi a prvok stačí uchovávať iba raz (Obr. 1.29).



Obr. 1.29
Reprezentácia zásobníka pomocou vektora s uchovávaním prvkov zásobníka mimo zásobníka

Implementáciu si ukážeme pre prípad, že prvky majú premennú dĺžku a informácia o dĺžke je zahrnutá v prvku. Aj samotný prvok bude reprezentovaný vektorom.

```
procedure push(var z : vektor; var i : index; p : vektor; d : integer);
          /* Pridanie prvku p s dĺžkou d do zásobníka z so smerníkom i */
          var j : index;
          begin
               for j := 1 to d do
               begin z[i] := p[j];
                     i := i+1
               end;
               z[i] := d;
               i := i+1
          end
```

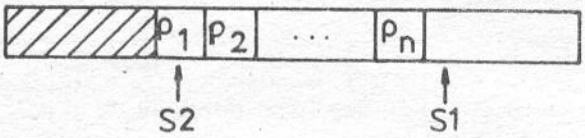
```
procedure top(var z : vektor; var i : index; var p : vektor;
                                             var d : integer);
        /* Výber prvku zo zásobníka z so smerníkom i. Prvok sa uloží
           do vektora p a jeho dĺžka do premennej d */
        var j,k : index;
        begin k := 1;
              d := z[i-1];
              i := i-1;
              for j := d down to 1 do
              begin
                    p[k ] := z[i-j];
                    k := k+1
              end;
              i := i-d
        end

procedure pop(var z : vektor; var i : index);
        /* Zrušenie prvku v zásobníku z so smerníkom i */
        begin
              i := i-z[i-1]-1
        end

function isempty(z : vektor; i : index): boolean;
        /* Test, či je zásobník z so smerníkom i prázdny */
        begin
              isempty := i = 1
        end
```

1.4.4 Implementácia frontu

Typ údajov front možno implementovať pomocou vektora aj pomocou zoznamu.

1.4.4.1 Implementácia frontu pomocou vektora

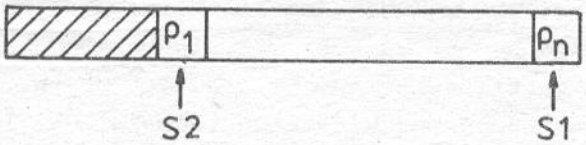
Problém reprezentácie prvku frontu je analogický ako pri zásobníku. Manipulácia s frontom si však vyžaduje dva smerníky; jeden používame na určenie prvku na výber, druhý na určenie miesta na vkladanie prvku. Pridávanie prvku je rovnaké ako pri zásobníku, pričom využívame smerník na pridávanie napr. S1. Výber prvku sa uskutočňuje pomocou druhého smerníka napr. S2. Potom po niekoľkých operáciách môže situácia vyzerať tak, ako to dokumentuje obr. 1.30.



Obr. 1.30
Reprezentácia frontu pomocou vektora

Ak pracujeme s prvkami s premennou dĺžkou a zahŕňame ju do prvku, ukladáme ju ako prvú. Takto vyberieme ako prvú informáciu dĺžky, a tým i informáciu ako dlhý prvok vybrať.

Stratégia "prvý dnu, prvý von" spôsobuje, že pri pridávaní prvkov môžeme naraziť na "koniec" vektora, kedy index dosiahne hornú hranicu a vektor je "plný", ďalší prvok už nemožno pridať. V skutočnosti však vektor nemusí byť zaplnený celý, pretože niektoré prvky sa už mohli vybrať (Obr. 1.31).



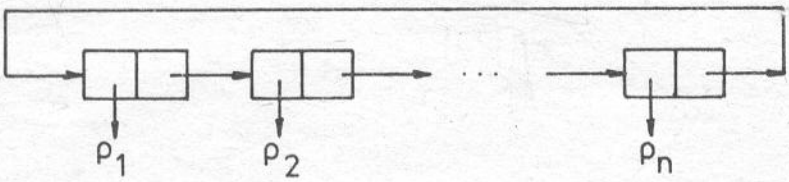
Obr. 1.31
Reprezentácia frontu pomocou vektora - "preplnenie"

Prirodzeným riešením tejto situácie je využitie voľného priestoru na "začiatku" vektora. To znamená, že sa budeme cyklicky vracať na začiatok, čo možno jednoducho dosiahnuť tak, že budeme smerník počítať podľa modulo určené dĺžkou vektora. Zostáva ešte rozriešiť situáciu, keď obidva smerníky ukazujú na rovnaké miesto. Táto situácia môže nastať vtedy, keď je front prázdny alebo plný a všetky prvky vektora sú obsadené. Front bude prázdny vtedy, ak smerník S2 "dobehne" smerník S1, a opačne, ak smerník S1 "dobehne" cyklicky smerník S2 front je plný. Rozhodujúca je teda posledná operácia.

Programová implementácia je analogická ako pri zásobníku a prenechávame ju na cvičenie.

1.4.4.2 Implementácia frontu pomocou zoznamu

Vzhľadom na spôsob práce s frontom, možno front veľmi prirodzene implementovať cyklickým zoznamom. Takáto implementácia zodpovedá v podstate stratégii výpočtu smerníka podľa modulo pri použití vektora. Reprezentácia frontu pomocou zoznamu je na obr. 1.32.



Obr. 1.32
Reprezentácia frontu pomocou cyklického zoznamu

Programovú implementáciu prenechávame na cvičenie.

1.4.5 Implementácia zoznamu

Najvhodnejším implementujúcim typom údajov pre zoznam je zreťazená voľná pamäť. Požiadavky na organizáciu prvku zoznamu sú dané charakterom operácií. Prvok bude pozostávať z troch smerníkov: na predchodcu, nasledovníka a vlastnú hodnotu. Hodnotou prvku môže byť atóm alebo zoznam pri hierarchickej štruktúre. Na ich rozlíšenie budeme používať príznak atómu, ktorý bude súčasťou prvku. Organizácia prvku zoznamu teda je:

succ - smerník na nasledujúci prvok,
szoznam - smerník na hodnotu - zoznam,
satom - smerník na hodnotu - atóm,
patom - príznak atómu,
pred - smerník na predchádzajúci prvok.

Keďže Pascal používame ako implementujúci jazyk, možno využiť jeho prostriedky na manipuláciu s dynamickými objektami. S dynamickou pamäťou možno manipulovať pomocou štandardných procedúr new a dispose. New(s) nastaví smerník s na voľné miesto v dynamickej pamäti, dispose(s) vráti dynamickú premennú, na ktorú ukazuje smerník s do voľnej pamäti.

Ukážeme si implementáciu niektorých operácií.

```
type point = ↑prvok
     prvok = record
             pred : point;
             succ : point;
             patom : boolean;
             case patom of
             true : satom : integer;
             false : szoznam : point
             end;
```

```
var s,r : point;

procedure empty(var s : point);
          /* Vytvorenie prázdneho zoznamu */
          begin s := nil end

procedure first(s : point; var r : point);
          /* Nastavenie smerníka r na začiatok zoznamu s, pričom predpo-
             kladáme, že začiatočný je prvok, ktorý má prázdny - nil
             smerník na predchodcu */
          begin
                r := s;
                while r < >nil do
                r := r↑. pred
          end

procedure last(s : point; var r : point);
          /* Nastavenie smerníka r na koniec zoznamu s */
          begin
                r := s;
                while r < >nil do
                r := r↑. succ
          end

procedure P-insert(var t : point; r : point);
          /* Vloženie prvku r do zoznamu pred prvok t */
          var z : point;
          begin
          r↑. succ := t↑. pred↑. succ;
          r↑. pred := t↑. pred;
          t↑. pred↑. succ := r;
          t↑. pred := r
          end

Procedure R-insert(var s : point);
          /* Vloženie prvku, ktorého atomickú hodnotu typu integer prečí-
             tame zo vstupu, do zoznamu pred prvok s */
          var atom : integer;
              t : pointer;
          begin
                read (atom);
                new (t);
                t.satom := atom;
                t.patom := true;
```

```
        if s = nil then
                        begin
                              t↑.pred := nil;
                              t↑.succ := nil;
                              s := t
                        end
                    else
        if s↑.pred = nil then
                              begin
                                    t↑.pred := nil;
                                    t↑.succ := s;
                                    s := t
                              end
                          else
                              begin
                                    t↑.succ := s↑.pred↑.succ;
                                    t↑.pred := s↑.pred;
                                    s↑.pred↑.succ := t;
                                    s↑.pred := t
                              end

procedure N-delete(var r : point);
        /* Zrušenie prvku zoznamu označeného smerníkom r a nastavenie
           smerníka r na nasledujúci prvok */
        begin
              if r = nil then error else
              if r↑.succ = nil then r := nil else
              if r↑.pred = nil then
                              begin
                                    r := r↑.succ;
                                    r↑.pred := nil
                              end
                          else
                              begin
                                    r↑.pred↑.succ := r↑.succ;
                                    r↑.succ↑.pred := r↑.pred;
                                    r := r↑.succ
                              end
        end
```

1.4.6 Implementácia poľa

Pole, až na špeciálne prípady, implementujeme pomocou vektora. Takým špeciálnym prípadom sú riedko obsadené polia, pre ktoré používame ako implementujúci typ údajov tabuľku. Teraz si rozoberieme implementáciu poľa pomocou vektora.

Problém implementácie poľa pomocou vektora spočíva v zobrazení vo všeobecnosti n-rozmerného poľa na vektor. V prípade jednorozmerného poľa, napr. A, s hranicami indexu L a H, prvok A[i] sa zobrazí na prvok vektora V[i-L+1]. Ak budeme začiatok vektora reprezentovať nejakou bázou, pre zobrazenie prvku A[i] možno použiť zápis

$$\text{báza} + (i-L)$$

Dvojrozmerné pole $A[L_1..H_1, L_2..H_2]$ možno zobrazovať "po riadkoch", keď sa rýchlejšie mení druhý - teda posledný index alebo "po stĺpcoch", keď sa rýchlejšie mení prvý index. Prvok $A[i_1,i_2]$ sa zobrazí:

po riadkoch: $\text{báza} + (i_1-L_1)(H_2-L_2+1) + (i_2-L_2)$

po stĺpcoch: $\text{báza} + (i_2-L_2)(H_1-L_1+1) + (i_1-L_1)$

Prvok poľa sprístupníme realizovaním príslušnej zobrazovacej funkcie, reprezentovanej uvedenými výrazmi. Požiadavka rýchlosti sprístupnenia prvkov poľa nás núti realizovať zobrazovaciu funkciu čo najefektívnejšie. Z uvedených výrazov vidíme, že niektoré veličiny sú v nich pre dané pole konštantné, a preto ich netreba počítať pri každom sprístupňovaní. Rozoberme si z tohto pohľadu zobrazenie dvojrozmerného poľa po riadkoch. Výraz možno rozdeliť na dve zložky : konštantnú a premennú:

Nech $H_2-L_2+1=d_2$. Potom pre zobrazenie prvku $A[i_1,i_2]$ možno písať:

$$\text{báza} + i_1d_2 + i_2 - (L_1d_2 + L_2)$$

kde $i_1d_2 + i_2$ je premenná časť, závislá od indexov,
$L_1d_2 + L_2$ je pre dané pole konštantná časť.

Rovnakú analýzu možno urobiť aj pre n-rozmerné pole $A[L_1..H_1, L_2..H_2, \ldots, L_n..H_n]$, pre zobrazenie ktorého platí:

$$\begin{aligned}\text{báza} &+ (i_1-L_1)d_2d_3\ldots d_n + \\ &+ (i_2-L_2)d_3d_4\ldots d_n + \\ &\quad\vdots \\ &+ (i_{n-1}-L_{n-1})d_n + (i_n-L_n)\end{aligned}$$

z čoho úpravou dostaneme pre konštantnú časť

$$(\ldots((L_1 d_2+L_2)d_3+L_3)d_4+\ldots+L_{n-1})d_{n-1}+L_n$$

a pre premennú časť

$$(\ldots((i_1 d_2+i_2)d_3+i_3)d_4+\ldots+i_{n-1})d_{n-1}+i_n$$

Zobrazenie možno písať v tvare

$$\text{báza} + \sum_{m=1}^{n} (i_m-L_m)d_m$$

kde pre d_m platí:

pri zobrazení "po riadkoch"
$$d_n = 1$$
$$d_m = (H_{m+1}-L_{m+1}+1)d_{m+1}$$
$$m = 1,2,\ldots,n-1$$

pri zobrazení "po stĺpcoch"
$$d_1 = 1$$
$$d_m = (H_{m-1}-L_{m-1}+1)d_{m-1}$$
$$m = 2,3,\ldots,n$$

Z hľadiska účinnosti sprístupňovania prvkov je výhodné zapamätať si jednotlivé D_m a $\sum_{m=1}^{n} i_m d_m$, čím sa vyhneme viacnásobnému počítaniu. Uchovávanie týchto informácií možno realizovať v zásade dvoma spôsobmi:

a. zapamätať ich priamo so samotným poľom,
b. použiť na ich zapamätanie špeciálny, tzv. informačný vektor.

Bežnejšie je použitie informačného vektora. Tento spôsob umožňuje použiť rovnaký informačný vektor pre dve rôzne polia, ktoré majú rovnaké hranice indexov.

V informačnom vektore sa uchovávajú aj ďalšie informácie umožňujúce kontrolu správnosti zápisu indexovanej premennej, ako sú rozsahy indexov, prípadne ich počet. Informačný vektor zvyčajne obsahuje:

1. počet rozmerov,
2. dolné a horné hranice každého indexu,
3. počet prvkov poľa $\left(\prod_{i=1}^{n} (H_i-L_i+1),\right)$
4. D_m pre $m = 1,2,\ldots,n$,
5. $\prod_{m=1}^{n} i_m D_m$.

Informačný vektor možno reprezentovať aj v tvare, ako dokumentuje obr. 1.33.

L_1	H_1	d_1
L_2	H_2	d_2
⋮	⋮	⋮
L_n	H_n	d_n
n	konštant.časť	
BÁZA		

Obr. 1.33
Informačný vektor

Druhý spôsob zefektívnenia sprístupňovania prvkov poľa je založený na tzv. Iliffových vektoroch. Tento prístup je náročnejší na pamäť, ale rýchlejší. S každým poľom je spojená množina vektorov, ktorých prvky sú smerníky zabezpečujúce hierarchickú organizáciu poľa. S prvým indexom je spojený jeden (H_1-L_1+1) prvkový vektor V_1, na ktorý ukazuje bázový smerník. S druhým indexom je spojených $d_1=(H_1-L_1+1)$ vektorov $V_{2,1}, V_{2,2}, \ldots V_{2,k}$, každý s (H_2-L_2+1) prvkami, pričom prvý smerník vektora V_1 ukazuje na vektor $V_{2,1}$, druhý na $V_{2,2}$ atď. Pre všetky ostatné úrovne je situácia analogická.

Ak máme pole $A[L_1..H_1,\ldots,L_n..H_n]$, potom prvok $A[i_1,\ldots,i_n]$ sa sprístupní týmto spôsobom:

$$((((\text{báza})+i_1)+i_2)+\ldots+i_{n-1})+i_n)$$

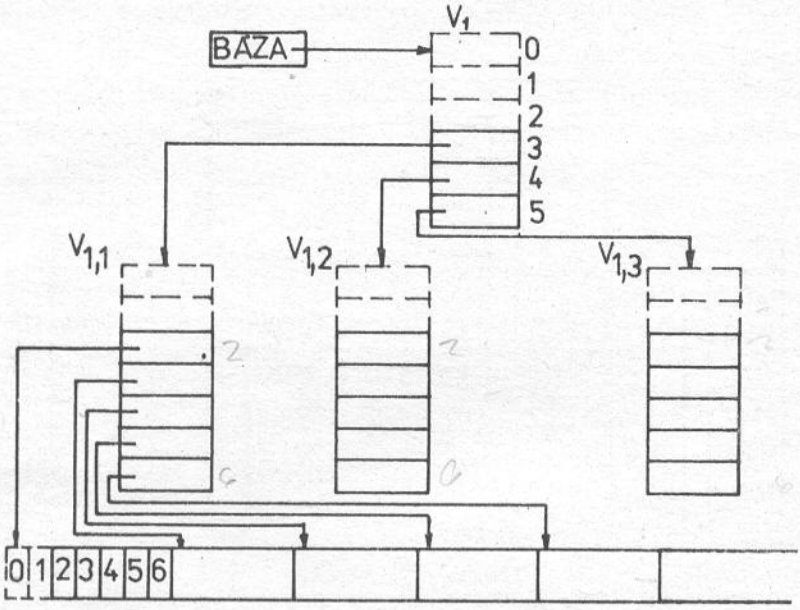
pričom vyhodnotenie treba robiť v zmysle smerníkov.

Výhoda uvedeného prístupu spočíva v tom, že sprístupnenie robíme iba pomocou operácie sčítania.

Príklad 1.4.1

Použitie Iliffových vektorov si ukážeme na sprístupnení prvkov poľa A[3..5,2..6,2..7]. Štruktúra sprístupňovania je na obr. 1.34.

Doterajšie úvahy sme venovali homogénnemu poľu. Uvedený spôsob implementácie homogénneho poľa nemožno priamočiaro použiť aj pre heterogénne pole. Nie je to reálne ani pre reprezentáciu prvkov, pretože pri heterogénnych po-



Obr. 1.34
Sústava Iliffových vektorov pre pole:
A[3..5,2..6,2..7]

liach nemusia mať prvky rovnakú dĺžku, ani implementáciu operácií, pretože závisia od typu prvkov. Naviac, typ prvkov môže byť vopred deklarovaný a počas spracovania sa už nemení, alebo je bez deklarácie a pripúšťame zmenu počas spracovania.

Najskôr vyšetríme heterogénne pole s deklaráciou typu prvkov. Uvažujeme heterogénne pole Student obsahujúce informácie o študentovi. O študentovi uchovávame: identifikačné číslo - päťciferné, meno - 20 znakové, dátum narodenia - šesťciferné číslo. V jazyku Pascal možno definovať zodpovedajúce heterogénne pole - záznam takto:

```
type T = record
            číslo: 1..99999;
            meno : alfa;
            dátum: 1..991231
         end
```

Ako implementujúci typ použijeme aj teraz vektor. O poli a jeho prvkoch však potrebujeme teraz viac informácii ako v prípade homogénneho poľa. O poli máme informácie, ako sú typ poľa, rozmernosť poľa (počet indexov), počet prvkov. O každom prvku máme informácie, ako typ prvku a jeho dĺžka. Tieto informácie uchovávame v tzv. deskriptore.

Štruktúra zodpovedajúceho deskriptora a vektora pre pole STUDENT bude podľa obr. 1.35. V tomto prípade ide o jednorozmerné heterogénne pole.

Deskriptor T :

Heterogénne pole	
3	
integer	1
alfa	5
integer	1

- typ poľa
- počet prvkov
- pre každý prvok: typ a dĺžka

Vektor V :
STUDENT: číslo

meno

dátum

Obr. 1.35
Heterogénne pole STUDENT: deskriptor, implementujúci vektor

Zobrazovacia funkcia poľa na vektor je analogická ako pri homogénnych poliach, musí však odrážať charakteristiky heterogénnosti, a to je rozdielna dĺžka prvkov. Na druhej strane dolná a horná hranica indexu je daná počtom prvkov na danej úrovni. Prvok STUDENT[i] sa zobrazí na prvok vektora daný zobrazovacou funkciou

$$\text{STUDENT } i = \text{BAZA} + \sum_{k=DH}^{i-1} \text{dĺžka} \cdot (\text{STUDENT}[k])$$

kde DH je dolná hranica indexu a bude rovná 1.

Aj pri heterogénnych poliach sa usilujeme zobrazovaciu funkciu zjednodušiť. Vzhľadom na charakter heterogénnych polí je najjednoduchšou stratégiou "stratégia rovnakých prvkov". Potom sumu v zobrazovacej funkcii nahradíme jedinou operáciou násobenia. Najčastejšie však priradíme prvkom adresu už počas prekladu a používame ich symbolické označenie.

Pochopiteľne, že aj heterogénne pole môže byť viacrozmerné. Takáto situácia nastáva vtedy, ak sú prvky opäť heterogénne polia. Aj v našom prípade poľa STUDENT môžeme užitočné prvky ďalej štruktúrovať. Pri čísle študenta by nás mohlo zaujímať: študijný odbor, ročník, krúžok. Pri mene: krstné meno a priezvisko. Dátum je jasný: rok, mesiac, deň. Dostaneme dvojrozmerné heterogénne pole, ktoré možno opísať takto:

```
type TS = record
            číslo : record
                        odbor : 1..10;
                        ročník: 1..6;
                        krúžok: 1..20;
                    end;
            meno :  record
                        krstné    : alfa;
                        priezvisko: alfa;
                    end;
            dátum : record
                        rok   : 1..99
                        mesiac: 1..12
                        deň   : 1..31
                    end
end
```

Pri heterogénnych poliach reprezentujú prvky zvyčajne nejakú sémantickú jednotku aplikačného prostredia, čo vplýva aj na spracovanie polí. Z tohto hľadiska je výhodnejšie zobrazovať heterogénne polia "po riadkoch", čím sa zjednoduší prístup, pretože jedna operácia umožňuje sprístupniť celú sémantickú jednotku.

Aj pri implementácii viacrozmerných heterogénnych polí uchovávame informácie o poli v deskriptore.

Problém implementácie heterogénnych polí bude ešte zložitejší, ak budeme uvažovať polia bez deklarácií. V tomto prípade pripúšťame zmenu typu prvkov počas spracovania, a ako dôsledok bude nielenže dĺžka prvkov rôzna, ale aj premenná. Preto pridávame k deskriptoru ešte vektor indexov, ktorý obsahuje pre každý prvok jeho typ a adresu umiestnenia.

1.4.7 Implementácia tabuľky

Tabuľku, až na špeciálny prípad, tzv. rozhodovacej tabuľky, implementujeme pomocou vektora. Hlavným problémom implementácie je sprístupnenie prvku. Prvky tabuľky sa sprístupňujú vždy pomocou kľúča spojeného s daným orvkom. Podľa spôsobu používania kľúča pri sprístupňovaní prvkov rozdeľujeme tabuľky na:

a. vyhľadávacie alebo sekvenčné,
b. tabuľky s rozptýlenými prvkami.

Vyhľadávacie alebo sekvenčné tabuľky sú charakterizované tým, že prvky sú v tabuľke uložené "husto" vedľa seba od začiatku tabuľky a sú sprístupňované bezprostredne na základe kľúča. Sprístupňovanie sa robí rôznou formou prehľadávania.

Tabuľky s rozptýlenými prvkami sú charakterizované tým, že tabuľka je obsadená "riedko", teda prvky sú rozptýlené po celej tabuľke a sprístupnenie sa robí na základe informácie odvodenej z kľúča. Informáciu na sprístupnenie prvkov získavame pomocou vhodnej funkcie.

Kvalitu určitej implementácie tabuľky stanovujeme na základe jej požiadaviek na pamäť a čas potrebný na sprístupnenie prvku. Pre každú implementáciu si zavedieme tieto parametre:

S je dĺžka hľadania položky v tabuľke. Ide o počet prvkov, ktoré je potrebné vyšetriť, kým nájdeme daný prvok.

T - priemerná dĺžka hľadania.

$$T = \frac{1}{m} \sum_{i=1}^{m} S_i$$

kde m je počet realizovaných prístupov do tabuľky.
Pretože prístupy S_i k rôznym prvkom nie sú vo všeobecnosti rovnaké, závisí priemerná dĺžka hľadania od frekvencie prístupu k jednotlivým položkám. Preto T určujeme štatistickým meraním na implementovanej tabuľke.

A je priemerná dĺžka hľadania za predpokladu rovnomerného prístupu k prvkom

$$A = \frac{1}{n} \sum_{i=1}^{n} S_i$$

kde n je počet prvkov v tabuľke.

P - zaplnenie tabuľky

$$P = \frac{n_s}{n_f} \cdot 100 \qquad [\%]$$

kde n_s je skutočný počet prvkov v tabuľke,
n_f - predpokladaný počet prvkov v tabuľke - vyhradená pamäťová oblasť.

1.4.7.1 Vyhľadávacia tabuľka

Na sprístupnenie prvkov vyhľadávacej tabuľky používame rôzne formy vyhľadávania. Najjednoduchším je lineárne alebo sekvenčné vyhľadávanie. Postupne prezeráme prvky tabuľky, kým nenájdeme prvok s daným kľúčom, alebo neprídeme na koniec tabuľky, kedy vhodným spôsobom indikujeme, že prvok s daným kľúčom nie je v tabuľke. Procedúru lineárneho vyhľadávania možno implementovať týmto spôsobom:

```
type index = 1..max;
     prvok = record
                    kľúč    : integer;
                    položka : T;
             end
     vektor = array[index] of prvok;

procedure LINHL(T : vektor; p : prvok; n : integer; var k : integer);
          /* Hľadanie prvku s kľúčom p v tabuľke T, ktorá má n prvkov.
             Výstup bude k, udávajúce index nájdeného prvku vo vektore,
             alebo k=0, ak tam prvok s daným kľúčom sa nenachádza */
          begin
               k := 1;
               T[n+1].kľúč := p;
               while T[k].kľúč <>p do
                     k := k+1;
               if k = n+1 then k := 0
          end
```

Ako vidíme, dĺžka hľadania S prvku závisí od jeho umiestnenia v tabuľke. Preto často používané prvky umiestňujeme na začiatok. Priemerná dĺžka hľadania A pre tabuľku s n prvkami je:

$$A = \frac{1}{n}\sum_{i=1}^{n} S_i = \frac{1}{n} \cdot \frac{1}{2} n(n+1) = \frac{1}{2}(n+1)$$

Naplnenie tabuľky je 100 %.

Z parametrov vyhľadávacích tabuliek vidieť, že sú výhodné z hľadiska využitia pamäti, menej však, najmä pri väčších tabuľkách, z hľadiska dĺžky hľadania.

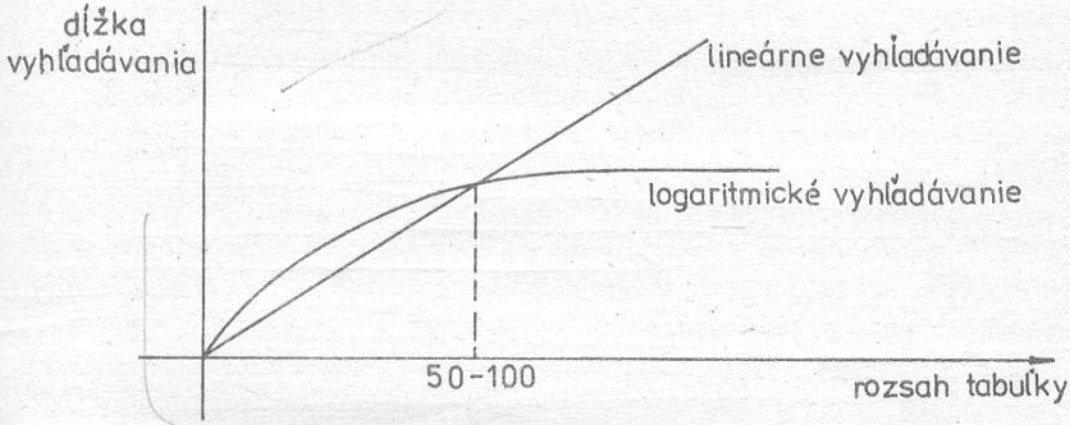
Ak sa tabuľka nemení, teda sa do nej prvky nepridávajú a ani sa v nej nerušia, možno sprístupnenie prvkov zefektívniť tak, že prvky tabuľky usporiadame podľa kľúčov a na sprístupnenie použijeme vyhľadávanie polením alebo logaritmické vyhľadávanie. Je založené na tomto postupe: hľadáme najskôr

v strede (v polovici) tabuľky. Ak tam hľadaný prvok je, STOP. Inak, vzhľadom na usporiadanie prvkov vieme, či hľadaný prvok je v prvej alebo druhej polovici a celý proces opakujeme pre túto polovicu. Pri zachovaní 100 % využitia pamäti pre priemernú dĺžku hľadania A platí

$$A = \frac{1}{2} \log_2 n$$

kde n je počet prvkov tabuľky.

Vzťah medzi logaritmickým a lineárnym vyhľadávaním je na obr. 1.36.



Obr. 1.36
Vzťah lineárneho a logaritmického vyhľadávania

Z obr. 1.36 vidieť, že do rozsahu tabuľky 50 až 100 prvkov je rýchlejšie lineárne vyhľadávanie, pre väčšie tabuľky logaritmické. Treba si však pamätať, že logaritmické vyhľadávanie predpokladá usporiadanosť prvkov do tabuľky.

1.4.7.2 Tabuľka s rozptýlenými prvkami

Sprístupnenie prvkov tabuľky s rozptýlenými prvkami sa uskutočňuje na základe informácie odvodenej od kľúča. Používame na to tzv. rozptylovú funkciu, definovanú ako zobrazenie

$$f: K \longrightarrow A$$

kde K je množina kľúčov,

A - množina adries alebo adresných kľúčov, ktoré reprezentujú v podstate index vektora, pomocou ktorého tabuľku implementujeme. Obor hodnôt je teda daný rozsahom indexu implementujúceho vektora.

Ak pre ľubovoľné dva kľúče k_1, k_2 platí

$$k_1 \neq k_2 \Longrightarrow f(k_1) \neq f(k_2)$$

dostávame tabuľku s priamym prístupom. Takáto situácia však nenastáva často. Už samotná skutočnosť, že počet možných kľúčov je zvyčajne oveľa väčší ako je rozsah tabuľky, naznačuje nároky na vytvorenie rozptylovej funkcie. Na druhej strane si treba uvedomiť, že z celkového počtu možných kľúčov používame zvyčajne iba veľmi obmedzený počet. Napríklad počet identifikátorov, ktoré možno vytvoriť z 26 znakov abecedy a 10 číslic, pri dĺžke identifikátora 6 je

$$26.36^5,$$

z ktorých však v bežnom programe použijeme iba malú časť. Rozptylová funkcia však musí dávať možnosť vytvoriť ľubovoľný prípustný identifikátor.

Aby sme vyriešili uvedený problém, upúšťame od požiadavky $k_1 \neq k_2 \Longrightarrow f(k_1) \neq f(k_2)$, t.j. pripúšťame, že na rovnaké miesto zobrazíme viac kľúčov. Takejto situácii hovoríme kolízia a prvkom synonymá. Usilujeme sa vytvoriť takú rozptylovú funkciu, ktorá poskytuje čo najmenej kolízií. Praktické použitie takejto funkcie si vyžaduje problém kolízie jednoznačne rozriešiť. Skôr ako si rozoberieme metódy riešenia kolízie, ukážeme si celý problém na príklade.

Príklad 1.4.2

Uvažujme tabuľku identifikátorov, ktorú implementujeme pomocou vektora V[0..9]. Ako rozptylovú funkciu použijeme funkciu KOD(α), kde α je identifikátor a KOD(α), získame ako súčet numerických hodnôt jednotlivých znakov α kódovaných takto:

0 - 1, 1 - 2, ..., 9 - 10, A - 11, B - 12, Z - 36. Teda
KOD(1) = 2, KOD(15) = 2 + 6 = 8, KOD(A7) = 11 + 8 = 19

Už z uvedených hodnôt funkcie KOD vyplýva, že nespĺňa podmienku rozptylovej funkcie vzhľadom na obor hodnôt, ktorým je interval <0,9>. Upravíme funkciu KOD tak, že jej hodnoty budeme počítať podľa určitého modulo, v našom prípade 10. Teda

$$f(\alpha) = \mathrm{KOD}(\alpha) \bmod 10$$

Ak máme teraz vložiť identifikátory (ako kľúče) A1, ABC, EF, I2, JK7, umiestnia sa nám takto (Obr. 1.37):

0	
1	EF
2	I2
3	A1
4	
5	
6	ABC
7	
8	
9	JK7

$KOD(A1)_{mod10} = (11+2)_{mod10} = 3$

$KOD(ABC)_{mod10} = (11+12+13)_{mod10} = 6$

$KOD(EF)_{mod10} = (15+16)_{mod10} = 1$

$KOD(I2)_{mod10} = (19+3)_{mod10} = 2$

$KOD(JK7)_{mod10} = (20+21+8)_{mod10} = 9$

Obr. 1.37
Tabuľka identifikátorov

Až doteraz nevznikla žiadna kolízia. Ak by sme však potrebovali vložiť identifikátor A4, potom $KOD(A4)_{mod10} = (11+5)_{mod10} = 6$. Na V[6] však už máme ABC. Vznikla kolízia, ktorú treba riešiť. Z obr. 1.37 vidieť, že prvok A4 možno vložiť na rôzne miesta. Kam ju však vložíme, nie je zatiaľ definované.

Kolíziu možno riešiť rôznym spôsobom. Najčastejšie používame:

a. tabuľky s otvoreným rozptýlením,
b. tabuľky s vnútorným zreťazením,
c. tabuľky s vonkajším zreťazením.

$A = C + \frac{P}{N}$ P – počet prv., N – miesto, konšt

A. Otvorené rozptýlenie

Otvorené rozptýlenie je taký spôsob riešenia kolízie, že rozptylová funkcia je definovaná ako postupnosť funkcií $f_0, f_2, \ldots, f_n$, kde f_0 je tzv. bázová funkcia, z ktorej sa ďalšie určitým spôsobom odvádzajú. Pre všetky f_i platí: $f_i: K \longrightarrow A$. Algoritmus sprístupnenia prvku je:

Vstup: prvok s kľúčom α,
f ako postupnosť $f_0, f_1, \ldots, f_m$
T[0..n-1] implementujúci vektor.

Výstup: $f(\alpha)$ - adresa (index) + informácia, či na danom mieste je prvok alebo miesto voľné.

Metóda:

1. i := 0;
2. Počítaj $f_i(\alpha)$ s týmto dopadom:
   a. ak $T[f_i(\alpha)] = \emptyset$, t.j. miesto je prázdne, potom $f(\alpha) = f_i(\alpha)$, STOP, miesto je prázdne.

b. ak $T[f_i(\alpha)] \neq \emptyset$, zisti, či na $T[f_i(\alpha)]$ nie je daný prvok. Ak áno, potom $f(\alpha) = f_i(\alpha)$, STOP, položka tam už je. Ak tam daný prvok nie je, nastala kolízia. i := i+1. Ak $i \leq n$, skok na 2, ak nie, STOP, prvok nemožno vložiť - vektor je plný, alebo f nie je dobrá.

Z algoritmu vidieť, že rýchlosť sprístupnenia závisí od "kvality" rozptylovej funkcie. Od rozptylovej funkcie vyžadujeme, aby sme dostali čo najmenej kolízií a bola čo najjednoduchšia z hľadiska jej vyhodnotenia.

Ako sme už povedali, funkcie $f_0, f_1, \ldots, f_m$ sú určitým spôsobom viazané, pričom f_{i+1} získame z f_i jej určitou modifikáciou. Najčastejšie sa používajú tieto modifikácie:

a. lineárna,
b. pomocou náhodných čísiel.

Lineárna modifikácia predstavuje pokus umiestniť prvok na miesto vzdialené od daného o nejakú konštantu. Najjednoduchšie je skúsiť susedné miesto. Potom

$$f_{i+1}(\alpha) = f_i(\alpha) + 1$$

Nevýhodou takejto modifikácie je vznik tzv. strapca, t.j. zoskupenia prvkov, ktoré nútia k ďalšej modifikácii rozptylovej funkcie. Ak by sme, napr. v našom príklade 1.4.2, vkladali ako ďalší prvok B3, potom $f_0(B3) = 6$ a testovaním by sme zistili, že toto miesto je obsadené iným prvkom. Počítame preto $f_1(B3) = 7$, ale toto miesto je tiež obsadené a až $f_2(B3)$ je voľné. Situácia by sa ešte skomplikovala pri vkladaní ďalšieho prvku, pretože by bolo treba uvažovať aj strapec z T[9]. Preto pristupujeme k modifikácii tvaru

$$f_{i+1}(\alpha) = f_i(\alpha) + P$$

kde P je celé číslo a nazýva sa modifikačná konštanta.

Modifikácia pomocou náhodných čísiel je taká, pri ktorej ako modifikačnú konštantu používame náhodné číslo.

Z algoritmu pre otvorené rozptýlenie vyplýva možnosť vzniku situácie, že rozptylová funkcia je konštruovaná tak, že v prípade kolízie nenájdeme voľné miesto ani vtedy, ak tabuľka nie je celkom zaplnená. Modifikačnú konštantu môžeme voliť tak, že vyšetrujeme iba "párne" alebo "nepárne" miesta a pod. Z tohto hľadiska má najjednoduchší prípad voľby modifikačnej konštanty, a to 1 výhodu v tom, že ak je v tabuľke voľné miesto, nájdeme ho; je to však na úkor sekvenčného vyhľadávania. Záruka vyhľadania voľného miesta v tabuľke prevyšuje hodnotu sekvenčného vyhľadávania. Modifikačná konštanta 1 sa používa veľmi často.

Priemerná dĺžka vyhľadávania prvku pre tabuľku s otvoreným rozptýlením je daná približným vzťahom:

$$A(P) = \frac{2 - \sigma}{2 - 2\sigma}$$

kde $\sigma = \frac{n_s}{n_f}$

Experimenty ukazujú, že ešte pri 80 % zaplnení tabuľky je dĺžka vyhľadávania prijateľná a pohybuje sa okolo 3.

B. Tabuľka s vnútorným zreťazením

Pri tabuľke s vnútorným zreťazením riešime kolíziu takým spôsobom, že synonymné prvky zreťazujeme. Pri vyhľadávaní nepostupujeme na základe modifikácie rozptylovej funkcie resp. sekvenčne, ale na základe zreťazenia. Zreťazenie sa zabezpečuje tak, že súčasťou každého prvku je smerník. Všetky problémy, o ktorých sme hovorili pri otvorenom rozptýlení, zostávajú, iba sa prenášajú na úroveň, keď synonymné prvky zreťazujeme. Preto sa ani v tomto prípade nevyhneme sekvenčnému vyhľadávaniu.

Sekvenčné vyhľadávanie možno vylúčiť tak, že do tabuľky vkladáme najskôr nesynonymné prvky a synonymá ukladáme mimo tabuľky. Po vložení nesynonymných prvkov do tabuľky vložíme do nej aj synonymá a zreťazíme ich.

Jeden možný stav tabuľky identifikátorov z príkladu 1.4.2 s použitím vnútorného reťazenia je na obr. 1.38.

Index	Identifikátor	Smerník na ďalšie synon.
0		
1	EF	4
2	I2	
3	A1	5
4	I1	7
5	J2	
6	ABC	
7	D6	
8		
9		

Obr. 1.38
Tabuľka identifikátorov s vnútorným zreťazením

Hľadanie v tabuľke s vnútorným zreťazením je rýchle. Za predpokladu náhodného rozloženia identifikátorov platí

$$A(n_s,n_f) = 1 + \frac{n_s - 1}{2n_f}$$

C. Tabuľka s vonkajším zreťazením

Tabuľka s vonkajším zreťazením je taká, v ktorej synonymné prvky zreťazujeme a umiestňujeme mimo tabuľky v tzv. zóne zreťazenia alebo preplnenia. Prednosťou tabuľky s vonkajším zreťazením je, že si nevyžaduje sekvenčné vyhľadávanie, pravda na úkor využitia vlastnej tabuľky.

Pri práci s tabuľkami sa stretávame ešte s jedným vážnym problémom - s preplnením tabuľky. Riešenie problému preplnenia závisí od druhu použitej tabuľky. Pri vyhľadávacej tabuľke stačí zväčšiť implementujúci vektor. Pri tabuľke s rozptýlenými položkami závisí tento problém od spôsobu riešenia kolízie. Pri použití tabuľky so zreťazením synonymných prvkov je situácia jednoduchá: rozptylová funkcia zostane nezmenená, rozšíri sa iba priestor na zreťazenie synonymných prvkov. Horšia situácia je pri použití tabuľky s otvoreným rozptýlením. Preplnenie možno riešiť takto:

a. Tabuľku rozšírime, rozptylovú funkciu ponecháme nezmenenú. Toto riešenie je jednoduché, ale náročné na vyhľadávanie.
b. Tabuľku rozšírime a zmeníme rozptylovú funkciu tak, aby prvky, ktoré sú v tabuľke, zobrazovala na pôvodné miesto a nemuseli sa presúvať.
c. Tabuľku rozšírime, zmeníme rozptylovú funkciu a urobíme nové rozptýlenie prvkov.

1.4.8 Implementácia grafu

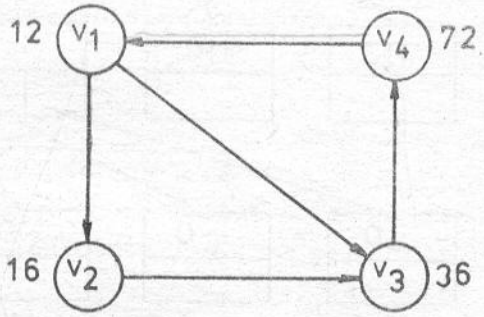
Základný problém implementácie grafu je jeho reprezentácia. Najčastejšie sa používajú reprezentácie:

a. incidenčnou maticou,
b. zoznamom.

Incidenčná matica grafu G = (V,H) je dvojrozmerné pole A s indexovým typom V x V, pričom za V sa uvažuje zvyčajne interval prirodzených čísiel. Prvkami incidenčnej matice sú logické hodnoty. A[i,j] = T práve vtedy, ak hrana (i,j) patrí do H. Ak je graf ohodnotený, potom hodnoty priradené vrcholom resp. hranám sa uchovajú v ďalšom poli, napr. B, s indexovým typom V resp. H.

Príklad 1.4.3

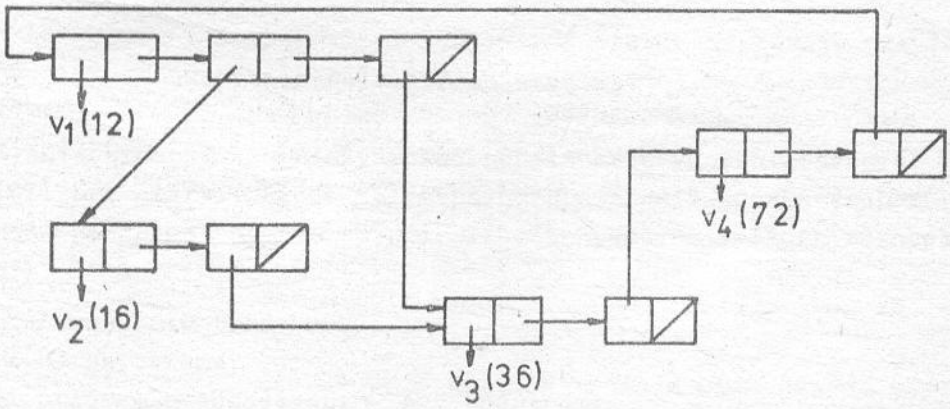
Majme graf podľa obr. 1.39. Potom tento graf možno reprezentovať incidenčnou maticou A a hodnoty priradené vrcholom poľom B.



Obr. 1.39
Orientovaný graf

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} F & T & T & F \\ F & F & T & F \\ F & F & F & T \\ T & F & F & F \end{pmatrix} \end{array} \qquad B = \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{pmatrix} 12 \\ 16 \\ 36 \\ 72 \end{pmatrix}$$

Implementácia grafu pomocou zoznamu druhým spôsobom. Každý vrchol grafu je reprezentovaný prvkom zoznamu obsahujúcim smerník na informáciu spojenú s vrcholom grafu a smerník na zoznam zodpovedajúci hranám vychádzajúcim z daného vrchola. Graf z obr. 1.39 možno potom reprezentovať zoznamom, ako dokumentuje obr. 1.40. Vidíme, že takúto reprezentáciu grafu zoznamom možno interpretovať aj takto: každý vrchol u grafu je reprezentovaný lineárnym zoznamom, ktorého hlavička obsahuje informáciu spojenú s vrcholom a zvyšok zoznamu je zoznam vrcholov v takých, že hrana (u,v) patrí do H.



Obr. 1.40
Graf reprezentovaný zoznamom

Implementácia operácií typu údajov graf si vyžaduje sprístupnenie vrcholov, ktoré je spojené s určitou formou prechádzania grafom alebo prehľadávania grafu. Aj keď je viac možností prehľadávania grafu, najčastejšie sa používajú dve:

a. prehľadávanie do šírky, paralelné prehľadávanie,
b. prehľadávanie do hĺbky.

Pri prehľadávaní do šírky prezrieme začiatočný vrchol. V ďalšom kroku všetkých priamych nasledovníkov, v ďalšom všetkých ich nasledovníkov atď. Pri prehľadávaní do hĺbky postupujeme po jednej hrane resp. vrchole. Najskôr prezrieme začiatočný vrchol. Potom zistíme, či má nasledovníkov, ktorých sme ešte neprezreli. Ak áno, vyberieme jedného a celý proces opakujeme na tomto vrchole. Ak nie, zistíme, či má predchodcu. Ak áno, celý proces opakujeme na tomto vrchole. Ak nie, STOP, graf je prezretý.

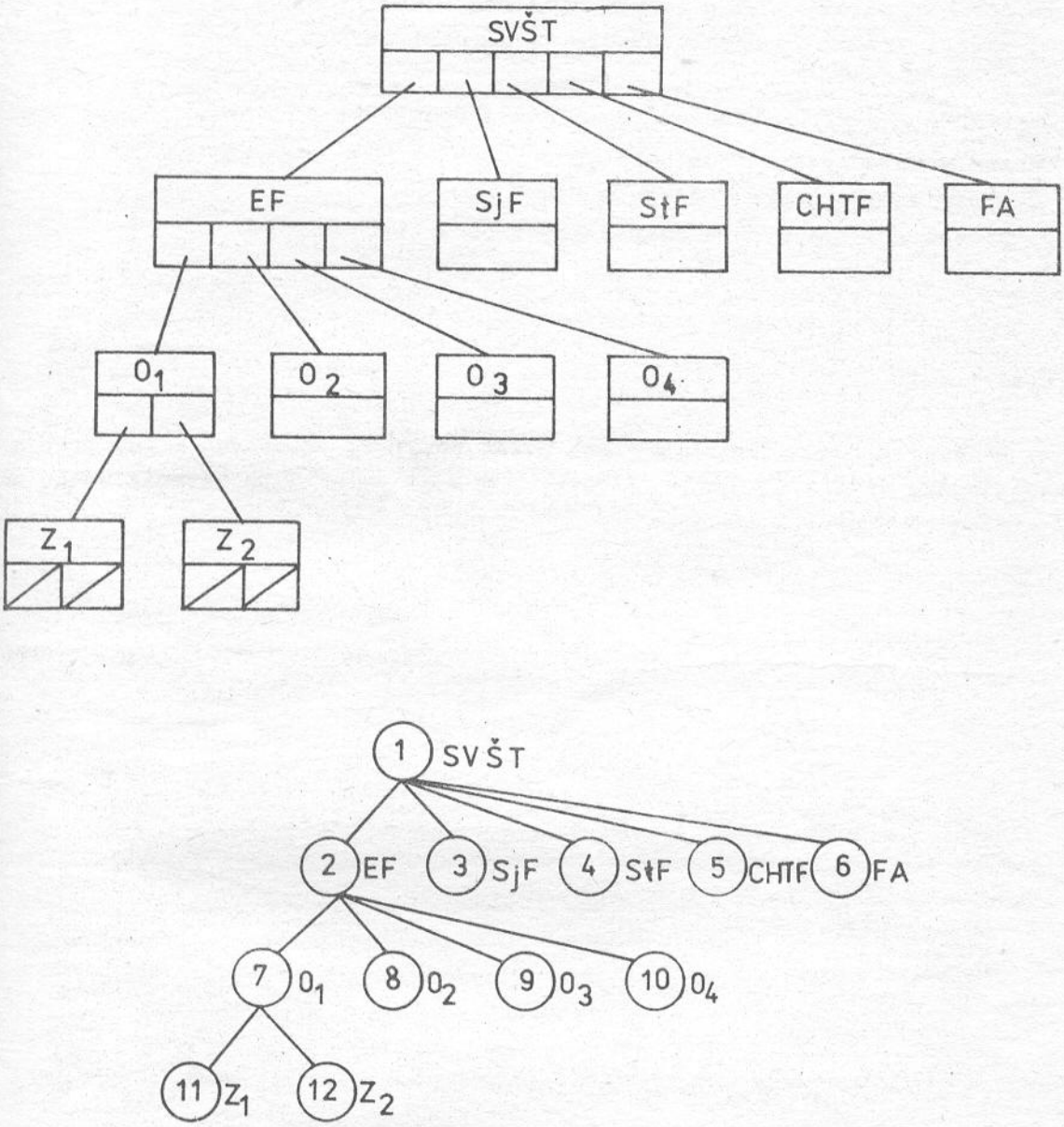
Pri grafe z obr. 1.39 a za predpokladu, že začiatočný vrchol je v_1, bude prezeranie prebiehať takto:

Prehľadávanie do šírky	Prehľadávanie do hĺbky
v_1	v_1
v_2, v_3	v_2
$(v_3), v_4$	v_3
(v_1)	v_4
	(v_1)
	(v_3)

1.4.9 Implementácia stromu

Strom, ako špeciálny prípad grafu, možno implementovať rovnakými prostriedkami ako graf. Okrem toho však využívame špecifické vlastnosti stromu. Strom možno teda implementovať pomocou incidenčnej matice, no reprezentácia incidenčnou maticou je neefektívna, pretože matica má väčšinu prvkov prázdnych. Najprirodzenejšia implementácia stromov je pomocou zoznamov, pričom každý prvok zoznamu obsahuje informáciu spojenú s vrcholom, a taký počet smerníkov, koľko má daný vrchol nasledovníkov. Reprezentácia stromu zoznamom je na obr. 1.41.

Osobitné miesto majú binárne stromy. Majú široké použitie a jednoduchú implementáciu. V prípade implementácie zoznamom, má prvok zoznamu okrem informácie spojenej s vrcholom dva smerníky na nasledovníkov daného vrchola.



Obr. 1.41
Reprezentácia stromu zoznamom

Na prehľadávanie binárnych stromov možno použiť metódy prehľadávania, ktoré sme si uviedli pre grafy. Metóda založená na prezeraní jedného vrcholu, resp. prechode po jednej hrane, sa pri stromoch používa v troch rôznych modifikáciách:

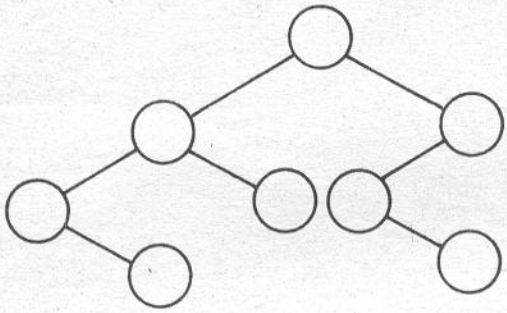
a. priame prehľadávanie (preorder):
   1. prezri koreň stromu,
   2. prezri priamym prehľadávaním ľavý podstrom,
   3. prezri priamym prehľadávaním pravý podstrom.

b. Vnútorné prehľadávanie - na prechode (inorder):
   1. prezri vnútorným prehľadávaním ľavý podstrom,
   2. prezri koreň,
   3. prezri vnútorným prehľadávaním pravý podstrom.

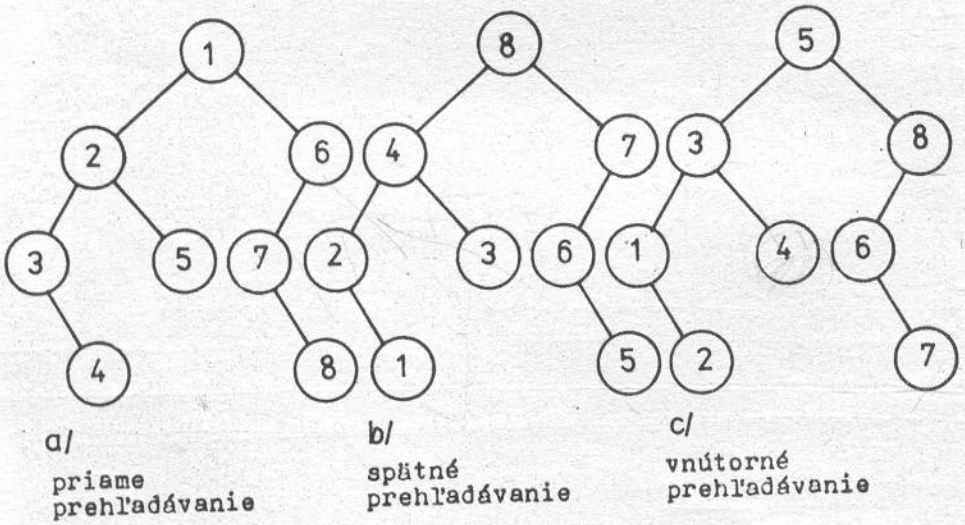
c. Spätné prehľadávanie (postorder):
   1. prezri spätným prehľadávaním ľavý podstrom,
   2. prezri spätným prehľadávaním pravý podstrom,
   3. prezri koreň.

Príklad 1.4.4.

Majme binárny strom podľa obr. 1.42. Potom použitie uvedených metód prehľadávania možno vidieť na obr. 1.43 a,b,c, kde je poradie prezerania udané celým číslom vo vrchole.



Obr. 1.42
Binárny strom



Obr. 1.43
Ohodnotenie stromu priamym, spätným
a vnútorným prehľadávaním

Procedúry prehľadávania bunárneho stromu možno implementovať týmto spôsobom:

```
type smer = ↑vrchol;
     vrchol = record
                    info  : alfa;
                    lnasl : smer;
                    pnasl : smer;
              end

procedure preorder(s : smer);
          begin if s < > nil then
                begin write (s↑.info);
                      preorder (s↑,lnasl);
                      preorder (s↑.pnasl);
                end
          end;

procedure inorder (s : smer);
          begin if p < > nil then
                begin inorder (s↑.lnasl);
                      write (s↑.info);
                      inorder (s↑.pnasl)
                end
          end

procedure postorder (s : smer);
          begin if p < >nil then
                begin postorder (s↑.lnasl);
                      postorder (s↑.pnasl);
                      write (s↑.info)
                end
          end
```

V praxi sa často stretávame so stromami, ktoré sa používajú na "špecifické" úlohy, čomu sú prispôsobené aj operácie. V tejto súvislosti si treba uvedomiť, že operácie pridávania a rušenia vrcholov resp. hrán závisia od problému, ktorý daný strom reprezentuje.

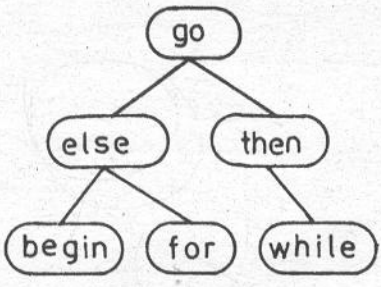
Špeciálnym stromom je aj strom binárneho hľadania, ktorý používame na reprezentáciu množiny. Strom binárneho hľadania pre množinu M nazývame vrcholovo ohodnotený binárny strom, ktorého každý vrchol v je ohodnotený prvkom h(v) z M, pričom:

1. h(u) < h(v) pre každý vrchol u z ľavého podstromu stromu s koreňom v,

2. $h(u) > h(v)$ pre každý vrchol u z pravého podstromu stromu s koreňom v,
3. pre každé $m \in M$ existuje práve jeden vrchol v taký, že $h(v) = m$.

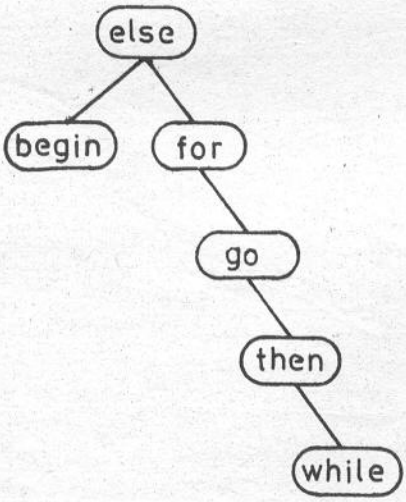
Príklad 1.4.5

Ak predpokladáme lexikografické usporiadanie, potom množinu M = {begin, else, for, go, then, while} možno reprezentovať týmto stromom binárneho hľadania (Obr. 1.44).



Obr. 1.44
Strom binárneho hľadania pre M

Už samotný názov stromu binárneho hľadania naznačuje, že je konštruovaný tak, že prehľadávanie takéhoto stromu zodpovedá binárnemu vyhľadávaniu. Pravda, počet porovnaní teraz závisí od štruktúry zodpovedajúceho stromu binárneho hľadania, teda od jeho konštrukcie. K množine M možno vytvoriť aj strom binárneho hľadania podľa obr. 1.45.



Obr. 1.45
Strom binárneho hľadania pre M

Príslušnosť vrchola s ohodnotením h(x) danému stromu, teda aj príslušnosť h(x) množine M, možno zistiť nasledujúcou procedúrou (funkciou), pričom organizácia vrcholu bude rovnaká ako pri procedure inorder.

```
function isin(s : smer; v : alfa): boolean;
        /* Test, či vrchol v, je v strome s */
        begin
              if v = s↑.info then isin := true
                             else
              if v < s↑.info then
                 begin if s↑.lnasl< >nil then isin(s↑.lnasl,v)
                                         else isin := false
                 end
                             else
                    if s↑.pnasl< >nil then isin(s↑.pnasl,v)
                                         else isin := false
        end
```

Pridávanie prvku do stromu binárneho hľadania možno realizovať nasledujúcou procedúrou insert.

```
procedure insert(var s : smer; v : alfa);
        /* Vkladanie vrchola v do stromu s */
        var t : smer;
        begin
              if s = nil then begin new(t);
                                    t↑.info := alfa;
                                    t↑.lnasl := nil;
                                    t↑.pnasl := nil;
                                    s := t
                              end
                         else
                    if v < s↑.info then
                              begin
                    if s↑.lnasl = nil then begin new(t);
                                                 t↑.info := v;
                                                 t↑.lnasl := nil;
                                                 t↑.pnasl := nil;
                                                 s↑.lnasl := t
                                           end
                                      else insert(s↑.lnasl,v)
                              end
        end
```

Zo spôsobu prehľadávania stromu binárneho hľadania vidíme, že prehľadávanie nebude dlhšie - väčší počet porovnaní, ako je výška stromu plus jedna. Pre-

to je výhodné, ak je strom konštruovaný tak, aby nebola jeho výška veľká. Ide najmä o také stromy, ktoré sú v istom zmysle "degenerované" - majú jednu dlhú cestu a zvyšok podstatne kratších ciest. Takéto stromy je vhodné upraviť do určitého "vyváženého" tvaru. Príkladom "vyvážených" stromov sú tzv. AVL stromy. AVL strom (po Adelsonovi-Veľskom a Landisovi) je taký strom binárneho hľadania, pre ktorého každý vrchol platí, že výšky jeho pravého a ľavého podstromu sa líšia najviac o 1. AVL stromom je napr. strom na obr. 1.44.

Až doteraz sme sa zaoberali implementáciou stromu zoznamom. Strom však možno implementovať aj vektorom. Binárny strom na vektor zobrazujeme takto:

1. Koreň stromu sa zobrazí na V[1],
2. Ak určitému vrcholu zodpovedá index i, potom jeho ľavý nasledovník sa zobrazí na V[2i], pravý nasledovník na V[2i+1].

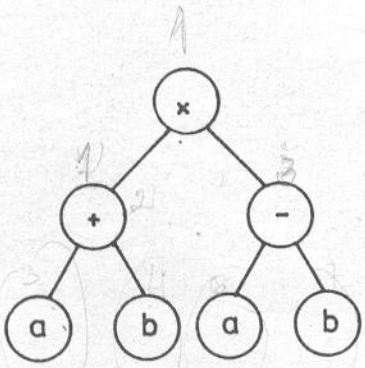
Pre binárny strom hĺbky h musíme vyhradiť vektor s dĺžkou $2^{h+1}-1$. Pri "degenerovaných" stromoch to môže mať veľmi nepriaznivý dopad na využitie pamäti, ako to vidíme z nasledujúceho príkladu.

Príklad 1.4.5

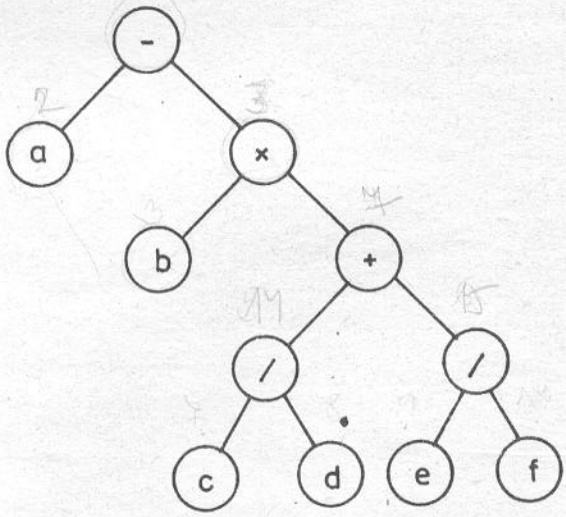
Majme dva aritmetické výrazy:

1. (a+b) x (a-b)
2. a-b x (c/d+e/f)

Možno ich reprezentovať stromami podľa obr. 1.46 a,b, ktoré možno reprezentovať vektorom podľa obr. 1.47 a,b.



Obr. 1.46a
Reprezentácia výrazu (a+b) x (a-b) stromom



Obr. 1.46b
Reprezentácia výrazu a-b x (c/d+e/f) stromom

a/

1	2	3	4	5	6	7
×	+	-	a	b	a	b

b/

1	2	3	4	5	6	7		14	15		28	29	30	31
-	a	×	b		b	+	...	/	/	...	c	d	e	f

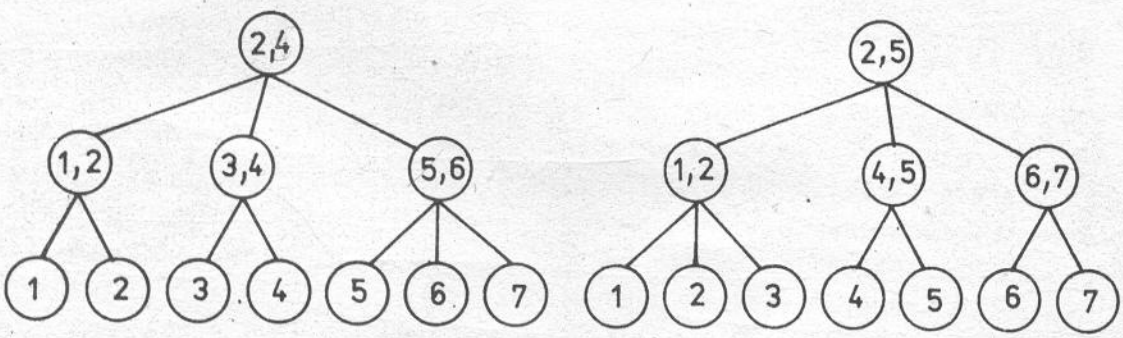
Obr. 1.47
Reprezentácia stromu vektorom

Ďalším špeciálnym stromom je 2-3 strom. 2-3 strom je taký strom, pre ktorý platí:

1. každý vnútorný vrchol má 2 alebo 3 priamych nasledovníkov,
2. všetky koncové vrcholy sú rovnako vzdialené od koreňa stromu,
3. koncovým vrcholom sú zľava doprava priradené všetky prvky množiny, a to v stúpajúcom poradí,
4. každému vnútornému vrcholu v je priradená dvojica hodnôt L(v), M(v), kde L(v) je najväčšia z hodnôt priradených koncovým prvkom podstromu s koreňom, ktorým je ľavý, (prvý zľava) priamy nasledovník vrchola v, M(v) je najväčšia z hodnôt priradených koncovým prvkom podstromu s koreňom, ktorým je druhý zľava priamy nasledovník vrchola v.

Príklad 1.4.6

Majme množinu $M = \{1,2,\ldots,7\}$. Túto množinu možno reprezentovať 2-3 stromami podľa obr. 1.48 a,b



Obr. 1.48
Reprezentácia množiny 2-3 stromom

Ukážeme si implementáciu niektorých operácií:

```
type smer = ↑vrchol;
     vrchol = record
                    L : alfa;
                    M : alfa;
                    prvý : smer;
                    druhý : smer;
                    tretí : smer;
              end;

function list(s : smer) : boolean;
          /* Test, či sa jedná o koncový vrchol */
          begin list := s↑.prvý = nil and s↑.druhý = nil
                                      and s↑.tretí = nil end

function isin(s : smer; v : alfa) : boolean;
          /* Test, či vrchol v je v strome s */
          begin if list(s) then isin := s↑.L = v
                           else
                if v <= s↑.L then isin (s↑.prvý,v)
                             else
                if v <= s↑.M then isin (s↑.druhý,v)
                             else
                if s↑.tretí <> nil then isin (s↑.tretí,v)
                                   else isin (s↑.druhý,v)
          end
```

Vlastnú funkciu isin by sme mohli zefektívniť, pretože končí prehľadávanie až na úrovni koncových vrcholov. V prípade, že v niektorom vrchole u je v > M(u) a u má iba dvoch priamych nasledovníkov, vieme, že v do danej množiny nepatrí a proces prehľadávania možno ukončiť. Prehľadávanie až na úroveň koncových vrcholov možno využiť pri operáciách vkladania a rušenia prvkov.

Pri implementácii operácii vkladania prvku insert nájdeme najskôr funkciou isin miesto, kde treba koncový prvok s danou hodnotou vložiť. Ak má príslušný predchodca iba dvoch priamych nasledovníkov, pridáme tretieho. Ak má troch, "rozdelíme" ho na dva vrcholy a rekurzívne upravíme problém vkladania prvku o úroveň vyššie. Rekurzia sa zastaví pred koreňom stromu, alebo treba rozdeliť aj koreň a predáme koreň nový.

Pri implementácii operácie rušenia vrchola-delete nájdeme najskôr miesto - koncový vrchol, ktorý treba zo stromu vylúčiť. Zrušenie vrchola, napr. v, robíme z hľadiska jeho predchodcu, napr. u, takto: zrušíme v. Ak u má teraz

dvoch nasledovníkov, skončíme. Ak u má iba jedného nasledovníka, zistíme, či ho nemôže "adoptovať" niektorý vrchol na tej istej úrovni ako u. Ak niektorý z vrcholov na úrovni vrchola u má dvoch nasledovníkov, pridáme uvažovaný vrchol k nemu a u rekurzívne odstránime. Ak všetky vrcholy na úrovni vrchola u majú troch nasledovníkov, potom u adoptuje jedného z ich nasledovníkov a končíme. Ak sa rekurzia končí v koreni stromu, odstránime ho a strom sa "zníži" o jednu úroveň.

1.4.10 Implementácia množiny

Najjednoduchšou implementáciou množiny je priama implementácia pomocou vektora. Takáto implementácia je však málo efektívna. V praxi volíme implementáciu podľa toho, ktoré operácie chceme implementovať najefektívnejšie.

Ak je počet prvkov množiny malý, možno množinu veľmi výhodne implementovať pomocou vektora, ku ktorému pridávame tzv. charakteristický vektor. Charakteristický vektor je logický a umožňuje implementovať množinové operácie priamo technickými prostriedkami počítača.

Predpokladajme, že máme množinu $M = \{m_i \ / \ 1 \leq i \leq n\}$. Na jej implementáciu použijeme vektor V[1..n] s charakteristickým vektorom B[1..n]. Prvky množiny M budeme reprezentovať prvkami vektora V. Prítomnosť i-teho prvku v množine vyjadríme pomocou vektora B tak, že B[i] nastavíme na 1, počas jeho neprítomnosti na 0. Test prítomnosti prvku v množine potom nerealizujeme prehľadávaním, ale priamo testom prvku vektora B. Tento spôsob implementácie množiny používame zvyčajne vtedy, keď počet prvkov množiny nepresahuje dĺžku slova príslušného počítača, vzhľadom na ktorý sa implementácia robí.

Množinové operácie sa potom dajú realizovať priamo inštrukciami pre logické operácie daného počítača. Operácie na jednej množine sa implementujú pomocou posuvov.

Pre množiny s veľkou kardinalitou používame iné spôsoby implementácie. Vhodným implementujúcim typom údajov je strom binárneho hľadania alebo 2-3 strom. Vlastnosti a prácu s nimi sme si prebrali v predchádzajúcej časti.

Ďalším implementujúcim typom údajov môže byť tabuľka. Prvok množiny sa reprezentuje prvkom tabuľky, pričom kľúčom je celý prvok.

2. Organizácia údajov na vonkajších pamätiach

Použitie vonkajších pamätí vo výpočtových systémoch v súčasnosti má viac dôvodov. K pôvodnému, ktorým bolo a zostáva rozširovanie kapacity pamäti, pribudli postupne s nástupom a zdokonalovaním operačných systémov ďalšie. Od zlepšenia a z efektívnenia organizácie prác počítača, efektívneho využívania procesora, cez ochranu samotného operačného systému, archiváciu údajov, prenos veľkých súborov údajov /z jedného výpočtového systému na iný, inštalovaný napríklad na druhom konci mesta/, tvorbu informačných systémov atď. V každom prípade vonkajšie pamäti podstatne rozširujú schopnosti a možnosti každého výpočtového systému.

2.1 ZÁKLADNÉ POJMY

Základným a jediným údajovým typom na vonkajších pamätiach je typ súbor. Je to množina n-tíc, nazývaných záznamy alebo vety súboru. Z hľadiska V/V operácií so súbormi je záznam nedeliteľnou jednotkou. To znamená, že môže sa prečítať len celý záznam, podobne sa môže zapísať len celý záznam. Z hľadiska logického spracovania sa záznam skladá z položiek, nazývaných tiež polia záznamu, resp. vety. Ak by sme mali napríklad súbor zamestnancov nejakého závodu, všetky údaje, týkajúce sa jedného zamestnanca, tvoria záznam a jeho zložkami sú jednotlivé údaje - napríklad dátum narodenia, pracovné zaradenie, stav atď.

Záznamy v jednom súbore môžu mať rovnakú dĺžku, vtedy hovoríme o záznamoch s pevnou dĺžkou, alebo môžu mať navzájom rôznu dĺžku, vtedy sú to záznamy s premenlivou dĺžkou.

Organizácia súborov veľmi úzko súvisí s organizáciou prídavných zariadení a spolu tvoria harmonický celok. Organizácia súborov tvorí v tejto hierarchii nadstavbu a organizácia prídavných zariadení jej základňu. Štandardné súbory majú v každom systéme predpísanú štruktúru. Vo všeobecnosti sa prístup k jednotlivým záznamom dá chápať ako vyhľadávanie položiek v tabuľke. Tento prístup je úplne závislý od charakteru použitého pamäťového média. To v konečnom dôsledku určuje, či sa môžu vytvárať určité hierarchie, alebo je možný len sekvenčný prístup. Moderné operačné systémy obsahujú už programy na spracovanie a ovládanie súborov, ktoré zohľadňujú všetky podrobnosti. Možno povedať, že je to samostatná oblasť z hľadiska operačných systémov, štruk-

túr údajov a programovacích techník. Práca so súbormi si okrem teoretických vedomosti vyžaduje aj určité praktické skúsenosti a poznatky. Keď sa povie súbor, rozumie sa tým väčšinou štruktúra údajov s veľkým počtom údajov. Práve ten "veľký počet" môže niekedy prekryť určité efekty a dopady, ktoré pri práci so súbormi vznikajú.

2.1.1 Súbor ako údajový typ

Súbor môžeme chápať aj ako údajový typ. Z doterajších informácií vyplýva, že v množine druhov máme zatiaľ súbor a záznam. Vzhľadom na prácu so súborom bude ďalší druh Boolean a k identifikácii záznamu potrebujeme kľúč, podobne ako pri práci s tabuľkou. Máme teda definovanú:

Množinu druhov:

- SÚBOR
- ZÁZNAM
- BOOLEAN
- KĽÚČ

Množina operácii obsahuje tieto základné operácie:

- READ - čítanie záznamu
- WRITE - zápis záznamu
- EMPTY - vytvorenie prázdneho záznamu
- KEY - stanovenie identifikačného kľúča k danému záznamu

Podľa bližšej špecifikácie uvedených operácií a pridaním ďalších môžeme definovať v zásade dva principiálne odlišné typy súborov:

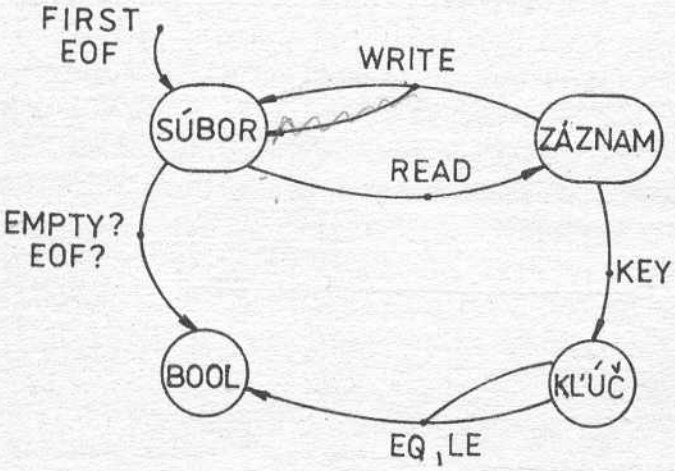
- sekvenčný súbor,
- súbor s priamym prístupom.

Nositeľom sekvenčného súboru je neadresovateľné médium /napr. magnetická páska/. Nositeľom súboru s priamym prístupom môže byť len adresovateľné médium /napr. magnetický disk/.

Pre údajový typ sekvenčný súbor doplníme množinu operácií týmito:

- FIRST - nájdenie prvého záznamu /pri čítaní/,
- EOF? - test na koniec súboru,
- EOF - definovanie konca súboru,
- EQ, LE - identifikácia daného kľúča.

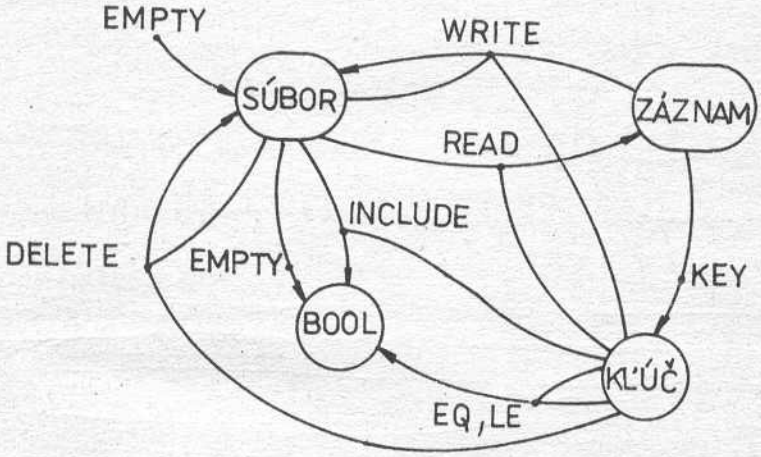
Signatúra:



Obr. 2.1
Signatúra údajového typu "sekvenčný súbor"

V prípade adresovateľného média sa špecifikácia údajového typu súbor podstatne zmení. Máme možnosť priameho prístupu k záznamom. Niekedy sa hovorí tiež o náhodnom výbere záznamu v takomto súbore. Takýto súbor môžeme chápať ako tabuľku.

Signatúra:



Obr. 2.2
Signatúra údajového typu "súbor s priamym výberom"

2.1.2 Fyzický a logický záznam

Vieme už, že pod súborom rozumieme množinu údajov, ktoré sú v istom zmysle kompletné k určitému spracovaniu. Tvorí teda logický celok, pozostávajúci zo záznamov, tiež logických celkov. Vzhľadom na médium, ktoré je nositeľom súboru, na štruktúru a spôsob prenosu určitých fyzických jednotiek hovoríme o fyzickom členení súboru na určité jednotky súboru. Všeobecne sa pre tieto fyzické jednotky súboru zaviedol názov blok. Ako rozlišujeme fyzický a logický záznam /blok/, rozlišujeme aj fyzické a logické čítanie a zápis záznamu. Logické čítanie a zápis realizujú operácie: READ a WRITE

READ - poskytne hodnotu záznamu v súbore,
WRITE - zapíše záznam do súboru.

Pod pojmom fyzické čítanie, resp. zápis, rozumieme fyzický prenos jedného bloku z vonkajšieho média do vyrovnávacej pamäti, resp. opačne. Logické čítanie/zápis sa realizuje pomocou fyzického čítania/zápisu.

Vzťah medzi logickým a fyzickým záznamom je v tejto tabuľke:

Médium	Fyzický záznam	Logický záznam
DŠ	1 DŠ	DŠ = záznam
MP	blok	blok = n záznamov
MD	blok	blok = n záznamov
LP	1 riadok	1 riadok = záznam

Obr. 2.3
Fyzické a logické členenie súborov (n≥1)

Súbor môže okrem záznamov, týkajúcich sa samotného spracovania úlohy, obsahovať niektoré doplňujúce údaje. Tieto slúžia na kontrolu totožnosti súboru a správnosti záznamov. Ukladajú sa do tzv. návestia súboru. Tieto môžu byť štandardné, ktoré ovláda a kontroluje systém pre ovládanie súborov. Štandardné návestia súborov sa vytvárajú a kontrolujú automaticky, nezávisle od užívateľa. Okrem štandardných návestí môžu byť aj návestia užívateľské, ktoré si užívateľ vytvára a ošetruje sám. Tieto slúžia väčšinou na špeciálnu ochranu pred čítaním nepovolanou osobou a podobne.

Podľa umiestnenia na médiu sa rozlišujú počiatočné a koncové návestia súborov.

Počiatočné návestia súboru:

Umiestňujú sa na začiatok súboru a obsahujú informácie pre identifikáciu súboru. Sú to:

- meno súboru,
- evidenčné číslo súboru,
- poradové číslo súboru,
- dátum vytvorenia,
- dátum skončenia platnosti súboru, atď.

Koncové návestia súboru:

Umiestňuje sa na koniec súboru. Obsahuje informácie, potrebné ku kontrole správnosti vytvoreného súboru. Napríklad:

- počet blokov v súbore,
- opakované údaje zo začiatku, atď.

Na rôznych pamäťových médiach sú rôzne zoskupenia informácií o súboroch na nich. Na magnetickej páske sú okrem počiatočného a koncového návestia súboru, počiatočné a koncové návestia cievky. Podobne sú na diskoch okrem návestia súborov ešte návestia resp. hlavičky zväzkov.

2.2 IMPLEMENTÁCIA SÚBOROV

Pri definovaní údajového typu súbor sme rozlíšili dva základné typy súborov - sekvenčný a
- s priamym prístupom.

Toto rozdelenie je dané typom pamäťových médií, a to ich základnou vlastnosťou, či sú alebo nie sú adresovateľné. Na adresovateľných médiách sa dajú implementovať obidva typy súborov. Takýmito médiami sú: magnetická páska, dierne štítky a tlačiareň sú neadresovateľné a dajú sa na nich vytvárať len sekvenčné súbory.

2.2.1 Sekvenčný súbor na neadresovateľných médiach

Ako abstraktný model sekvenčného súboru môžeme použiť front so svojimi základnými operáciami. Po spracovaní jedného záznamu sa sprístupní jej fyzicky najbližší sused - nasledovník.

Pri implementácii sekvenčných súborov je väčšina operácií realizovaná pomocou základných operácií pre vstup a výstup. Spravidla sa tu rozdeľujú súbory na vstupné a výstupné. Ich sémantika je:

Vstupný súbor:
- FIRST - nastavenie súboru na začiatok,
- READ - čítanie aktuálneho záznamu,
- EOF? - test, či je prečítaný záznam FM - značka konca súboru,
- EMPTY? - test, či po nastavení súboru na začiatok je prvým prečítaným znakom FM.

Výstupný súbor:
- FIRST - nastavenie súboru na začiatok (pri magnetických páskach - previnutie pásky na začiatok),
- WRITE - zápis záznamu,
- EOF - zápis FM.

Operácie sa implementujú pomocou operácií V/V. Musia byť ďalej presne špecifikovaní reprezentanti, resp. nosiči druhov, periférne zariadenie, na ktorom bude súbor, dĺžka bloku, dĺžka záznamu a iné.

Pri pridávaní záznamu do súboru, môžeme tento doplniť len vtedy, ak je tento súbor posledný na magnetickej páske a je za ním ešte fyzicky voľné miesto. V takom prípade sa vymaže ukončovacia značka súboru (TM - tape mark), doplní sa záznam a za ním sa zapíše TM. Ak nie je voľné miesto, musí sa vytvoriť nový súbor, ako je to pri ukladaní nových záznamov do už predtým vytvoreného súboru.

2.2.2 Súbory na magnetických diskoch

Pre správne pochopenie implementácie súborov na magnetickom disku je dôležité aspoň približne poznať spôsob záznamu. Dôležitým faktorom je usporiadanie média. Základ tvorí disk. Disky sú organizované do zväzkov. Počet diskov vo zväzku je rôzny (bežne 6). Samotný disk je rozdelený na stopy, stopy na sektory. Stopy s rovnakým polomerom tvoria valec.

Najmenšou adresovateľnou jednotkou na disku je sektor. Adresu sektoru tvorí:

- číslo diskovej jednotky, ak ich je viac,
- číslo valca,
- číslo plochy,
- číslo sektoru.

Niekedy sa hovorí o indexe (valca, plochy atď. - preto indexovosekvenčné súbory). Vybavovacia doba ľubovoľnej imformácie na disku je daná dobou, potrebnou pre nastavenie hlavičky do príslušnej stopy a časom, ktorý je potrebný pre natočenie príslušného sektoru.

Vzhľadom na uvedený spôsob prístupu na disk je organizácia súborov na magnetických diskoch veľmi rozmanitá. Od čisto sekvenčných súborov až po súbory s adresovanými záznamami.

2.2.2.1 Sekvenčný diskový súbor

Sú to súbory analogické so súbormi na magnetickej páske alebo diernych štítkoch. Takéto súbory sa ľahko vytvárajú, ale ťažko sa s nimi pracuje. Pri aktualizácii tohto súboru sa zvyčajne vytvorí z nových požiadaviek dávka. Z nej a pôvodného súboru sa generuje nový súbor. Nové záznamy sa zaraďujú tak, aby záznamy vo vzniknutom súbore boli v správnom poradí a fyzicky susedné.

2.2.2.2 Sekvenčné zreťazené súbory

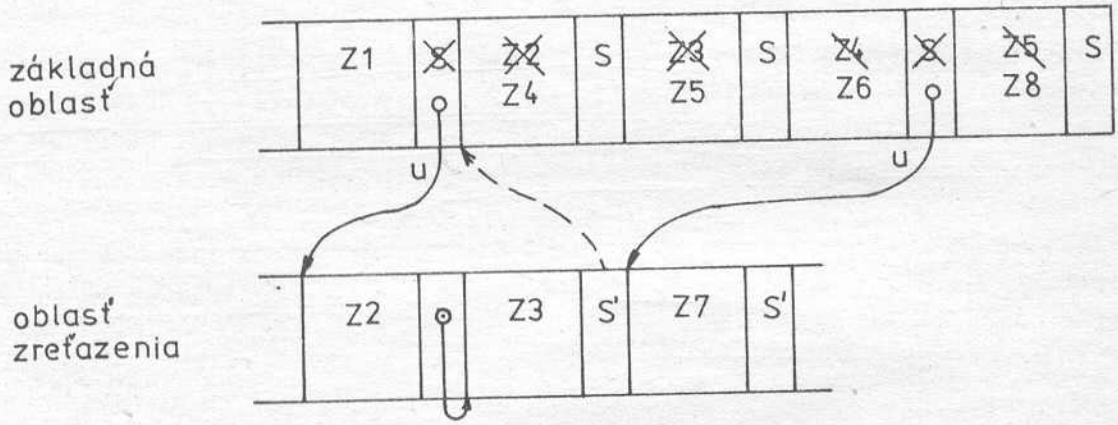
Podobne ako sa v hlavnej pamäti realizujú zoznamy zreťazených prvkov pomocou odkazov, možno vytvárať zreťazené súbory na vonkajších pamätiach. V tomto prípade nemusia byť logicky susedné záznamy (patriace do toho istého súboru) fyzicky susedné na disku. Sekvenčné zreťazené súbory sú kombináciou sekvenčných a zreťazených súborov.

Zreťazenie sa používa predovšetkým pri vkladaní alebo výbere záznamu, prípadne jeho výmene, ak si nezodpovedajú dĺžkou vymieňané záznamy. Pokiaľ je možné, t.j. ak logické poradie záznamov súhlasí s fyzickým poradím, používa sa sekvenčná technika. Pre techniku zreťazovania musí mať takýto kombinovaný súbor k dispozícii pamäťový priestor, skladajúci sa z dvoch oblastí:

základná oblasť: tu sa zaraďujú záznamy sekvenčným spôsobom (pri vytváraní súboru),

oblasť zreťazenia: tu sa zaraďujú záznamy pri aktualizácii, pridávaní záznamov do súboru.

Každý záznam v takomto súbore obsahuje položku, obsahujúcu príznak toho, či je logicky nasledujúci záznam fyzicky susedný (S) alebo ukazovateľ (U), ak nie je fyzicky susedný. Aktualizácia sekvenčne zreťazeného súboru je na obr. 2.4. Príznak alebo ukazovateľ sú uložené na konci záznamu (spravidla).



Obr. 2.4
Sekvenčne zreťazený súbor

Vidíme, že pri aktualizácii súboru nemusíme generovať nový súbor, ako pri čisto sekvenčných súboroch. Príznak S´ v oblasti zreťazenia je vlastne tiež ukazovateľ, ale naznačuje prechod do základnej oblasti so sekvenčným spracovaním.

Doba prístupu pri týchto súboroch sa v porovnaní so sekvenčným súborom predĺži, pretože čítacia hlavička sa musí presúvať na stopy zreťazenia a späť. Aby sa novými aktualizáciami táto doba nepredlžovala, po určitom čase je dobré vygenerovať nový súbor, a tým vyprázdniť oblasť zreťazenia.

2.2.2.3 Adresované súbory

Tieto súbory sú analógiou tabuliek s priamym prístupom. Rozdiel medzi už uvedenými typmi súborov a adresovanými súbormi je v spôsobe identifikácie záznamu v súbore. Pre zadaný rozsah identifikačných kľúčov Ik a zodpovedajúci rozsah adresných kľúčov Ak treba zostrojiť transformačnú funkciu f

$$f(Ik) = Ak \tag{2.1}$$

taká, aby platilo:

$$\text{pre } Ik_1 \neq Ik_2 \Longleftrightarrow f(Ik_1) \neq f(Ik_2)$$

Musíme si pritom uvedomiť, že adresu na disku vytvára určitá hierarchia čísiel (indexov). Rovnako ako pri tabuľkách, aj tu vzniknú problémy, spôsobe-

né možným veľkým rozsahom identifikačných kľúčov a nemôžeme si pritom dovoliť jednoznačné priradenie na množinu adresných kľúčov. Synonymá v tomto prípade ukladáme zreťazené do oblasti zreťazenia.

2.2.2.4 Indexované súbory

Pri implementácii indexovaných súborov, podobne ako pri adresovaných, je potrebné akceptovať skutočnosť, že disk, na ktorom sa takýto súbor implementuje, je priamo prístupné médium len v prvom priblížení. Doba prístupu je súhrn viacerých dôb - obrátky disku a presuny hlavičky (resp. hlavičiek). Abstraktným modelom uvedeného typu súboru je aj tabuľka. Identifikácia príslušného záznamu sa realizuje v takomto súbore cez tabuľku indexov, v ktorej ku každému identifikačnému kľúču Ik je priradený adresný kľúč Ak, t.j. adresa uloženia záznamu. Treba si uvedomiť rozdiel medzi predchádzajúcim typom súboru - adresovaným - a týmto. Pri adresovaných súboroch sa toto preradenie realizuje cez transformačnú funkciu, tu cez tabuľku. Môže sa to prejaviť ako určitá nevýhoda, pretože priradený pamäťový priestor pre súbor musí mať vyhradené miesto pre túto tabuľku - zvýši sa nárok na pamäť. Nie je to však veľmi podstatné, resp. nevýhodné. Problém aktualizácie takéhoto súboru potom prejde na problém údržby tabuľky (výber, zaradenie).

Pri čítaní záznamu z indexovaného súboru:

- musí sa nájsť v tabuľke indexov k zadnému Ik príslušný Ak.

Pri zápise záznamu do súboru:

- musí sa do tabuľky vložiť zodpovedajúci Ik a Ak - adresu uloženia záznamu.

V organizácii tabuliek indexov sa môžu uplatniť všetky techniky, ktoré sa používajú pri tabuľkách. Pri rozsiahlych súboroch sa môže identifikačný kľúč rozdeliť na viac častí

$$Ik = Ik^1 . Ik^2 . \dots . Ik^n \qquad (2.2)$$

a vytvoriť hierarchiu tabuliek s funkciami $f^1, f^2, \dots, f^n$ tak, aby platilo:

$$f^j(Ik_i^j) = Ak_i^j \qquad (2.3)$$

Ak_i^j je adresa tabuľky kľúčov, ktoré majú rovnaké časti Ik^k pre $k=1,2,\dots,j$. Je to podobné, ako pri implementácii viacrozmerných polí Iliffovými vektormi.

Pri indexovaných súboroch sa dá realizovať prístup aj podľa viacerých kľúčov, čo podstatne rozširuje možnosti takého súboru. Pre každý kľúč musí existovať zvláštna tabuľka indexov.

2.3 OVLÁDANIE SÚBOROV

Programovanie vstupu a výstupu údajov pomocou základných V/V operácií nie je jednoduché a ľahké. Vyžaduje si perfektné ovládanie strojového jazyka počítača. Pri implementácii súboru údajov pristupuje ešte samotná organizácia súborov. Preto sa implementácia súborov neponecháva užívateľom, ale zveruje sa skúseným systémovým programátorom. Väčšina počítačov má v základnom vybavení systém podprogramov pre ovládanie súborov (SOS), niekedy tiež systémom IOCS (Input Output Control System), čo je vlastne implementácia údajového typu: súbor s už uvedenou signatúrou.

Skôr, ako sme schopní vykonať prvú operáciu nad súborom, musíme mať tabuľku, ktorá obsahuje názov, dĺžku a začiatočnú adresu každého súboru. Táto tabuľka sa nazýva adresár súborov alebo hlavička zväzku. Ďalej každé pamäťové médium má svoje charakteristické ukončenia. Na magnetickej páske funkciu ukončenia môže vykonávať zónová značka, na disku fyzický záznam s nulovou dĺžkou poľa údajov. Okrem toho má každé médium fyzický koniec média - - zväzku. Koniec zväzku však nemusí znamenať koniec súboru. Väčší súbor môže zaberať viac zväzkov, alebo sa môže nachádzať vo viacerých zväzkoch. Preto sa spôsob organizácie súboru a stav súboru zaznamenáva v tzv. hlavičke súboru. Táto môže byť tesne pred telom súboru, prípadne na rovnakom zväzku, ale na inom mieste (ďalej od tela súboru), prípadne môže byť na inom zväzku.

2.3.1 Magneticko-páskové súbory

Vieme už, že jediná možná organizácia súborov na uvedenom médiu je sekvenčný súbor. Na jednej magnetickej páske môže byť jeden súbor, viac súborov alebo časť súboru, ktorý si vyžaduje viac magnetických pások (viac zväzkov).

Najjednoduchšia organizácia súborov na páske je len pomocou zónovej značky. Súbory nemajú hlavičky, ich ukončenie je vytvorené len zónovou značkou. Dvojica zónových značiek sa interpretuje všeobecne ako koniec zväzku. Ta-

káto organizácia súborov nie je vo väčšine prípadov postačujúca. Obvykle majú súbory hlavičky súboru a ukončenie býva tiež zložitejšie.

Najčastejšie je hlavičkou jeden alebo viac záznamov. Tieto tvoria potom tzv. systémové záznamy hlavičky, ktorých štruktúra je daná výrobcom. Okrem nich môže mať užívateľ v hlavičke súboru vlastné záznamy, tzv. používateľové záznamy. Prvé slúžia systémovým programátorom a podprogramom ako zdroj informácií pri práci so súborom a druhé používateľovi ako pomôcka. Ukončenie môže byť rovnako definované systémovými aj používateľovými znakmi. Existuje ešte špeciálny záznam patriaci na koniec pásky - koniec zväzku. Slúži na rozlíšenie konca súboru od konca zväzku, ak máme viaczväzkový súbor.

Okrem toho treba aj pásku identifikovať ako zväzok. Každý zväzok dostane svoje "archívne číslo", vo výpočtovom stredisku unikátne. Tento údaj, spolu s ďalšími (meno užívateľa), uchováva sa v prvom zázname na páske.

2.3.2 Diskové súbory

Magnetický disk ako médium s priamym prístupom dovoľuje organizovať súbory efektívnejšie ako páskové jednotky. Na jednom disku - zväzku je bežne viac súborov, z ktorých každý môže patriť inému používateľovi. Z tohto dôvodu, okrem "pohodlného" prístupu k súborom a ich záznamom, je dôležitá aj ochrana súborov pred úmyselným alebo neúmyselným poškodením.

Vieme už, že na páske je súbor uložený súvisle, t.j. hlavička súboru, telo súboru, ukončenie súboru. Disk s ohľadom na prístup už nevyžaduje takéto súvislé uloženie. Nesúvislosť môže prechádzať dokonca cez viaceré zväzky.

V stratégiach prideľovania pamäti procesu sa pracuje s pojmom oblasť. Budeme ním rozumieť súvislú oblasť pamäti. V hlavičke diskového súboru sú potom okrem iných aj tieto údaje o oblastiach súboru:

- veľkosť každej oblasti,
- zväzok, na ktorom sa oblasť nachádza,
- disková adresa začiatku oblasti.

Každý súbor má svoje meno, ktoré je reprezentované znakovým reťazcom určitej (danej, resp. obmedzenej) dĺžky. Toto musí byť unikátne, vzhľadom na to, že na jednom disku sa môže stretnúť viac používateľov. Ináč môžu nastať kolízie a ich pravdepodobnosť je dosť veľká. Jedna z možností, ako vylúčiť tieto kolízie, je hierarchizovanie súborov tak, že používateľ má jeden súbor na najvyššej úrovni a meno tohto súboru dostane direktívne pridelené. Položky tohto súboru sú zase súbory. Mená súborov musia byť rôzne len v rám-

ci jednej úrovne. Súbor je pridelený používateľovi a má charakter knižnice. Miesto adresára sa používa tzv. katalóg súborov. Je to vlastne hlavička súboru na najvyššej úrovni a aj súbor, ktorý obsahuje hlavičky podsúborov. Hierarchia sa môže ďalej rozvíjať. Súbor, ktorého hlavička je v katalógu, môže byť znova katalógom.

Uvedený spôsob ovládania a ochrany súborov nie je jediný. Ak sú súbory organizované do knižníc podľa používateľov, potom meno, resp. kód používateľa musí byť unikátny. V takomto prípade rovnaké meno súborov rôznych užívateľov nespôsobí kolíziu. Hierarchizácia súborov je podobná prvému prípadu.

Organizácia súborov je veľmi úzko prepojená s organizáciou V/V zariadení. Tieto problémy sú riešené v rámci operačných systémov a bežnému užívateľovi je zásah do tejto organizácie nemožný. Cieľom tejto kapitoly bolo nazrieť na súbory ako určité štruktúry údajov, určité usporiadanie, ktoré je podstatne podmienené samotným médiom a rovnako aj práca s nimi.

3. Triedenie a vyhľadávanie

V programovaní je veľmi často potrebné usporiadať postupnosť prvkov do rastúcej (resp. neklesajúcej) alebo klesajúcej (resp. nerastúcej) postupnosti, prípadne nájsť určitý prvok v tejto postupnosti. Proces, v ktorom prvky usporadúvame, nazýva sa triedenie. Proces, v ktorom hľadáme určitý prvok - - vyhľadávanie. Samotný výraz "triedenie" má viac významov v hovorovej reči, prípadne v rôznych odboroch, ale v programovaní sa zaužíval v uvedenom zmysle. Prvkom môže byť údaj rôzneho charakteru. Môže to byť číslo, reťazec, znak, veta v súbore alebo položka so svojou vlastnou štruktúrou.

Mnohé metódy pre prácu s tabuľkami, slovníkmi, súbormi, predpokladajú usporiadanosť postupnosti prvkov, nad ktorými sa má operovať. A to usporiadanosť počas celej práce.

Všeobecne operácie, definované a používané nad štruktúrovanými údajmi, môžeme rozdeliť na tri skupiny:

- V prvej skupine sú operácie, ktorými sa na základe vstupných údajov generuje nový stav štruktúry. Aplikáciou takejto operácie sa mení stav štruktúry.
- V druhej skupine sú operácie, ktoré na základe vstupnej informácie poskytnú samotnú položku, pričom sa stav štruktúry nemení. Položka sa len identifikuje, resp. sprístupní.
- V tretej skupine sú operácie - predikáty, ktoré definujú hodnotu true alebo false pravdivostného výroku o štruktúre napr. pri prehľadávaní. Aplikáciou operácie z tejto skupiny sa stav štruktúry nemení.

Pozorným sledovaním procesu triedenia a vyhľadávania prídeme k typickým operáciam. Je to:

- hľadanie položky,
- zaradenie novej položky,
- zrušenie (už) nepotrebnej položky,
- zmena obsahu, resp. hodnoty položky.

Každá z týchto operácií sa dá zaradiť do niektorej z uvedených skupín. Všetky vychádzajú z rovnakej úlohy - sprístupnenia, t.j. určenia alebo vyhľadania pamäťového miesta, kde sa nachádza, alebo kde bude uložená aktuálna položka.

3.1 TRIEDENIE

Triediacich algoritmov je veľa. Líšia sa výpočtovou a programovacou zložitosťou. Spoločnou vlastnosťou je schopnosť utriediť ľubovoľnú množinu prvkov. Vo väčšine prípadov platí, že čím je metóda jednoduchšia, čím jednoduchšie sa programuje, tým pomalšia je pri používaní. Preto je vhodnejšia pre malé počty prvkov alebo s určením pre zriedkavé používanie. Pri veľkých počtoch prvkov, prípadne často používanej metóde, je výhodné "investovať" čas a námahu na naprogramovanie zložitej, ale rýchlej metódy.

Triedenie sa tradične spája so spracovaním hromadných údajov. Môže vzniknúť názor, že sa mu zbytočne veľa venujeme. Vylepšujú sa staršie metódy, vznikajú nové a vôbec, venuje sa tomuto problému v ostatných rokoch veľa pozornosti. Jednou z podstatných motivácií je poznanie, že priemerne až 25 % strojového času sa spotrebúva, niekedy aj stráca, práve pri rôznom triedení údajov. V niektorých výpočtových systémoch dokonca viac ako polovica strojového času. Zo štatistiky, v súvislosti s predchádzajúcim tvrdením, dá sa urobiť takýto záver:

1. Triedenie sa musí často používať - vyžaduje si to okruh riešených úloh.
2. Používa sa aj vtedy, keď to nie je potrebné.
3. Používajú sa nevhodné, neefektívne metódy.

Okrem uvedenej motivácie sledovania triediacich algoritmov, ktorá súvisí s ekonomikou výpočtových systémov, sú zaujímavé aj z hľadiska teoretického ako samostatný objekt skúmania. Tieto algoritmy sú vhodný objekt na demonštrovanie vlastností algoritmov, analýzu algoritmov a umožňujú získavať rôzne pracovné charakteristiky algoritmov. Teda všetky prostriedky, ktoré slúžia k vyhodnoteniu, porovnaniu a na základe toho výberu vhodných metód a im zodpovedajúcich algoritmov.

3.1.1 Základné pojmy a charakteristiky

Zavedieme si základnú, v ďalšom texte používanú terminológiu.

Máme usporiadať n prvkov:

$$p_1, p_2, \ldots, p_n$$

Jednotlivé prvky p_i nazývame položkou a všetky N položky nazveme postupnosťou položiek. Každá položka p_i má kľúč K_i. Cez kľúč sa položka sprístupňu-

je. Kľúč tvorí časť informácie obsiahnutej v položke. Vo všeobecnosti sa každá položka skladá z kľúča a z hodnoty. Hodnotou môže byť napríklad aj ukazovateľ na samotný záznam na jeho hodnotu. Hodnota položky nemá vplyv na triedenie. Z bežnej praxe všetci poznáme napríklad telefónny zoznam. Je to v podstate množina položiek, kde je meno účastníka kľúčom a jeho telefónne číslo hodnotou položky. Alebo slovník, napríklad anglicko-slovenský a slovensko-anglický. V prvej časti je kľúčom anglický výraz a hodnotou jeho slovenský ekvivalent. V druhej časti je slovenský výraz kľúčom a anglický výraz hodnotou položky.

V úvode sme zaviedli pojem triedenie v zmysle usporiadania množiny prvkov do nejakej postupnosti. Vzťah "$\leq$" je definovaný v takomto zmysle:

Hodnoty troch rôznych kľúčov K_1, K_2, K_3 musia spĺňať podmienky, platné pre usporiadanú množinu.

Usporiadaná množina je dvojica $(X, \leq)$, kde X je množina a $\leq$ je binárna relácia s vlastnosťami

$$\begin{aligned} &1.\ x \leq x \quad \text{pre } x \in X \\ &2.\ x_1 \leq x_2 \text{ a } x_2 \leq x_1, \text{ potom } x_1 = x_2, \text{ pre } x_1, x_2 \in X \\ &3.\ x_1 \leq x_2 \text{ a } x_2 \leq x_3, \text{ potom } x_1 \leq x_3, \text{ pre } x_1, x_2, x_3 \in X \end{aligned} \qquad (3.1.1)$$

Tieto vlastnosti definujú matematické chápanie lineárneho usporiadania. Pochopiteľne, že rovnako sa dá definovať vzťah "$\geq$".

Vo všeobecnosti problém triedenia je nájdenie takého preusporiadania položiek $p_1, p_2, \ldots, p_n$, po ktorom budú kľúče v neklesajúcej postupnosti

$$K_{p_1} \leq K_{p_2} \leq \ldots \leq K_{p_n} \qquad (3.1.2)$$

Triedenie sa nazýva stabilné, ak pre položky p_i, p_j, ktorých kľúče sa rovnajú, platí:

$$p_i < p_j \quad \text{ak} \quad K_{p_i} = K_{p_j} \quad \text{pre } i < j \qquad (3.1.3)$$

Pri výbere metódy v konkrétnom prípade nás zaujíma odpoveď na otázku: "Ktorá metóda je najlepšia? Ktorú vybrať?" Jednoznačná odpoveď na tieto otázky neexistuje. Všeobecne najlepšia metóda neexistuje. Výber metódy je ovplyvnený viacerými faktormi. Ako podstatné môžeme vybrať:

1. Rozsah spracúvanej postupnosti (počet položiek - n).
2. Či ide o postupnosť, ktorá sa zmestí do operačnej pamäti, alebo bude spracúvaná na etapy.

3. Či budeme triediť danú množinu položiek len raz, prípadne veľmi zriedkavo, alebo to je súbor, ktorý sa bude permanentne aktualizovať, prípadne nami vybraný algoritmus sa bude často používať.

Každá metóda je charakterizovaná dvoma základnými veličinami, ktoré nám priblížia vlastnosti metódy a v konečnom dôsledku by mali ovplyvniť jej výber. Je to:

- počet prechodov cez celý blok údajov, ktorý je potrebný pre úplné utriedenie a
- počet porovnaní, ktoré sa musia vykonať od začiatku triedenia po úplné utriedenie.

Niekedy sa uvažuje ešte ďalšia veličina, ktorá je charakteristická pre určitú metódu, napr. počet výmien, vkladaní, t.j. počet akcií, ktoré menia poradie položiek a tvar celého bloku. Všetky tieto veličiny sú udávané vo väčšine metód ako dolné a horné odhady, prípadne aj stredné, závislé:

- od počtu všetkých triedených položiek N,
- od počiatočného stavu triedenej množiny,
- a niekedy aj od rozsahu kľúčov.

V staršej literatúre, podobne ako pri práci s tabuľkami, pri posudzovaní kvality metódy sa uvažuje T - priemerná dĺžka hľadania

$$T = \frac{1}{m} \sum_{i=1}^{m} S_i \qquad (3.1.4)$$

kde m je počet realizovaných prístupov,
S_i - doby prístupov k rôznym položkám (p_i),
môžu byť všeobecne rôzne, preto sa častejšie uvažuje parameter: A je priemerná dĺžka hľadania, za predpokladu rovnomerného prístupu k položkám a

$$A = \frac{1}{n} \sum_{i=1}^{n} S_i \qquad (3.1.5)$$

kde n je počet položiek.

Ako miera vyhodnotenia sa tiež používa tzv. výpočtová zložitosť algoritmov. Vyjadruje rast času, potrebného na realizáciu algoritmu, prípadne kapacitu použitej pamäti, v závislosti od rozsahu údajov.

Vstupné údaje algoritmu A sú charakterizované jedným číslom N, ktoré nazývame rozsah problému. Toto reprezentuje rozsah vstupných údajov. Môže to byť napr. počet cifier, čísiel, ktoré treba násobiť, počet záznamov, ktoré treba vytriediť, stupeň polynómu atď.

Potom časovou zložitosťou algoritmu A, vzhľadom na počítač M, nazývame funkciu:

$$T_A^M(N)$$

a pamäťovou zložitosťou algoritmu A nazývame funkciu (3.1.6)

$$P_A^M(N)$$

Zložitosť algoritmu je pojem relatívny vzhľadom na zvolený model počítača. Možno však získať vyhodnotenie výpočtovej zložitosti algoritmov, nezávisle od tohto modelu. Treba si uvedomiť, že cieľom analýz algoritmov vzhľadom na čas výpočtu (u $P_A^M(N)$ to býva zväčša jednoduchšie) je získať také výsledky, ktoré budú dostatočne charakterizovať realizáciu algoritmu na reálnych počítačoch. Tie sú však príliš komplikované. Preto sa tieto problémy študujú na jednoduchších modeloch počítačov, podobných reálnym počítačom. Takýmto vhodným modelom je napr. počítač RASP - počítač s ľubovoľným výberom a zapamätaným programom (random access stored program machine):

Predpokladajme, že chceme určiť časovú zložitosť algoritmu A. $T_A^M(N)$ závisí od počítača M. Vo všeobecnosti teda $T_A^M(N)$ nevieme určiť. Dá sa ale určiť taká funkcia g(N), že $T_A^M(N) = O(g(N))$ pre každý počítač typu RASP. Potom hovoríme, že algoritmus A má časovú zložitosť O(g(N)).

Definícia 3.1.1

$f(N) = O(g(N))$ vtedy a len vtedy, ak existujú kladné konštanty C a N_0 také, že $f(N) \leq C|g(N)|$ pre $N \geq N_0$.

To znamená, že keď algoritmus realizujeme na nejakom reálnom počítači pre vstupné údaje rozsahu N, potom celkový čas výpočtu bude menší ako $C|g(N)|$ pre nejaké C, ktoré závisí len od zvoleného počítača, t.j. $T_A^M(N)$ nerastie rádove rýchlejšie ako g(N). Pochopiteľne usilujeme sa určiť čo najmenšie g(N), aby platilo:

$$T_A(N) = O(g(N))$$

O je označenie, ktoré znamená horné ohraničenie časovej alebo pamäťovej zložitosti algoritmov. Dolné ohraničenie sa označuje ako Ω a je definované takto:

Definícia 3.1.2

$f(N) = \Omega(g(N))$ vtedy a len vtedy, ak existujú kladné konštanty C a N_0 také, že $|f(N)| \geq C|g(N)|$ pre $N \geq N_0$.

Okrem uvedených ohraničení pre zložitosti je zavedená tzv. asymtotická zložitosť (rádová), s označením Θ, pre ktorú platí:

Definícia 3.1.3

$f(N) = \Theta(g(N))$, ak existujú kladné konštanty C_1, C_2 a N_0 také, že $C_1|g(N)| \leq f(N) \leq C_2|g(N)|$ pre $N \geq N_0$.

Samotný výpočet zložitosti zahŕňa zložité odhady a kombinatorické vzťahy, ktoré však nie sú predmetom týchto skrípt i samotných prednášok. Uvádzame len pojmy a základné definície, ktoré ich definujú preto, aby bolo zrozumiteľné hodnotenie a porovnávanie triediacich a prehľadávacích metód.

3.1.2 Klasifikácia metód

Všetky metódy triedenia sa v zásade delia na dve základné skupiny podľa toho, na akom pamäťovom prostriedku sú triedené položky.

1. Vnútorné triedenie - predstavuje skupinu metód, ktoré predpokladajú, že všetky položky sú v hlavnej pamäti počas celého triedenia, resp. že všetky sa zmestia do operačnej pamäti. Tieto metódy sa niekedy označujú ako metódy triedenia tabuliek.

2. Vonkajšie triedenie - kde nie sú všetky položky v hlavnej pamäti. Buď sa nezmestia, alebo nie je vhodné a potrebné, aby boli. Tieto metódy sa používajú v súvislosti s ošetrením súborov.

Pri metódach vnútorného triedenia sa vychádza z možnosti rýchleho operovania, prístupu k jednotlivým položkám, z možnosti vytvárať účinné štruktúry údajov s efektívnym prístupom k položkám. Pri metódach vonkajšieho triedenia sa vychádza zo skutočnosti, že sú to vždy súbory na vonkajších médiách, vysokokapacitných, ale s výrazne ohraničeným prístupom.

Podľa spôsobu identifikácie kľúča, prípadne porovnania kľúčov, možno metódy triedenia rozdeliť do troch základných typov:

1. Distributívne triedenie - pri triedení sa preveruje kľúč postupne po cifrách.
2. Komparatívne triedenie - v procese triedenia sa porovnávajú dve kľúčové slová.
3. Výpočtom adries - tieto metódy transformujú kľúč do adresy, nachádzajúcej sa priamo alebo blízko miesta, kde sa očakáva uloženie resp. výber položky.

V každej skupine metód je charakteristická určitá akcia, napr. pri metóde triedenia vkladaním, sa hľadá zodpovedajúce miesto a prvok sa tam "vloží", pri metóde výmenou sa porovnávajú dva kľúče a vymenia sa v zmysle definovaného usporiadania atď. V každej takejto skupine je jedna metóda, ktorú možno chápať ako základnú - priamu a z nej sú odvodené ďalšie. Niekedy sa uvádzajú ako nepriame metódy. Sú to metódy, ktoré v niektorej charakteristike zlepšujú vlastnosti pôvodnej metódy alebo metódy, ktoré využívajú pôvodnú k vytvoreniu novej metódy.

3.1.3 Vnútorné triedenie

Metódy vnútorného triedenia sme uviedli ako metódy, ktoré pracujú nad relatívne malým počtom prvkov - položiek. Základným predpokladom je, že sa zmestia do hlavnej pamäti počítača. S ohľadom na rôznorodosť štruktúry samotnej položky, môže pri triedení vzniknúť viac situácií. Napríklad ak položka a (alebo) kľúč zaberá niekoľko slov pamäti, je vhodné vytvoriť si tabuľku adries a pri triedení pracovať s touto tabuľkou bez premiestňovania samotných položiek. Tabuľka adries je vlastne tabuľka ukazovateľov na tieto položky. V takomto prípade sa hovorí o triedení tabuľky adries. Ak sú kľúče krátke a položky dlhé, pre zvýšenie rýchlostí sa môžu preniesť kľúče do tabuľky adries. Vtedy hovoríme o triedení kľúčov. V iných prípadoch sa využívajú určité väzbové informácie, obsiahnuté priamo v položke. V konečnom dôsledku položky prvky tvoria lineárny zoznam. V takomto prípade ide o triedenie zoznamov, resp. na zoznamoch. Pri týchto spôsoboch triedenia je často potrebná pomocná pamäť pre všetky informácie o stave triedenia, resp. pre uchovanie spracúvanej položky.

Najjednoduchšie triedenia vychádzajú z jednoduchej myšlienky porovnávať prvky, a ak nie sú usporiadané, vymeniť ich. Jednotlivé algoritmy sa líšia kritériami, podľa ktorých sa vyberajú dvojice na porovnávanie. V zásade môžeme hovoriť o metódach triedenia:

- vkladaním,
- výmenou,
- výberom, resp. priamym, lineárnym výberom,
- zlučovaním,
- distribúciou a
- výpočtom adries.

Uvedené metódy tvoria v skutočnosti skupiny metód, ktoré majú spoločný princíp operovania s položkou, ale sa môžu líšiť v nárokoch na pamäť a čas, potrebný k realizácii. O každej uvedenej skupine si povieme podrobnejšie, prípadne s uvedením najznámejšieho reprezentanta metódy. Prvé štyri z vymenovaných metód patria do skupiny komparatívneho triedenia.

3.1.3.1 Triedenie vkladaním

Triedenie vkladaním je jednou z komparatívnych metód. Postupne sa porovnáva kľúč položky, ktorej miesto hľadáme už vo vytriedenej množine, s kľúčami položiek tejto vytriedenej množiny. Počet porovnaní je v najhoršom prípade m, ak už vytriedených položiek je m a aktuálna položka "si nájde" miesto až na konci, ako posledná. Postup pri triedení je vidieť na obr. 3.1.1.

Jednoduché vkladanie

Postup pri jednoduchom vkladaní je takýto. Máme utriediť N prvkov. Z nich je už j-1 usporiadaných, to znamená, že pre ich kľúče platí:

$$K_1 \leq K_2 \leq \ldots \leq K_{j-1} \qquad (3.1.8)$$

Ďalší prvok - p_j, kde $j \in \langle 1,N \rangle$, treba zaradiť na zodpovedajúce miesto v zmysle usporiadania. Postupne porovnávame kľúč K_j s kľúčmi K_{j-1}, K_{j-2},..., resp. s $K_1, K_2, \ldots$ až K_{j-1} dovtedy, až kým ho môžeme zaradiť medzi nejaké dva prvky, t.j. medzi p_i a p_{i+1}, prípadne na jeden z dvoch koncov usporiadanej postupnosti prvkov. Prvky p_{i+1} až p_{j-1} (všetky prvky vpravo od nájdenej pozície) posunieme o jedno miesto doprava a prvok p_j vložíme na pozíciu i+1. Spracovaný prvok ako by sa "prepadol" do usporiadanej postupnosti. Preto sa môžeme stretnúť aj s označením ako prepadové triedenie alebo vnáranie, prípadne preosievanie. Tieto názvy (zvlášť preosievanie) sa niekedy používajú na označenie všetkých metód, ktorých základom je porovnávanie kľúčov. Program, ktorý realizuje opísaný postup, je:

```
Program VKLADANIE;
const dmax := 100;
  type index = 0 .. dmax;
       polozka = record
                 kluc : integer;
                 hodnota : chor;
                 /* ostatne zlozky recordu */
       end;

at = array [index] of polozka;
var postupnost : at;
        dlzka : 0 .. dmax;
            i : index;
        prvok : polozka;
procedure VKLAD (var t : at; var n : index);
     var i,j : index;
         pom : polozka;
     begin
          for i := 2 to n do
          begin pom := t[i];
                t[Ø] := pom;
                   j := i-1;
          while pom.kluc < t[j].kluc do
                begin t[j+1] := t[j];
                            j := j-1
                end;
          t[j+1] := pom;
          end
     end /* koniec procedury VKLAD */;
     begin
          prvok.hodnota := 'Z';
                read (prvok.kluc);
                      dlzka := Ø;
                      /* Ø je koncovy znak */;
                while prvok.kluc < >Ø do
                begin
                      dlzka := dlzka+1;
                      postupnost [dlzka] := prvok;
                  read (prvok.kluc)
                end;
                begin VKLAD(postupnost,dlzka);
                for i := 1 to dlzka do
                  WRITE (postupnost [i].kluc);
                  WRITELN.
                end
     end.
```

Príklad 3.1.2

Máme utriediť desať prvkov s kľúčami:

403,067,416,015,750,150,690,307,457,632

Vkladané prvky :	Utriedené prvky :	
067 :	403	- 1 posuv
416 :	067 403	
015 :	067 403 416	- 3 posuvy
750 :	015 067 403 416	
150 :	015 067 403 416 750	- 3 posuvy
690 :	015 067 150 403 416 750	- 1 posuv
307 :	015 067 150 403 416 690 750	- 4 posuvy
457 :	015 067 150 307 403 416 690 750	- 2 posuvy
632 :	015 067 150 307 403 416 457 690 750	- 2 posuvy
:	015 067 150 307 403 416 457 632 690 750	

Obr. 3.1.1
Triedenie vkladaním

Metóda je pomerne jednoduchá, ale má nevýhodu. Pri nájdení pozície pre aktuálny záznam sa musia všetky prvky napravo posunúť o 1 miesto vpravo. V prípade, že prvok rušíme - posunúť v opačnom smere. V uvedenom príklade sa muselo urobiť 16 posuvov a dva vklady boli na konci postupnosti.

Čas, potrebný pre utriedenie je determinovaný dvoma druhmi operácií:

1. počtom porovnaní - P
2. počtom posunov - Q

Pre hodnotenie metódy platí: Ak sa spracúva j-tý prvok, jeho kľúč sa porovnáva v priemere s j/2 už usporiadanými kľúčami. Stredná hodnota porovnaní je daná výrazom:

$$(1+2+...+N)/2 \doteq N^2/4 \qquad (3.1.9)$$

Pre veľké N je to veľký počet.

Pre počet posuvov platí:

$$Q = N(N-1)/4 \qquad (3.1.10)$$

Vidíme, že v obidvoch prípadoch vystúpi vo výrazoch N^2. To znamená, že výpočtová zložitosť metódy je $O(N^2)$.

3.1.3.2 Metódy triedením výmenou

Princíp tejto metódy, resp. skupiny metód je jednoduchý a v mnohom podobný triedeniu vkladaním. Dokonca sa niekedy metódy triedenia výmenou, výberom a vkladaním explicitne nerozlišujú. Preto sa môžeme v literatúre stretnúť aj pri tejto metóde s označením, ako je triedenie preosievaním, resp. bublinkové.

Pri metóde výmenou sa tiež porovnávajú dvojice kľúčov. Na základe porovnania sa dávajú do správneho poradia v zmysle utriedenia. To znamená, že sa buď navzájom vymenia - preto triedenie výmenou - alebo, ak sú v správnom poradí, druhý kľúč z nich sa porovná s ďalším nasledujúcim. Pri každom prechode triedenej množiny prvkov minimálne jeden prvok sa dostane na zodpovedajúce miesto. To znamená, že tento sa už nebude vymieňať. Existuje niekoľko metód založených na výmene prvkov. Možno povedať, že najjednoduchšie je triedenie, pri ktorom sa porovnávajú susedné dvojice kľúčov, ďalej triedenie zlučovaním, nazývané tiež paralelné triedenie Betčerovo, Shellovo triedenie, pri ktorom sa porovnávajú kľúče vzdialené o určitú vzdialenosť a známe tzv. rýchle triedenie - Quicksort.

3.1.3.2.1 Triedenie výmenou

Pod týmto názvom rozumieme triedenie s porovnávaním dvoch susedných kľúčov. Postup je tento: Máme usporiadať N prvkov. Porovnáme kľúč K_1 (patriaci prvku p_1), ktorý je na pozícii R_1, s kľúčom K_2 na pozícii R_2. Ak vzhľadom na definované usporiadanie zostanú prvky na pôvodnej pozícii, porovná sa kľúč K_2 na pozícii R_2 s kľúčom K_3 na pozícii R_3. Ak sa vzhľadom na definované usporiadanie vymenili K_1 s K_2, tak sa porovnáva K_1 na pozícii R_2 s kľúčom K_3 na pozícii R_3. Vykonávaním týchto operácií sa záznam s veľkým kľúčom (ak triedime do rastúcej postupnosti) postupne "prepadáva", "preosieva sa" až na svoju pozíciu R_N. Celý proces je názorne zachytený na obr. 3.1.2 pri rovnakom zadaní, aké je v príklade 3.1.1, ale metódou výmeny.

Máme utriediť prvky s kľúčami:

403,067,416,015,750,150,690,307,457,632

neutriedené kľúče	Prechody blokom 1.	2.	3.	4.
403	067	067	015	015
067	403	015	067	067
416	015	403	150	150
015	416	150	403	307
750	150	416	307	403
150	690	307	416	416
690	307	457	457	457
307	457	632	632	632
457	632	690	690	690
632	750	750	750	750

Obr. 3.1.2
Triedenie výmenou

Vidíme, že bolo potrebné urobiť 16 výmen (pri vkladaní - 16 vkladaní), ale na štyri prechody cez celú postupnosť sú kľúče utriedené. Procedúra TRVYM používa rovnaké typy údajov ako v programe VKLADANIE.

```
procedure TRVYM (var t : at, n : index);
        var i,j : index;
            pom : polozka;
            bolavymena : boolean;
        begin i := 2;
            bolavymena := true;
            while i ≤ n ∧ bolavymena do
        begin bolavymena := false;
            begin for j := n downto i do
                if t[j-1].kluc > t[j].kluc then
                begin pom := t[j-1];
                      t[j-1] := t[j];
                        t[j] := pom;
                      bolavymena := true;
                end
            end;
                i := i+1
            end /* koniec TRVYM */;
```

Na indikáciu stavu, kedy je už celá postupnosť usporiadaná, slúži logická premenná "bolavymena".

Pri vyhodnotení časovej náročnosti uvažujeme:

- počet prechodov,
- počet výmen,
- počet porovnaní.

Výpočtová zložitosť metódy je $O(N^2)$. Je to časovo náročná metóda. Existuje však dôkaz, že pre maximálny počet prvkov rovný 11, je to najrýchlejšia metóda triedenia.

3.1.3.2.2 Shellovo triedenie

Metóda je založená na výmene dvoch prvkov. Je to teda komparatívne triedenie, niekedy uvádzané ako rýchle komparatívne alebo so zmenšujúcim sa krokom. Postup je rovnaký ako pri výmennom triedení, ale sa neporovnávajú kľúče susedných prvkov, ale prvkov vzdialených o určitú dĺžku d. Po každom prechode celou tabuľkou, sa hodnota d znižuje (zvyčajne) podľa vzťahu:

$$d_{i+1} = \frac{d_i}{2} \quad \text{a} \quad d_0 = \frac{N}{2} \qquad (3.1.11)$$

Pri každom prechode sa každý kľúč porovná s kľúčom vzdialeným o d_i. Ak je potrebná výmena, realizuje sa a porovnanie pokračuje, ak existuje pár pre porovnávanie. Potom sa určí d_{i+1} a celý proces, ale s novým krokom sa opakuje od začiatku. Každý prechod je modifikácia metódy výmenou a v prechode, kedy d=1, je zhodný s touto metódou.

Priebeh triedenia je dokumentovaný na obr. 3.1.3.

Kľúč	1. prechod $d_1=6$	2. prechod $d_2=3$	3. prechod $d_3=2$	4. prechod $d_4=1$	
503	503	503,052	052	052	052
092	092	092	092	092	092
515	515	515,167	167,152	152	152
052	052	052,503	503	503,167	167
920	920,152	152	152,167	167,503,270	270
167	167	167,515	515,270	270,503,425	425
890	890,680	680,425	425	425,503	503
270	270	270	270,515	515	515
632	632	632,610	610	610	610
425	425	425,680	680,632	632	632
152	152,920	920	920,890	890,680	680
610	610	610,632	632,680	680,890	890
680	680,890	890	890,920	920	920

Obr. 3.1.3
Shellovo triedenie

V procedúre SHELL môžeme použiť ako pomocnú procedúru VYMEN na realizáciu výmeny dvoch prvkov, použitím jednej pomocnej premennej. Túto môžeme použiť vôbec vo všetkých programoch pre takúto výmenu.

```
procedure VYMEN (var kus1, kus2 : polozka);
        var pom : polozka;
    begin
        pom := kus1;
        kus1 := kus2;
        kus2 := pom
    end /* koniec VYMEN */;
procedure SHELL (var t : at; var n : index);
        var i,m,d : index;
            bolavymena : boolean;
    begin
        d := n;
        while d > 1 do
            begin
                d := (d) div 2;
              repeat
                    bolavymena := false;
            for i := 1 to n-d do
              if t[i].kluc > t[i+d].kluc then
              begin VYMEN (t[i], t[i+d]);
                    bolavymena := true
              end
              until not bolavymena
            end
    end /* koniec SHELL */;
```

Pre túto metódu sa dosiaľ nepodarilo nájsť exaktný vzťah, vyjadrujúci časovú zložitosť metódy. Nie je zatiaľ známy presný vzťah medzi N, d, rozložení položiek a vplyv jedného prechodu na druhý. Z toho dôvodu sa nepodarilo zatiaľ odvodiť najlepšiu postupnosť prechodov, resp. fáz určenú d, pre veľké N. Vieme len, že v konečnom dôsledku časovú zložitosť ovplyvňuje päť faktorov:

N - rozsah triedeného bloku,
T - počet prechodov,
S - súčet všetkých d_i,
B - počet výmen a
výraz: (B+NT-S-A) - počet porovnaní, kde A - počet "ľavostranných miním" (vždy, keď sa nemusí robiť výmena na začiatku bloku, pretože len prvok na začiatku je minimálny).

Experimentovaním sa zistilo, že čas potrebný pre utriedenie je viac závislý od rozloženia kľúčov ako od ich počtu. Empiricky sa získal odhad $O(\alpha.N.(\log N))^2$, kde α je konštanta úmernosti.

3.1.3.2.3 Rýchle triedenie - Quicksort

Určite je jednoduchšie vytriediť dve menšie tabuľky položiek ako jednu veľkú. Na myšlienke rozdeliť pôvodnú tabuľku na dve a tieto triediť je založená metóda rýchleho triedenia. Dá sa povedať, že žiadnemu algoritmu triedenia sa nevenovala väčšia pozornosť odborníkov ako práve tomuto.

Kľúčovým krokom algoritmu je určenie miesta, v ktorom sa pôvodná tabuľka rozdelí na dve časti. Ideálne je, ak sú obidve časti rovnaké, alebo aspoň nie podstatne rozdielne vzhľadom na počet položiek. Pretože nijako nevieme zaručiť ideálne rozdelenie, ako vhodné priblíženie sa uvažuje priemer hodnoty prvého, prostredného a posledného kľúča, prípadne sa zoberie prostredný kľúč (v anglickej literatúre označený ako "pivot" - stred). Prípadne sa uvažuje jedna z krajných hodnôt kľúčov. Potom celý proces prebieha nasledovne:

Máme dva ukazovatele i,j, ktoré na začiatku majú hodnotu i = 1, j = N. Porovnávame K_i a K_j, ak sa nerealizuje výmena, potom j := j-1 a porovnávame K_i a K_{j-1}. j sa zmenšuje po prvú výmenu. Po prvej výmene: i := i+1 a porovnávame K_i a K_j zase až po prvú výmenu.

Takýmto porovnávaním a výmenami - raz zľava, raz sprava - sa určený "stredový" kľúč dostane na zodpovedajúce miesto (i=j), všetky kľúče $K_1 \ldots K_{i-1}$ sú už menšie ako K_i a všetky kľúče $K_{i+1} \ldots K_M$ sú väčšie ako K_i. Tento kľúč sa už viac nepresúva. Rozdelil nám pôvodný úsek zase na dva a celý proces opakujeme na takto vydelený menší úsek. Celý proces je názornejší na uvedenom obr. 3.1.4 (l,r - aktuálne intervaly) a v stromovej reprezentácii na obr. 3.1.5.

V obr. 3.1.6 je vždy vyznačený úsek oválom, ktorý sa má triediť a (*) má rovnaký význam ako v tabuľke.

Rekurzívna procedúra pre rýchle triedenie:

```
procedure QUICK (var t : at; var d,h : index; n : index);
procedure SORT (d,h : index);
          var i,j : index;
          delbod,pom : polozka;
   begin
          i := d; j := h;
          delbod := t[(d+h)div 2];
```

```
repeat
        while t[i].kluc < delbod.kluc do
           i := i+1;
        while delbod.kluc < t[j].kluc do
           j := j-1;
        if i ≤ j then
        begin
              pom := t[i]; t[i] := t[j]; t[j] := pom;
              i := i+1; j := j-1
        end
until i > j;
        if  d<j  then  SORT (d,j);
        if  i<h  then  SORT (i,h)
end;
begin SORT (1,n)
end /* koniec QUICK */;
```

Metóda sa dá realizovať aj iteračným (nerekurzívnym) spôsobom. V takom prípade potrebujeme údajovú štruktúru, do ktorej ukladáme (a postupne vyberáme) informácie s hranicami podúsekov, ktoré boli už vytvorené, ale ešte neutriedené. Štruktúra, ktorá rešpektuje definovaný postup (viď obr. 3.1.4 a

													(l,r)
|503	092	515	052	920	167	890	270	632	425	152	610	680	(1,13)
|152	092	515	052	920	167	890	270	632	425	503 |	610	.	
.	.	|503	052	920	167	890	270	632	425	| 515	.	.	
.	.	425	|052	920	167	890	270	632	503	| 515	.	.	
.	.	.	.	|503	167	890	270	632|	920	.	.	.	
.	.	.	.	270|	167	890	503|	632	.	.	.	.	(1,6)
152	092	425	052	270	167 |	503|	890	632	920	515	610	680	(8,13)
052	|092	425	152|	270	167								
052	092	|152|	425	270	167								
		(*)	167	270	425								
							|890	632	920	515	610	680	(8,13)
							|680	632	920	515	610	890	
							610	|632	920	515	680|	.	
							.	.	|680	515	920	.	
							610	632	515 |	680|	920	890	
						(*)	515	610	632 |		.	.	(8,10)
										(*)	890	920	(12,13)
052	092	152	167	270	425	503	515	610	632	680	890	920	

Obr. 3.1.4
Rýchle triedenie

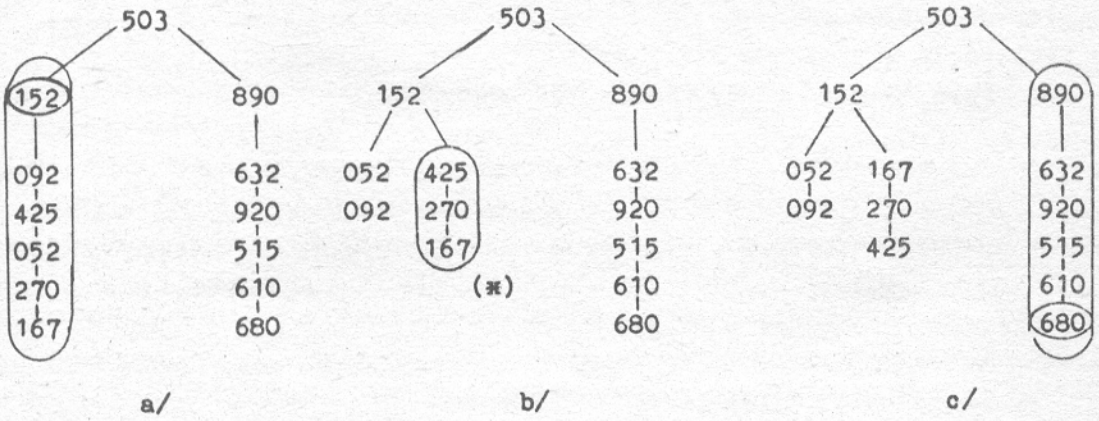
3.1.6) výberu úseku na triedenie, je zásobník. V programe RYCHLETRIEDENIENR je nerekurzívne realizované rýchle triedenie:

```
Program RYCHLETRIEDENIENR;
/* deklarácie z programu 3.1.1 a
        type zasob : array [index] of record
                                        dolna : index
                                        horna : index
                                    end */
procedure QUICKNR (var t : at; var z : zasob; var d,h : index; n : index);
      var k : index;
/* deklarácia procedury SORT */;
begin
      k := 1;  z[1].dolna := 1; z [1].horna :=n ;
repeat /* utriedi sa úsek určený vrcholom zásobníka */
      d := z[k].dolna;
      h := z[k].horna;
      k := k-1;
repeat
       SORT (d,h);
   if j < h then
       begin /* uložia sa hranice pravého podúseku */
       k := k+1;  z[k].dolna := j;  z [k].horna := h

   end;
   /* pokračuje spracovávanie ľavého podúseku */
       h := i;
   while d ≥ h
        until k=0
   end /* koniec QUICKNR */
   /* príkazy programu RYCHLETRIEDENIENR */
end.
```

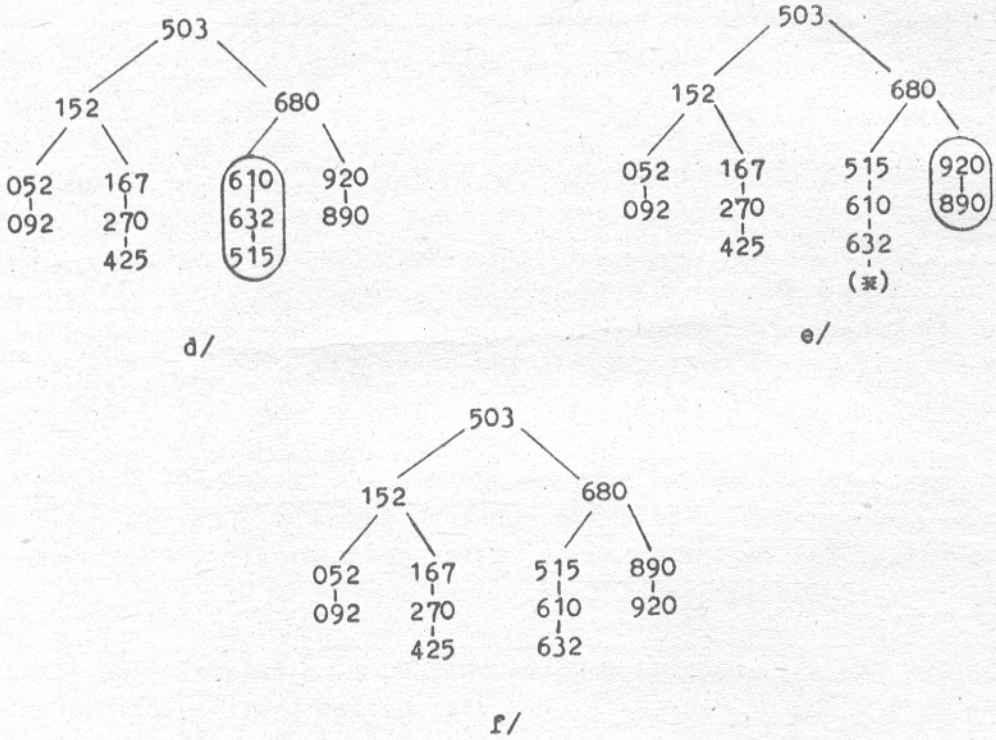
Po každom rozdelení na podúseky sa pokračuje v spracúvaní ľavého podúseku a pravý - jeho d a h hranica (ak obsahuje viac ako jeden prvok) - sa uložia do zásobníka. Ten sa spracúva, až všetky podúseky vľavo sú usporiadané t.j. všetky prvky vľavo sú usporiadané.

Z hľadiska časovej zložitosti patrí QUICKSORT do triedy metód, ktorých zložitosť je O(N log N), metóda skrýva však určitú protichodnosť. Čím usporiadanejšia je východzia postupnosť, tým je časová zložitosť tejto metódy horšia. V krajnom prípade, t.j., ak by sme vyšli už z vytriedenej postupnosti kľúčov, t.j. že by platilo $K_1 < K_2 < \ldots < K_N$, stáva sa táto metóda "nie rýchla" a pre jej výpočtovú zložitosť platí $O(N^2)$.

Ak je už interval l,r malý, usporiada sa už napr. výmennou metódou viď:



Obr. 3.1.5
Stromová reprezentácia rýchleho triedenia-začiatok



Obr. 3.1.6
Stromová reprezentácia rýchleho triedenia

Na rozdiel od ostatných metód, QUICKSORT jednoznačne "dáva prednosť" v súvislosti s jej vyhodnotením, nevytriedením (resp. čo najmenej usporiadaným) množinám prvkov. Potom pre $N \rightarrow \infty$ sa zložitosť asymptoticky blíži k N.log N.

3.1.3.3 Triedenie výberom

Triedenie výberom predstavuje triedu metód, ktorých princíp sa dá opísať takto: treba vytriediť N prvkov s kľúčami K_i. V každej fáze sa hľadá maximálny alebo minimálny kľúč. Po nájdení sa vyberie a zaradí ako ďalší prvok do vytriedenej množiny prvkov. Na pozíciu, z ktorej sa vybral, zapíše sa 0 alebo iná zvolená náhrada taká, aby bolo jasné, že je to už "vybrané" miesto. Metóda si vyžaduje doplnkovú pamäť kapacity pre všetkých N prvkov. Pri tzv. jednoduchom výbere sa nepoužíva doplnková pamäť, ale vytvára sa vytriedená postupnosť kľúčov z jedného konca. Až do rastúcej postupnosti sa hľadá minimálny prvok a uloží sa na zodpovedajúcu pozíciu z ľavého konca. Prvok z tejto pozície sa vymení s nájdeným minimálnym. Takto sa realizuje triedenie procedúra VYBER. Pri klesajúcej postupnosti sa ukladá z pravého konca. Prvky, ktoré boli minimálne v jednotlivých fázach, tvoria už usporiadanú postupnosť, to znamená, že po každej fáze pribudne jeden prvok v usporiadanej postupnosti prvkov a ubudne jeden v tej, ktorú treba ešte triediť. Postup je na obr. 3.1.7.

403	067	416	015	750	150	690	307	457	632
403	067	416	015	632	150	690	307	457 |	750
403	067	416	015	632	150	457	307 |	690	750
403	067	416	015	307	150 |	457	632	690	750
403	067	150	015	307 |	416	457	632	690	750
307	067	150	015 |	403	416	457	632	690	750
015	067 |	150 |	307	403	416	457	632	690	750

Obr. 3.1.7
Triedenie výberom

Metóde zodpovedá procedúra VYBER:

```
procedure VYBER (t : at; n : index);
          var i,j,k : index;
                pom : polozka;
begin
      for i := 1 to n-1 do
  begin k := i;
        pom := t[i];
```

```
    for j := i+1 to n do
      if t[j].kluc < pom.kluc then
         begin k := j;
               pom := t[j]
         end;
      t[k] := t[i];  t[i] := pom
  end
end /* koniec VYBER */;
```

Čas potrebný na usporiadanie N prvkov touto metódou je závislý od

- počtu triedených prvkov N,
- počtu porovnaní P a
- od počtu výmen V (krok(*)).

Počet porovnaní sa po každom prechode zmenší o jedno. Platí:

$$P = \binom{N}{2} = \frac{1}{2} N(N-1) \qquad (3.1.12)$$

Premennou veličinou je len V, ktoré sa nedá pochopiteľne presne analyzovať a doteraz sa ho nepodarilo presne definovať. V najhoršom prípade platí:

$$V = \max (N^2/4) \qquad (3.1.13)$$

Výpočtová zložitosť metódy teda je $O(N^2)$. Vidíme, že je to rovnaká zložitosť ako pri triedení výmenou a vkladaním.

Detailnejšie analýzy však ukazujú, že táto metóda je trochu pomalšia ako metóda vkladaním a asi dvakrát pomalšia, ako metóda výmenou. Určitým vyvážením tohto nedostatku je zase skutočnosť, že nerobia sa tu presuny ako pri vkladaní.

3.1.3.4 Triedenie zlučovaním

Z historického hľadiska počítačov a triediacich metód patria metódy triedenia zlučovaním k najstarším triediacim metódam. Prvý algoritmus z týchto metód vytvoril roku 1945 vo Neuman. Dnes je to už celá škála metód, ktoré sa odvíjajú od základného princípu - od zlučovania. Zlučovanie znamená vytvorenie jednej usporiadanej množiny prvkov z dvoch alebo viacerých už usporiadaných postupností prvkov. Tieto metódy sa častejšie používajú pri triedení súborov, teda pri triedení údajov na vonkajších pamätiach.

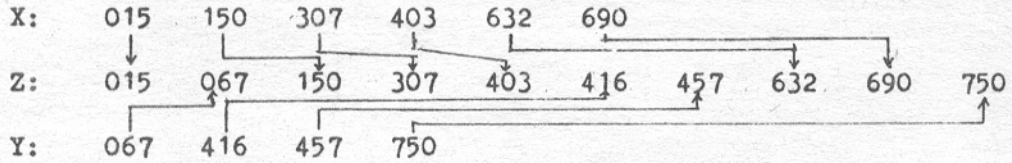
Majme usporiadané dve postupnosti kľúčov:

$$X: x_1 \leq x_2 \leq \ldots \leq x_m$$
$$Y: y_1 \leq y_2 \leq \ldots \leq y_n$$

Zlúčením v uvedenom zmysle dostaneme jednu usporiadanú postupnosť Z:

$$Z: z_1 \leq z_2 \leq \ldots \leq z_{m+n}$$

Na obr. 3.1.8 je znázornený postup pri zlučovaní dvoch postupností do jednej:



Obr. 3.1.8
Postup pri zlučovaní

Takýto spôsob zlučovania sa nazýva dvojcestné zlučovanie. Ak m = n, potom je táto metóda "najlepšia z možných" spôsobov zlučovania. Všeobecne je množstvo práce úmerné m+n. To znamená, že zlučovanie je podstatne jednoduchší proces ako triedenie.

Pri vnútornom triedení sa princíp zlučovania tiež používa, ale s iným postojom k vstupným, už usporiadaným - postupnostiam prvkov. Rozdiel je predovšetkým v tom, že pri vnútornom triedení je vo väčšine prípadov potrebné usporiadať jednu množinu prvkov. Môžeme ju chápať ako niekoľko usporiadaných podmnožín/postupností. V limitnom prípade pri počte prvkov N, ako N usporiadaných podmnožín s dĺžkou = 1. Problém triedenia potom môžeme previesť na zlučovanie, a to postupným zlučovaním jednotlivých podmnožín - vždy po dva do jedného. Ale vzhľadom na okolie aktuálneho prvku je to už proces vkladania, pretože vloženie jedného prvku do usporiadanej množiny zodpovedá zlučovaniu pri n=1.

Všetky usporiadané prvky sú rozdelené na podmnožiny, ktoré sú usporiadané, pričom sa postupuje z obidvoch koncov a usporiadanosť sa chápe v smere postupu, t.j. z $\longrightarrow$k a k$\longrightarrow$z. Vyberajú, porovnávajú a zlučujú sa prvky z obidvoch koncov zodpovedajúcich si podmnožín. Na obr. 3.1.9 je šípkou, skratkou L (ľavý), P (pravý) a číslom naznačené, ktorá podmnožina zlúčením vznikne.

```
   1.L    2.L      3.L     4.L    4.P          3.P      2.P  1.P
   503 | 092  515 | 052  920 | 167 | 890  270 | 632  425  152 | 610 | 680
1. fáza:
   | 1.L    1.P    2.L     2.P   2.P     2.L      1.P     1.L
   503  680 | 052  152  425  632  920 | 890  270  167 | 610  512  092
2. fáza:
     1.L              1.P              1.P      1.L

   092  503  512  610  680 | 920  890  632  425  270  167  152  052
3. fáza:

   052  092  152  167  270  425  503  512  610  632  680  890  920
```

Obr. 3.1.9
Prirodzené dvojcestné zlučovanie

To znamená, že 1.L a 1.P vytvoria v ďalšej fáze 1.L, 2.L a 2.P vytvoria 1.P atď., do stredu triedenej postupnosti. Vertikálnymi čiarami sú oddelené jednotlivé usporiadané úseky. Tento postup zlučovania nazýva sa <u>prirodzené dvojcestné zlučovanie</u> alebo zjednodušene prirodzené zlučovanie. Za predpokladu, že počet prvkov v zlučovaných postupnostiach je N - v každej, potom nárok na pamäť pri tejto metóde je 2N. Pred zlučovaním, t.j. po prvom rozdelení na podmnožiny, budeme mať približne $\frac{1}{2}$ N usporiadaných podmnožín, pretože $K_i > K_{i+1}$ s pravdepodobnosťou $\frac{1}{2}$. Po každej fáze, t.j. po zlúčení po dvoch cez celú množinu, počet takýchto intervalov sa zmenší na polovicu. Počet prezretí prvkov je úmerný $\log_2 N$. Pri každej fáze sa prepíše N prvkov. Výpočtová zložitosť metódy je O(N log N). Rovnakú výpočtovú zložitosť má aj metóda <u>jednoduchého</u> alebo aj <u>priameho</u> dvojcestného zlučovania, ktorá je v praxi o niečo rýchlejšia. Postup pri priamom zlučovaní je nasledovný. Na začiatku sa rozdelí celá množina s N prvkami na N podmnožín. Každý prvok tvorí jednu podmnožinu. Každým prechodom sa zdvojnásobní počet už usporiadaných prvkov v podmnožinách. Počet podmnožín sa pochopiteľne úmerne tomu zmenšuje. Postup je znázornený na obr. 3.1.10. Tomuto zodpovedá procedúra PRIAMZLUC.

Treba si uvedomiť, že v programe, v ktorom použijeme nasledujúcu procedúru, musíme uvažovať DMAX (viď program VKLADANIE) s dvojnásobnou dĺžkou, resp. 2N. Ostatné zostáva nezmenené.

```
procedure PRIAMZLUC (var t : at; var n : index);
          var i,j,k,l,t : index;
              h,m,p,q,r : integer;
                     up : boolean;
```

```
begin up := true;  p := 1;
    repeat h := 1; m := n;
      if up then
         begin i := 1; j := n; k := n+1; l := 2*n
               end else
               begin k := 1; l := n; i := n+1; j := 2*n
                 end;
    repeat
      /* q - dĺžka postupnosti s indexom i, r - dĺžka postupnosti
         s indexom j */
      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 ≠ Ø) ∧ (r ≠ 0) do
    begin
          /* začiatok zlučovania */
          if t[i].kluc < t[j].kluc then
          begin t[k] := t[i]; k := k+h; i := i+1; q := q-1
          end
              else
           begin t[k] := t[j]; k := k+h; j := j-1; r := r-1
           end
    end;
    /* kopírovanie ostatných prvkov s indexom j */
    while r ≠ 0 do
    begin t[k] := t[j]; k := k+h; j := j-1; r := r-1
    end;

    /* kopírovanie ostatných prvkov s indexom i */
    while q ≠ Ø do
    begin t[k] := t[i]; k := k+h; i := i+1; q := q-1
    end;
    h := -h; t := k; k := l; l := t
       until m = Ø;
          up := not up;
           p := 2*p
       until p ≥ n;
          if not up then
          for i := 1 to n do t[i] := t[i+n]
  end /* PRIAMZLUC */;
```

Ostatné metódy zlučovania sa používajú pri vonkajšom triedení a sú uvedené v kapitole o vonkajšom triedení.

503 | 092 | 515 | 052 | 920 | 167 | 890 | 270 | 632 | 425 | 152 | 610 | 680

1. fáza:

503 680 | 152 515 | 632 920 | 890 | 270 167 | 425 052 | 610 092 |

2. fáza:

092 503 610 680 | 167 270 632 920 | 890 | 515 425 152 052

3. fáza:

052 092 152 425 503 515 610 680 | 890 920 632 270 167

4. fáza:

052 092 152 167 270 425 503 515 610 632 680 920 890

5. fáza:

052 092 152 167 270 425 503 515 610 632 680 890 920

Obr. 3.1.10
Priame dvojcestné zlučovanie

3.1.3.5 Distributívne triedenie

Základný princíp týchto metód je v triedení kľúčov - číselných kľúčov - tak, že pri triedení sa preveruje kľúč po jednotlivých cifrách. Najznámejšou distributívnou metódou je tzv. radixové triedenie alebo triedenie grupovaním.

Postup pri triedení je takýto:

- najskôr sa preveruje rádove najnižšia cifra kľúča a vytvárajú sa - distribujú sa - grupy s rovnakými ciframi v tomto ráde, t.j. vytvorí sa grupa s 0, ďalšia s 1 atď. až po 9.
- Grupy sa zoradia v zostupnom poradí.
- Proces sa opakuje v najbližšom vyššom ráde a zase sa vytvoria grupy z tohto rádu.

Tento proces sa opakuje po najvyšší rád. Po zoradení grúp podľa najvyššieho rádu sú všetky prvky usporiadané. Viď obr. 3.1.11.

Z opisu postupu pri tomto triedení i z obrázku vidíme, že každá fáza sa skladá z dvoch druhov činnosti:

1. distribúcie grúp a
2. zoradenia.

Kľúče v číselnej sústave so základom Z sa rozdelia na Z grúp. Počet fáz - - prechodov pre všetky rády je daný maximálnym počtom rádov kľúčov R.

Tieto údaje spolu s počtom záznamov N ovplyvňujú časovú zložitosť radixovej metódy triedenia. Výpočtová zložitosť samotnej distribúcie je rovná $O(\log_Z R)$. Výpočtová zložitosť celého procesu pre N prvkov je $O(N \log_Z R)$.

Kľúče:		1. distribúcia (v 1-vom ráde)	1. zoradenie	2. distribúcia (v 10-vom ráde)		2.zoradenie	3. distribúcia (v 100-vom ráde)		3. zoradenie
503	0	920,890,270	920	0	503	503	0	052,092	052
092		610,680	890	1	610,515	610	1	152,167	092
515	1		270			515			152
052	2	092,052,632,152	610	2	920,425	920	2	270	167
920	3	503	680	3	632	425	3		270
167			092	4		632	4	425	425
890	4		052	5	052,152	052	5	503,515	503
270	5	515,425	632			152			515
632	6	167	152	6	167	167			610
425	7		503	7	270	270	6	610,632,680	632
152	8		515	8	680	680	7		680
610			425	9	890,092	890	8	890	890
680	9		167			092	9	920	920

Obr. 3.1.11
Radixové triedenie

V porovnaní s inými metódami vidíme, že je to rýchla metóda, ale má jednu veľkú nevýhodu. Je náročná na pamäť. Pre každú grupu musíme rezervovať N miest, t.j. celkove Z.N miest je pamäťová náročnosť tejto metódy. Preto sa častejšie ako pri vnútornom triedení používa pri vonkajšom triedení - pri triedení štítkov na tabelátoroch.

Procedúra RADIX okrem typov defonovaných v programe VKLADANIE, používa ešte tieto:

```
var grupa : array [Ø.. 9] of integer;
        tk : array [Ø.. 9,1.. n ] of at;

procedure RADIX (var t : at; n : index; rad : integer);
        var i,j,l,m,q : index;
  begin q := 1;
    for l := 1 to rad do
    begin
        for i := Ø to 9 do
            grupa[i] := 1;
        for i := 1 to n do
      begin j := POSLRAD (T[i].kluc div q);
  /* funkcia, ktorá vyberá - postupne - spracovávaný rád */
        tk[[j], grupa [j]] : = t[i];
                grupa [j]: = grupa [j] + 1
      end;
        m := 1;
      for i := Ø to 9 do
        begin j := 1;
        while j < grupa[i] do
              begin t[m] := tk[i,j];
              m := m+1;
              j := j+1
              end
        end;
          q := q*10
    end
  end /* koniec RADIX */;
```

Výmenné distributívne triedenie je v princípe podobné triedenie ako radixové, ale vychádza z predpokladu, že kľúče sú lineárne kódované. Porovnávajú sa kľúče v binárnej reprezentácii, t.j. "0" a "1" po jednotlivých bitoch. Ďalšia odlišnosť je v tom, že robí sa okamžite výmena a grupu tvorí tá skupina prvkov, ktoré sa zhodujú v prvých R rádoch. Svojou podstatou - reprezentáciou kľúčov - je táto metóda predurčená pre používanie na počítači. Patrí preto k najzákladnejším metódam triedenia.

Ako sme už spomenuli, grupu tvorí vždy skupina prvkov, resp. kľúčov, ktoré sa zhodujú v prvých R rádoch. Postupuje sa teda od najvyššieho rádu k najnižšiemu. V zásade môžeme metódu opísať takto:

1. Pri triedení sa prechádza cez najvyšší rád a postupuje sa zhora nadol a zdola nahor. Kľúč, začínajúci jednotkou (zhora) sa vymení s kľúčom začínajúcim sa nulou (idúc zdola). Po prechode cez všetky kľúče sú všetky kľúče s nulou v najvyššom ráde v hornej časti bloku a kľúče s jednotkou v najvyššom ráde v dolnej časti bloku. Vznikli tak dve grupy. Vymieňajú sa vždy najľavejší zhora (s "1") a najpravejší zdola (s "0").

2. Takto spracované kľúče tvoria už dve skupiny. Zoberieme prvú skupinu (grupu) v najvyššom bite s "0", posunieme sa o rád nižšie a triedime túto skupinu - zase vzniknú dve skupiny - jedna s kľúčmi, ktoré v tomto ráde majú "0" a "1".

Máme utriediť tieto kľúče

27,32,5,21,3,29,7,9,13,19,8

10-vá reprez. \ 2-vá reprez.						
27	11011	01000	00111	00011	00011	00011
30	11110	01101	00011	00111	00101	00101
5	00101	00101	00101	00101	00111	00111
21	10101	01001	01001	01001	01001	01000
3	00011	00011	01101	01000	01000	01001
29	11101	00111	01000	01101	01101	01101
7	00111	11101				
9	01001	10101				
13	01101	11110				
19	10011	10011				
8	01000	11011				

Obr. 3.1.12
Výmenné distributívne triedenie

Pretože sa robia hneď výmeny, nie je táto metóda náročná na pamäť. Časová zložitosť je rádove rovnaká, ako pri rýchlom distributívnom triedení, a to $O(N \log_2 N)$.

3.1.3.6 Triedenie výpočtom adries

Základnú myšlienku tejto metódy si môžeme priblížiť príkladom. Dostaneme hárok čistého papiera, na ktorý máme zapisovať diktované slová tak, aby boli po ukončení diktátu usporiadané v abecednom poriadku. Usporadúvame ich podľa možnosti bez presunov. To znamená, že ak počujeme slovo začínajúce písmenom A, zapíšeme ho do hornej časti hárku, ak slovo začína napr. písmenom M - niekde do stredu atď. Ak takúto metódu aplikujeme na triedenie čísiel, z toho vyplýva, že pre efektívne spracovanie treba poznať rozsah čísiel, ktoré sa majú triediť a tiež niečo o ich rozmiestnení.

Základný princíp je podobný ako pri triedení vkladaním s tým rozdielom, že vytvorí sa niekoľko skupín - intervalov, s ohľadom na rozsah kľúčov. Každá skupina má k dispozícii určitú kapacitu pamäti. Kľúče sa zaraďujú do jednotlivých skupín. Postup je ukázaný na obr. 3.1.13, kde máme už našu známu postupnosť kľúčov:

503,092,515,052,920,167,890,270,632,425,152,610,680

Ak vieme, že kľúče majú rozsah hodnôt od 52 po 920 a poznáme ich rozloženie, vytvoríme tri skupiny s uvedenými rozsahmi:

1. 0 - 200,
2. 201 - 600,
3. 601 - 1000.

	Po prečítaní prvých 5 kľúčov	Po prečítaní prvých 10 kľúčov	Po prečítaní všetkých kľúčov
1.:	052,092	052,092,167	052,092,152,167
2.:	503,515	270,425,503,515	270,425,503,515
3.:	920	632,890,920	610,632,680,890,920

Obr. 3.1.13
Postup triedenia výpočtom adries

Vidíme, že metóda si vyžaduje doplnkovú pamäť v tom zmysle, že v každom intervale musí byť určitá rezerva. Samotný proces triedenia nie je jednoduchý a bez problémov. A sú to hneď dva problémy.

Po prvé - treba vypočítať adresu, t.j. index I, miesta M vo vektore, v ktorom má byť položka s hodnotou t. Význam M a I vidieť na obr. 3.1.14.

(M[I] = I). Ak N čísiel má rovnomerné rozloženie a rozsah od k_1 po k_2 a M má rozsah D (D miest), potom je odhad I daný ako

$$I \cong (t - k_1) \cdot C \qquad (3.1.14)$$

kde

$$C = D/(k_2 - k_1)$$

Ak bude rozloženie položiek nerovnomerné, odhady budú tiež komplikovanejšie.

Po druhé - rovnaká adresa sa môže vypočítať pre viac ako jednu položku, t.j. nastane duplicita. To znamená, že M[I] je už obsadené, a treba hľadať iné miesto pre novú položku. Všetky položky ale musia byť v ich správnom poradí. Zavedenie novej polozky teda vyvolá určité posuvy.

Ak použijeme odhad I (3.1.14) na výpočet indexu pre naše kľúče, dostaneme takéto uloženie kľúčov:

Kľúč	Vypočítaný index I	Miesto: M	Kľúče
503	10	0	052
092	1	1	092
515	10	2	167 ← 152
052	0	3	↓
920	20	4	
167	2	5	270
890	19	6	
270	5	7	
632	13	8	425
425	8	9	
152	2	10	503 515
610	12	11	↓
680	14	12	610
		13	632
		14	680
		15	
		16	
		17	
		18	
		19	890
		20	920

Obr. 3.1.14
Triedenie výpočtom adries

Za predpokladu, že kľúče sú rovnomerne rozložené, výpočtová zložitosť tejto metódy je O(N).

3.1.3.7 Porovnanie metód

Ukázali a opísali sme si celý rad metód vnútorného triedenia. Je teda celkom prirodzené, že si položíme otázku: "Ktorá metóda vnútorného triedenia je najlepšia?" Na túto otázku neexistuje jednoznačná odpoveď. Problém je totiž v tom, že najskôr by sme museli presne definovať, čo rozumieme pod slovom "najlepšia" a neexistuje vlastne ani najlepší spôsob definovania tohto slova.

V oblasti ohodnotenia metód triedenia sa dosiahli zaujímavé a dôležité výsledky, ale mnohé súvislosti sú zatiaľ neodhalené a v konečnom dôsledku nevyhodnotené. Skúmanie vnútorného mechanizmu týchto metód viac-menej vždy smerovalo k minimalizácii počtu porovnaní kľúčov pri triedení n-prvkov, zlučovaní m a n prvkov alebo výbere i-teho najväčšieho prvku z neutriedenej množiny prvkov.

Prebrali sme tieto rôzne metódy triedenia: vkladaním, výmenou, výberom, zlučovaním, grupovaním, radixove-výmenné, výpočtom adries, Shellovo a Quicksort. Každá z nich sa dá charakterizovať takto:

Typ	Časová zložitosť	Doplňujúca pamäť
Vkladaním	$O(N^2)$	žiadna
Výmenou	$O(N^2)$	žiadna
Výberom	$O(N^2)$	žiadna
Zlučovaním dvojcestné	O(N log N)	2N
Grupovaním	$O(N \log_Z R)$	NxZ
Výmenné radixove	O(N log N)	R+1
Výpočtom adries	O(N)	2,2xN(približne)
Shellovo	$O(\alpha . N(\log N))^2$	
	event. O(N log N)	žiadna
Quicksort	O(N log N)	

Obr. 3.1.15
Zložitosť metód vnútorného triedenia

N je rozmer triedeného bloku,
R - maximálna dĺžka (počet cifier) kľúča,
Z - základ (radix) radixového triedenia,
α - konštanta úmerná N.

Všeobecne môžu komparatívne triedenia vyžadovať čas úmerný približne od N^2- - pre najhorší prípad a N log N - pre priemerný prípad. Sú nezávislé od rozloženia položiek, vhodne používajú prirodzené usporiadanie prvkov a vo všeobecnosti nepotrebujú doplňujúcu pamäť.

Distributívne triedenia vyžadujú čas úmerný $N \log_Z R$. Vo všeobecnosti sú nezávislé od rozdelenia vstupných údajov okrem rýchleho triedenia. Často vyžadujú doplňujúcu pamäť.

Triedenie výpočtom adries pracuje dostatočne rýchle na začiatku, resp, s menším počtom prvkov, pretože nevznikajú konflikty medzi adresami. Počas triedenia sa však čas zvyšuje exponenciálne.

Na základe vyhodnotenia metód a skúseností z praxe môžeme odporučiť takéto používanie metód:

a. Triedenie vkladaním je najjednoduchšia metóda z hľadiska programovania, nevyžaduje si doplňujúcu pamäť a je efektívna pre malé N ($N \leq 25$).
b. Triedenie výmenou je v podstate vzhľadom na vlastnosti a uplatnenia rovnaké ako vkladaním.
c. Triedenie výberom - podobne ako v predchádzajúcom prípade.
d. Dvojcestné zlučovanie je rýchla metóda so zložitejším algoritmom a doplňujúcou pamäťou.
e. Grupovaním - je účinné triedenie z hľadiska časového, ale vyžaduje si veľa doplňujúcej pamäti, používa sa skôr pre triedenie štítkov.
f. Radixove-výmenné je veľmi rýchle triedenie, vyžaduje si relatívne málo doplňujúcej pamäti, ale je náročné na naprogramovanie a odladenie.
g. Triedenie s výpočtom adries je efektívne vtedy, ak sú kľúče rovnomerne rozdelené, inak potrebuje veľa doplňujúcej pamäti.
h. Shellovo triedenie sa relatívne jednoducho programuje, potrebuje minimálnu pamäť a efektívne je len po určité N ($N \leq 1000$).
i. Rýchle triedenie sa dá označiť ako univerzálna metóda vnútorného triedenia. Vyžaduje si minimum doplňujúcej pamäti a predstihuje svojich konkurentov (zrovnateľné metódy v časovej zložitosti) v priemernom potrebnom čase - na väčšine počítačov.

Vidíme, že každá metóda v daných súvislostiach - čas, rozsah, kapacita pamäti, čas na naprogramovanie a odladenie - má, alebo nemá opodstatnenie pre jej použitie.

3.1.4 Vonkajšie triedenie. Základné pojmy.

Vieme už, že súbory sú určitým spôsobom organizované, ovládané, udržiavané. Hoci organizácia súboru sa stáva závislejšia od ukladania podľa výpočtov, neklesá význam triedenia. Súbory sa udržiavajú, aby poskytovali informácie, ktoré vyžadujeme v "prirodzenom" usporiadaní. Bez ohľadu na organizáciu súborov v počítači, vety výstupného súboru musia byť v lexikografickom usporiadaní hodnôt podľa určitého kľúča.

V základnom rozdelení sme metódy triedenia rozdelili na dve skupiny - vnútorné a vonkajšie. Vnútorné, pri ktorých je základným predpokladom, že všetky triedené záznamy sa zmestia do operačnej pamäti počítača, a ak sa nezmestia, použijeme niektorú z metód tzv. vonkajšieho triedenia. Tieto dve skupiny metód majú spoločné snáď len to slovo "triedenie" a pochopiteľne jeho dôsledok. Metódy vonkajšieho triedenia sa musia v princípe líšiť. Výnimku tvorí princíp zlučovania, pri ktorom sme uviedli (už pri vnútornom triedení), že má väčší význam a viac alternatív práve pri vonkajšom triedení.

Z hľadiska efektívnosti práce nad súbormi je limitujúcim faktorom doba prístupu k jednotlivým záznamom. Preto vlastnosti samotného média, na ktorom je súbor uložený, ovplyvňujú organizáciu súboru a procedúry jeho spracovania. Ak máme napr. sekvenčne vytriedenú množinu záznamov vo vnútornej pamäti, pre vyhľadanie určeného záznamu je vhodné logaritmické vyhľadávanie. Ak by sme túto metódu použili napr. pri súbore, uloženom na magnetickej páske, tak určite by sme sa pobavili pohľadom na neustále sem a tam prevíjajúcu sa pásku, ale to by bolo všetko. Vieme už, ako sú organizované súbory na magnetických páskach.

Prístup k n-tému záznamu je cez všetkých n-1 záznamov. Magnetické disky a bubny si vyžadujú vzhľadom na priamy prístup tiež inú stratégiu. Pre tieto odlišnosti sa musia vyberať aj rôzne triediace algoritmy pre rôzne pamäťové média.

Napriek predpovediam, že disky postupne nahradia pásky, si vďaka rozvoju technológie magnetických pások a ich cenovej dostupnosti pásky udržujú prvenstvo predovšetkým v aplikáciách, kde sa súbory spracúvajú sekvenčne.

Metódy triedenia, používané pri triedení na magnetických páskach, ako základný princíp používajú zlučovanie. Podstatu zlučovania a dve základné varianty zlučovania sme uviedli v kapitole 3.1.3.4. Nadviažeme na túto kapitolu s tým, že uvažované postupnosti budeme mať na dvoch alebo viacerých páskach.

3.1.4.1 Prirodzené a vyvážené triedenie zlučovaním

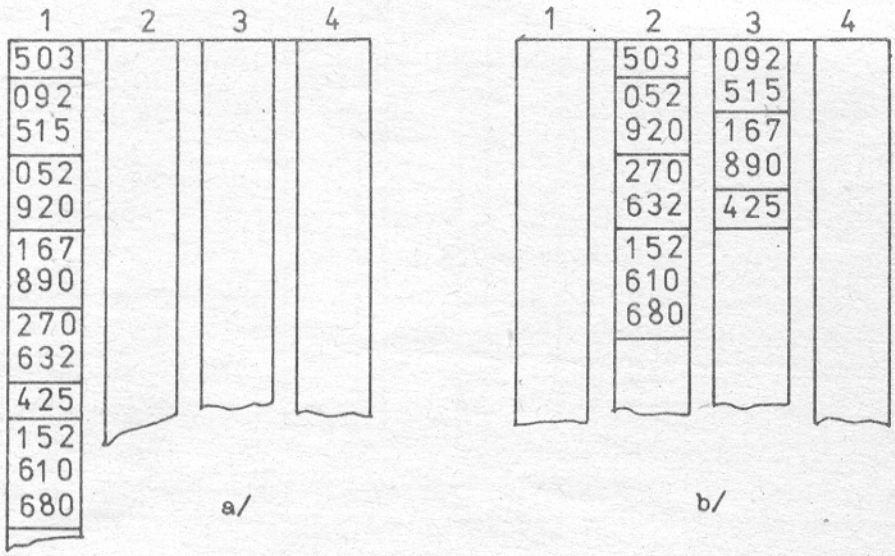
V kapitole 3.1.3.4 sme si uviedli dve varianty základného triedenia zlučovaním - prirodzené a jednoduché dvojcestné zlučovanie. Pre jednoduché sa pri vonkajšom triedení zaužíval názov vyvážené alebo balansované. Tento názov je odvodený zo spôsobu definovania a vytvárania postupnosti triedených prvkov. Na začiatku má každá postupnosť dĺžku 1, po prvej fáze - dĺžku 2, po druhej fáze 4 atď. Všetky postupnosti okrem jednej sú zhodné veľkosťou.

Pri vnútornom zlučovaní sme postupovali z dvoch koncov triedenej postupnosti: vyberali sme usporiadané podúseky a ukladali raz zo začiatku a potom z konca vektora.

Pri vonkajšom triedení máme k dispozícii magnetické pásky. V prípade prirodzeného a vyváženého dvojcestného zlučovania potrebujeme štyri pásky.

Prirodzené zlučovanie:

Máme štyri magnetické pásky a na jednej z nich je vstupný súbor:



Obr. 3.1.16
Prirodzené zlučovanie

Vstupný súbor rozdelíme na podsúbory, ktoré oddeľujeme znakom "koniec súboru". Na obr. 3.1.16 je naznačený vodorovnou čiarou. V a je východisková situácia. V b je situácia po prvej fáze, ak je páska č. 3 po prvej fáze prázdna, proces ukončíme. Ak nie, pokračujeme. Pochopiteľne, že pred začiatkom ďalšej fázy treba pásku, ktorá sa uvoľnila - aby sme mali k dispozícii celú jej dĺžku - a pásky, na ktoré sme ukladali usporiadané podsúbory - aby sme sa dostali pred prvý posúbor - previnúť. Prvú pásku vymažeme. V ďalšej