5. ABSTRAKTNÍ DATOVÉ STRUKTURY

Abstraktní řídicí struktury realizující v programu sekvenci, alternativu a iteraci příkazů, se objevily již v jazyce Algol 60 ve formě složeného příkazu, podmíněného příkazu a cyklu. Umožnily výraznější odpoutání algoritmizace od úrovně strojových instrukcí. Přiblížily algoritmizaci úrovni lidského myšlení a znamenaly kvalitativní posun při tvorbě programů.

V oblasti datových struktur byla vazba na technické vlastnosti počítače silnější. Datové struktury, jež nebyly přímo reprezentovatelné v paměti počítače a pro něž nebyly v repertoáru základních strojových instrukcí odpovídající operace, se pro obavy z nízké funkční účinnosti (malá rychlost, velká spotřeba paměti) příliš nepoužívaly.

Zásadní obrat v hodnocení významu datových struktur nastal na základě poznatků o souvislosti datových struktur s efektivností algoritmů a s procesem systematického vytváření programů. Výsledky zkoumání výpočetní složitosti algoritmů ukázaly, že v mnohých případech vede použití nových, zdánlivě složitých datových struktur k vytvoření nových, kvalitativně účinnějších algoritmů. Od datového typu (reprezentujícího množinu datových objektů a množinu operací nad těmito objekty) se potlačením hledisek jak jsou data zobrazena a jak se nad nimi provádějí operace a zdůrazněním co data reprezentují a co s nimi operace provádějí (tedy zdůrazněním vnějšího chování datové struktury) dospívá k abstraktním datovým strukturám (ADS) a k abstraktnímu typu dat (ATD), jako představiteli určité třídy ADS. Abstraktní datový typ je tedy odvozen z datového typu abstrahováním jeho nejvýznamnějších vlastností a pominutím ostatních, především technicko-implementačních stránek reprezentace v paměti a realizace operací konkrétními programovými prostředky. Odsunutí implementační stránky ADS do fáze jejího rozkladu a zejména možnost uzavření této implementace do samostatného programového celku (procedury, modulu) významně zjednodušuje algoritmus pracující s ADS. Jednoduchost, možnost standardizace a adaptibilita takové struktury snižuje námahu při návrhu a tvorbě programu. Dalším krokem zvyšujícím zejména spolehlivost programu jsou prostředky, které dovolí přístup a manipulaci s ADS výhradně prostřednictvím stanovených operací. Tyto prostředky zabrání vědomému i náhodnému přímému přístupu ke všem (tedy i k pomocným) objektům ADS. V té souvislosti se hovoří o "viditelnosti" objektů "na scéně" a o "neviditelnosti" objektů "za scénou".

Vezmeme-li typ integer, jako příklad jednoduchého ATD, pak tento typ je definován množinou hodnot z oboru celých čísel, jichž může nabýt a množinou stanovených operací nad tímto typem. Množinu operací ATD vynucuje především aplikace daného ATD. (Ve známém příkladě, ze základního kurzu programování, ve kterém se měly tisknout všechny mocniny čísla 2 až do mocniny, která měla právě 100 dekadických číslic, nebylo možné použít standardní typ integer pro shora omezený obor celých čísel, který nevyhovoval zadání. Pro nově definovaný ATD "celé kladné číslo" stačily 4 základní operace : ustavení hodnoty "1", násobení číslem 2, zjištění počtu dekadických číslic hodnoty daného typu a tisk hodnoty daného typu). "Za scénou" typu integer zůstává způsob zobrazení celého čísla v paměti, použitý kód a pravidla jeho aritmetiky. Nepřístupné jsou složky struktury (např. jednotlivé bity) i jiné nedefinované operace (např. posun nebo rotace slova, v němž je typ integer zobrazen).

Na závěr tohoto úvodního odstavce lze říci, že proces snižování úrovně abstrakce hierarchické struktury programu jeho postupným rozkladem, který na každé

nižší úrovni zvyšuje implementační závislost, se neoddělitelně týká jak funkční (řídicí) abstrakce, tak abstrakce datové struktury.

5.1. Principy specifikace abstraktních typů dat

Specifikace ATD specifikuje objekty vytvářející datovou strukturu a operace, která lze nad specifikovaným ATD provádět. Existuje více způsobů, jak lze specifikovat ATD. V tomto odstavci budou uvedeny základní metody tak, jak byly shrnuty v [1].

Na metodu specifikace ATD se nejčastěji kladou tyto požadavky :

1. Formálnost metody specifikace musí umožňovat matematicky přesný a jednoznačný zápis specifikace. Umožňuje využívat specifikace při vyšetřování správnosti programů a používat při řešení problémů s ATD matematického aparátu.
2. Obecnost metody umožňuje specifikovat dostatečně širokou třídu typů dat.
3. Abstraktnost má za důsledek, že se specifikují jen podstatné vlastnosti typů dat.
4. Jednoduchost je podstatnou vlastností pro specifikaci základních typů. Složitější typy dat by se měly specifikovat přehledným a strukturovaným způsobem pomocí jednodušších typů dat.

Další požadavky se vztahují na praktické využití specifikací :

5. Implementovatelnost je vlastnost specifikace, která by měla dávat co nejširší možnost implementace jednoho typu dat v rámci jiného typu dat, a která úzce souvisí s požadavkem (3). Čím méně nepodstatných vlastností datového typu specifikujeme, tím širší jsou možnosti jeho implementace.
6. Praktická realizovatelnost požaduje, aby vedle formálního zápisu specifikace existoval i reálný systém, který by automaticky implementoval specifikovaný typ dat.
7. Flexibilnost (adaptibilita) specifikace zaručuje, že malá změna datového typu má za následek malou změnu specifikace.

Většina metod specifikace typů dat chápe strukturu dat jako objekt s jistou (ne vždy nezbytně vnitřní) strukturou, přičemž přístup ke struktuře je možný pouze prostřednictvím jistých "přípustných" operací. Specifikace proto neurčují strukturu dat přímo, ale nepřímo vlastnostmi přípustných operací. Nejtypičtějšími současnými metodami jsou operační a algebraické specifikace ATD.

Operační specifikace sestávají přímo z algoritmů, implementujících jednotlivé operace nad strukturou dat. Vlastnosti těchto operací nejsou popsány explicitně, ale je nutno je abstrahovat analýzou odpovídajících algoritmů. Pro zápis operačních specifikací se používá většinou určitý známý programovací jazyk. Operační specifikace ATD "polynom" byla algoritmy operací INICPOL, PRIDCLEN, PRIRADP, DER, DERIVUJ, STUPEN, VYCISLI a TISUPOL uvedena jako rozsáhlý příklad programování v Pascalu ve skriptech Počítače a programování [7].

Výhodou operační specifikace je skutečnost, že je zapsána obyčejně v jazyku blízkém programátorům. Pro zápis operačních specifikací ATD je vhodný libovolný jazyk dostatečně vysoké úrovně. Nevýhodou operační specifikace je explicitní zavedení způsobu implementace, která nemusí být vždy nejvýhodnější. Operační speci-

fikace vnáší do popisu i pomocné objekty a operace, které mohou ztěžovat vyšetřování vlastností ATD i dokazování správnosti programů, jež s daným ATD pracují.

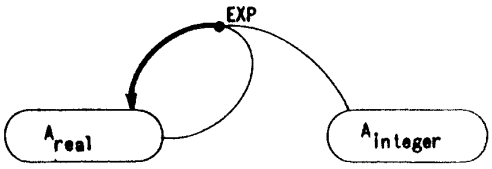
Algebraické specifikace popisují vlastnosti operací pomocí axiomů a neudávají přitom algoritmy, jakými mají být operace implementovány. Axiomy mohou sloužit jako invariantní tvrzení, pomocí nichž lze dokazovat správnost programů. Algebraická specifikace sestává ze dvou částí - ze syntaktické a sémantické specifikace.

Syntaktická specifikace (nebo také signatura ATD) vychází z tzv. typové algebry. Typová algebra je libovolná dvojice $[V,F]$, kde V je konečná množina druhů (tj. určitých množin prvků) a F je množina operací nad prvky jednotlivých druhů. Operaci definujeme jako zobrazení kartézského součinu druhů na jistý druh. Jsou-li např. množinou druhů množiny čísel real (A_{real}) a čísel integer ($A_{integer}$), pak operaci umocnění (EXP) lze syntakticky definovat takto :

$$EXP: A_{real} \times A_{integer} \rightarrow A_{real}$$

Tento zápis definuje pro každou operaci počet a typ vstupních a výstupních argumentů. V množině druhů je vždy jedna množina, o jejíž specifikaci vlastně jde. O ostatních množinách se předpokládá, že jsou známé, nebo že se dají podobným způsobem definovat.

Syntaktickou specifikaci lze graficky velmi názorně vyjádřit diagramem signatury. Diagram sestává z pojmenovaných typů dat vyjádřených ovály se jménem, z pojmenovaných operací vyjádřených vyplněnými kroužky a z argumentů vyjádřených spojnicemi, které spojují vždy kroužek operace s oválem typu. Tenké spojnice představují vstupní argumenty operace, tučné spojnice výstupní argumenty. Spojnice jsou pro větší názornost orientovány šipkou. Příklad diagramu signatury je na obr. 5.1.



Obr. 5.1. Příklad diagramu signatury

Sémantickou část specifikace vytváří množina axiómů. Axiómy vyjadřují vzájemné vztahy mezi operacemi specifikovaného ADT. Alespoň jednou z operací musí být tzv. generátor specifikovaného typu, který jako vstupní argument nepoužívá specifikovaný (generovaný) typ. Generátorem může být tzv. "nulární" operace, která nemá žádný vstupní argument a představuje tudíž konstantu.

Jako příklad algebraické specifikace uveďme specifikaci ATD "POSINT", představující množinu celých kladných čísel, nad níž mají být definovány tyto operace: "ADD" pro součet, "SUCC" pro určení následujícího prvku množiny, generátor "JEDNA" generující hodnotu 1 a predikát "JEJEDNA", jehož Booleovská hodnota je true, je-li vstupním argumentem prvek hodnoty JEDNA.

signatura :

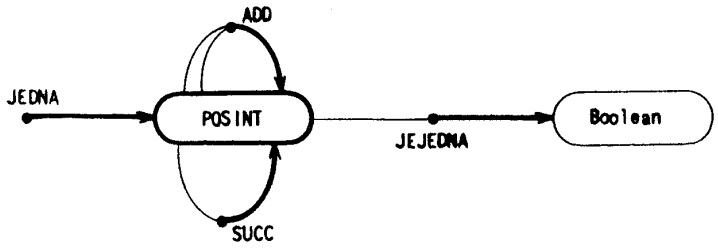
JEDNA : → POSINT
ADD : POSINT x POSINT → POSINT
SUCC : POSINT → POSINT
JEJEDNA : POSINT → Boolean

Grafické vyjádření signatury ATD POSINT je na obr. 5.2.

Tečným oválem označujeme specifikovaný typ.

Axiómy sémantické specifikace jsou tyto :

1. ADD(X,Y)=ADD(Y,X)
2. ADD(JEDNA,X)=SUCC(X)
3. ADD(SUCC(X),Y)=SUCC(ADD (X,Y))
4. JEJEDNA(JEDNA)=true
5. JEJEDNA(SUCC(X))=false



Obr. 5.2. Diagram signatury ATD POSINT

V následujících částech této kapitoly se seznámíme se specifikací některých typických ATD. Pro syntaktickou specifikaci budeme využívat diagramů signatury. Pro definici sémantiky užijeme axiomů, případně jiného vhodného formalismu, doplněného slovním popisem.

5.2. Specifikace typických abstraktních datových typů

5.2.1. Seznam

Seznam (angl. "list") je jednou z nejobecnějších datových struktur, používaných zejména v oblasti nenumerického zpracování. Je to lineární, homogenní, obecně dynamická datová struktura představovaná posloupností jednotlivých prvků vytvářejících seznam. Lineárnost seznamu je dána tím, že ke každému prvku seznamu lze určit jediný bezprostředně následující prvek.

Specifikovat ATD seznam znamená především specifikovat operace nad touto strukturou. Přístup k návrhu množiny operací může být různý, podle toho, jaké vlastnosti typu dat a jaký účel specifikace se akcentuje. Více teoretický přístup specifikuje většinou užší množinu základních operací. Je pak snazší vytvořit pro ně zvládnutelný systém sémantických axiomů a ze základních operací lze vhodnou kompozicí sestavit složitější operace. Praktičtější přístup nebere na zřetel obtíže vystupující s vytvořením sémantických axiómů. Při volbě operací se řídí především požadavky odrážejícími vlastnosti reálného problému, jehož řešení na počítači využívá seznamu jako struktury zobrazující vlastnosti reálných objektů problému. Pro sémantickou specifikaci ATD lze zvolit např. méně formální, ale srozumitelný a přehledný popis účinků jednotlivých operací. Považujme tedy zde uvedenou specifikaci ATD seznamu za jednu z možných specifikací a to takovou, jež se v praxi dobře osvědčila (viz také [8]).

Pro účelné stanovení množiny operací nad lineárním seznamem se definuje zvláštní vlastnost seznamu: seznam buď obsahuje <u>jeden z prvků, který je aktivní</u>, a pak je i seznam aktivní, nebo neobsahuje <u>žádný</u> aktivní prvek a pak je seznam neaktivní. Pro účely sémantické specifikace popisem účinků jednotlivých operací zavedeme tuto notaci :

< >	prázdný seznam (je neaktivní, neobsahuje žádný aktivní prvek)
< A B C >	neaktivní seznam sestávající z posloupnosti prvků A,B a C. (Žádný prvek není aktivní.)
< <u>A</u> B C >	aktivní seznam prvků A,B a C; prvek A je aktivní (je podtržen)
L	libovolný neprázdný neaktivní seznam

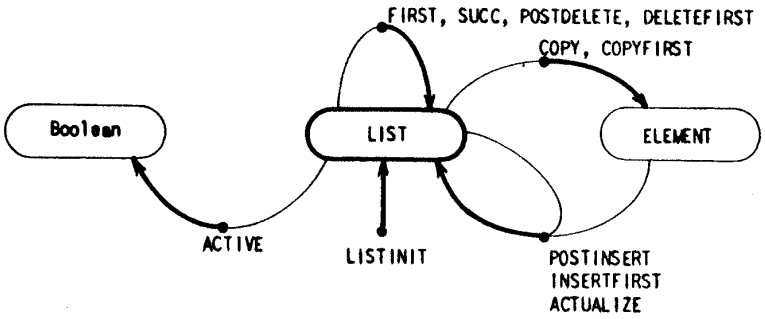
Pozn.: Názvy jednotlivých operací i typů dat vychází z obvyklé anglické terminologie.

Názvy a stručný význam operací je uveden v tab. 5.1.

Název operace	Stručný význam operace
LISTINIT	Vytvoř prázdný seznam
FIRST	Aktivní bude první prvek seznamu
SUCC	Aktivní bude prvek následující za aktivním prvkem
POSTDELETE	Zruš prvek za aktivním prvkem
DELETEFIRST	Zruš první prvek seznamu
COPY	Získej hodnotu aktivního prvku
COPYFIRST	Získej hodnotu prvního prvku
POSTINSERT	Vlož nový prvek za aktivní prvek
INSERTFIRST	Vlož nový prvek na začátek seznamu
ACTUALIZE	Dosaď novou hodnotu do aktivního prvku
ACTIVE	Dotaz, zda seznam obsahuje aktivní prvek

Tab. 5.1. Operace nad seznamem

Syntaktická specifikace ATD seznam je vyjádřena diagramem signatury na obr. 5.3.



Obr. 5.3. Diagram signatury ATD seznam

Sémantickou specifikaci plní dostatečný popis účinků jednotlivých operací v tab. 5.2.

LISTINIT LISTINIT = < >	**DELETEFIRST** DELETEFIRST (< >) = < > DELETEFIRST (< A >) = < > DELETEFIRST (< A >) = < > DELETEFIRST (< AL >) = < L > DELETEFIRST (< AL >) = < L >
FIRST FIRST (< >) = < > FIRST (< A >) = < A > FIRST (< A >) = < A > FIRST (< AL >) = < AL > FIRST (< AL >) = < A L > FIRST (< ABL >) = < ABL >	**COPY** COPY (< >) = error COPY (< L >) = error

FIRST (< AL_1BL_2 >) = < AL_1BL_2 > FIRST (< ALB >) = < ALB >	COPY (< A >) = A COPY (< AL >) = A COPY (< L_1AL_2 >) = A COPY (< LA >) = A
SUCC SUCC (< >) = < > SUCC (< A >) = < A > SUCC (< L >) = < L > SUCC (< ABL >) = < ABL > SUCC (< L_1ABL_2 >) = < L_1ABL_2 > SUCC (< LAB >) = < LAB > SUCC (< LA >) = < LA >	**POSTINSERT** POSTINSERT (A,< >) = < > POSTINSERT (A, < L >) = < L > POSTINSERT (A, < B >) = < BA > POSTINSERT (A, < BL >) = < BAL > POSTINSERT (A, < L_1BL_2 >) = < L_1BAL_2 > POSTINSERT (A, < LB >) = < LBA >
POSTDELETE POSTDELETE (< >) = < > POSTDELETE (< L >) = < L > POSTDELETE (< A >) = < A > POSTDELETE (< AB >) = < A > POSTDELETE (< L_1ABL_2 >) = < L_1AL_2 > POSTDELETE (< LAB >) = < LA > POSTDELETE (< LA >) = < LA >	**INSERTFIRST** INSERTFIRST (A,< >) = < A > INSERTFIRST (A,< L >) = < AL > INSERTFIRST (A,< B >) = < AB > INSERTFIRST (A,< BL >) = < ABL > INSERTFIRST (A,< L_1BL_2 >) = < AL_1BL_2 > INSERTFIRST (A,< LB >) = < ALB >
ACTUALIZE ACTUALIZE (A,< >) = < > ACTUALIZE (A,< L >) = < L > ACTUALIZE (A,< B >) = < A > ACTUALIZE (A,< BL >) = < AL > ACTUALIZE (A,< L_1BL_2 >) = < L_1AL_2 > ACTUALIZE (A,< LB >) = < LA >	**COPYFIRST** COPYFIRST (< >) = error COPYFIRST (< A >) = A COPYFIRST (< A >) = A COPYFIRST (< AL >) = A COPYFIRST (< AL >) = A
	ACTIVE ACTIVE (< >) = false ACTIVE (< L >) = false ACTIVE (< A >) = true ACTIVE (< AL >) = true ACTIVE (< L_1AL_2 >) = true ACTIVE (< LA >) = true

Tab. 5.2. Popis operací ATD seznam

Vlastnosti jednotlivých operací, dostatečně popsaných v tab. 5.2. lze komentovat takto :

- LISTINIT — vytvoří prázdný seznam
- FIRST — nad prázdným seznamem nezpůsobí žádnou změnu; nad neprázdným seznamem způsobí, že aktivním začne být první prvek bez ohledu na to, zda seznam byl aktivní a když ano, který prvek byl aktivní
- SUCC — nad neaktivním seznamem nezpůsobí žádnou změnu; je-li aktivním prvkem poslední (resp. jediný) prvek seznamu, vznikne neaktivní seznam týchž prvků, v jiném případě se aktivita přenese na následující prvek

. POSTDELETE nad neaktivním seznamem nebo seznamem, jehož poslední prvek je aktivní nezpůsobí žádnou změnu; v aktivním seznamu se ze seznamu vypustí prvek za aktivním prvkem

. DELETEFIRST nad prázdným seznamem nezpůsobí žádnou změnu; z neprázdného seznamu vypustí první (resp. jediný) prvek bez ohledu na aktivnost/neaktivnost seznamu; byl-li první (jediný) prvek aktivní je výsledný seznam neaktivní (prázdný).

. COPY nad neaktivním seznamem není definována; nad aktivním seznamem je výsledek operace hodnota aktivního prvku; operace nezpůsobí žádnou změnu v seznamu

. COPYFIRST pokud seznam není prázdný je výsledkem operace hodnota prvního prvku seznamu

. POSTINSERT nad neaktivním seznamem nezpůsobí žádnou změnu; do aktivního seznamu vloží nový prvek za aktivní prvek

. INSERTFIRST vloží do seznamu nový prvek na jeho začátek, aniž se změní aktivnost seznamu

. ACTUALIZE nad neaktivním seznamem nezpůsobí žádnou změnu; v aktivním seznamu změní hodnotu aktivního prvku dosazením hodnoty nového prvku

. ACTIVE je predikát aktivnosti seznamu

Pozn.: Má-li seznam jediný prvek, může se pro vyloučení použít pouze operace DELETEFIRST. Obdobně pro vložení prvního prvku do prázdného seznamu je nutné užít operace INSERTFIRST.

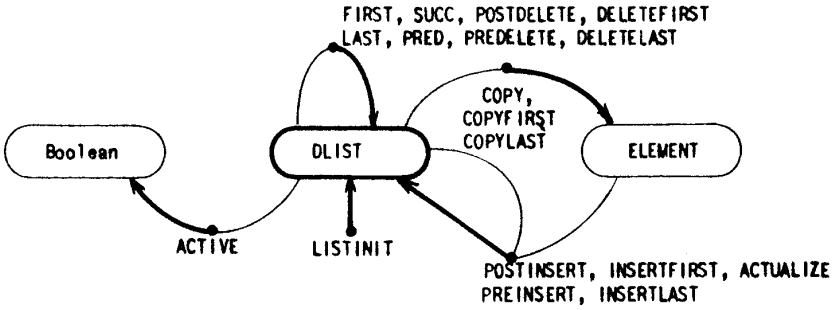
5.2.1.1. Obousměrný seznam

Všechny operace ATD seznam s výjimkou operací LISTINIT, COPY, ACTUALIZE a ACTIVE předurčovaly směr zpracování seznamu zleva doprava (resp. od začátku ke konci) seznamu. Takto specifikovanému seznamu říkáme jednosměrný seznam. Nenáročným rozšířením souboru operací o inverzní operace k operacím směrově závislým, získáme obousměrný seznam. V tab. 5.3. jsou uvedeny názvy původních směrově závislých operací, názvy operací k nim inverzních a jejich stručný význam.

Původní operace	Inverzní operace	Stručný význam inverzní operace
FIRST	LAST	Aktivní bude poslední prvek seznamu
SUCC	PRED	Aktivní bude prvek předcházející původní aktivní prvek
POSTDELETE	PREDELETE	Zruš prvek před aktivním prvkem
DELETEFIRST	DELETELAST	Zruš poslední prvek seznamu
POSTINSERT	PREINSERT	Vlož nový prvek před aktivní prvek
INSERTFIRST	INSERTLAST	Vlož nový prvek na konec seznamu
COPYFIRST	COPYLAST	Získej hodnotu posledního prvku seznamu

Tab. 5.3. Operace pro obousměrný seznam

Syntaktická specifikace ATD obousměrný seznam je vyjádřena diagramem signatury (obr. 5.4.), který vznikl doplněním signatury ATD seznam o operace inverzní ke směrově závislým operacím ATD seznam



Obr. 5.4. Diagram signatury ATD obousměrný seznam (DLIST)

Vytvoření dostatečného popisu účinku inverzních operací ponechejme čtenáři. Sémantika operací LISTINIT, COPY, ACTUALIZE a ACTIVE je obdobná jako u jednosměrného seznamu.

ATD obousměrný seznam je nejobecnější podobou seznamové struktury. Vhodným výběrem a kombinací operací nad obousměrným seznamem lze odvodit některé speciální ATD.

5.2.1.2. Soubor

S typickými vlastnostmi i operacemi ATD soubor jsme se seznámili v odst. 2.3.4. Zavedeme-li pomocnou proměnnou pro režim práce souboru MODE: (WRITING , READING), pak lze typické operace pro ATD soubor odvodit z ATD seznam následujícím způsobem :

```
1. REWRITE (FILE) = LISTINIT(LIST); MODE := WRITING

2. RESET (FILE) = FIRST (LIST); ELEMENT := COPY(LIST); MODE := READING

3. PUT (FILE) = if MODE = READING then error
                                  else
          if ACTIVE (LIST)then begin INSERTLAST (ELEMENT,LIST); SUCC(LIST) end
                          else begin INSERTFIRST (ELEMENT,LIST); FIRST(LIST) end

4. GET (FILE) = if MODE = WRITING
                then error
                else if EOF (FILE) then error
                                   else
                  begin SUCC (LIST);
                        if not EOF(FILE) then ELEMENT:=COPY(LIST)
                  end;
5. EOF(FILE) =    ACTIVE(LIST)
```

V uvedených sémantických specifikacích je FILE označení souboru, LIST označení seznamu implementujícího soubor a ELEMENT implementuje hodnotu přístupové proměnné souboru (FILE↑).

Jak jsme již uvedli v 2.3.4. lze ATD soubor interpretovat mnoha způsoby a lze jeho vlastnosti ztotožnit s vlastnostmi jedno i obousměrného seznamu. Výše uvedené specifikace popisují vlastnosti souboru, jak je zavedl programovací jazyk Pascal.

Pozn.: V Pascalu je hodnota přístupové proměnné souboru nedefinovaná, následuje-li po operaci REWRITE (FILE) operace RESET (FILE). Důsledkem této situace podle naší specifikace je chyba (error), což vyplývá z vlastností operace COPY.

5.2.1.3. Kruhový seznam

ATD kruhový (cyklický) seznam je nelineární, homogenní, obecně dynamická datová struktura. Sestává-li kruhový seznam z N prvků ($N \geq 1$), pak lze kruhový seznam nahradit nekonečným lineárním seznamem, v němž bude platit

$$Prvek_i = Prvek_{i+kN} \qquad \text{kde} \quad i > 0, \quad K = 1,2, \ldots, \infty$$

Podobně jako seznam, může být i kruhový seznam jednosměrný nebo obousměrný.

ATD obousměrný kruhový seznam o N prvcích, lze odvodit z lineárního obousměrného seznamu o N prvcích zavedením platnosti tvrzení :

SUCC (LAST (LIST)) = FIRST (LIST)

a

PRED (FIRST (LIST)) = LAST (LIST)

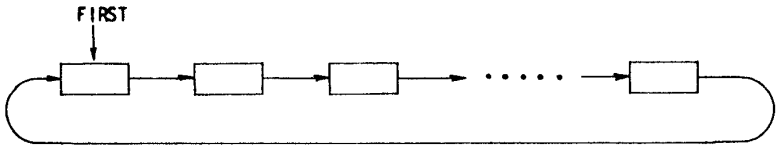
a promítnutím platnosti těchto tvrzení do vlastností všech směrově závislých operací.

Jednosměrný kruhový seznam lze z jednosměrného lineárního seznamu odvodit změněným popisem především operací u těchto případů :

SUCC (< ALB >) = < ALB >
POSTDELETE (< ALB >) = < LB >

Obecně nemají u kruhového seznamu smysl operace, které se vztahují k počátku či konci, protože kruh obecně nemá začátek ani konec. Z praktického hlediska však lze na kruhu vyznačit významný bod (v kruhovém seznamu významný prvek), ke kterému se vztahují operace závislé na začátek a konec.

Kruhový seznam a jeho odvození z lineárního seznamu je znázorněno na obr. 5.5.



Obr. 5.5. Znázornění kruhového seznamu

5.2.1.4. Vyšší operace nad seznamy

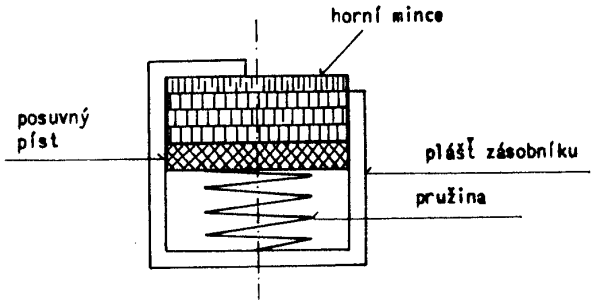
Pro úplnost uveďme některé typické vyšší operace nad seznamy, které lze odvodit ze základních operací. Jsou to :

- Sloučení dvou nebo více seznamů do jednoho seznamu (konkatenace)
- Rozdělení jednoho seznamu na dva nebo více seznamů (dekatenace)
- Zjištění počtu položek (prvků) seznamu

- Zjištění, zda se v seznamu nachází položka s danou hodnotou své složky (klíče) -(vyhledávání angl. searching)
- Vytvoření kopie seznamu
- Uspořádání (seřazení) položek seznamu podle hodnoty určité složky položek (podle klíče)
- Relace uspořádání nad dvěma (stejně dlouhými) seznamy podle hodnot klíčů položek.(Princip této operace je stejný jako princip relace nad dvěma textovými řetězci v Pascalu.)

5.2.2. Zásobník

ATD zásobník (angl. stack) je homogenní, lineární, obecně dynamický typ, který nachází široké uplatnění v programovacích technikách a jež určuje podstatu řady významných algoritmů. Vlastnosti zásobníku jsou velmi podobné řadě objektů, které mají v praktickém životě stejné jméno (např. zásobník nábojů pro ruční zbraň, zásobník korunových mincí pro osobu často telefonující z veřejného automatu aj.). Na obr. 5.6. je znázorněn zásobník mincí, u kterého lze snadno odvodit charakter základních "operací".



Obr. 5.6. Zásobník mincí

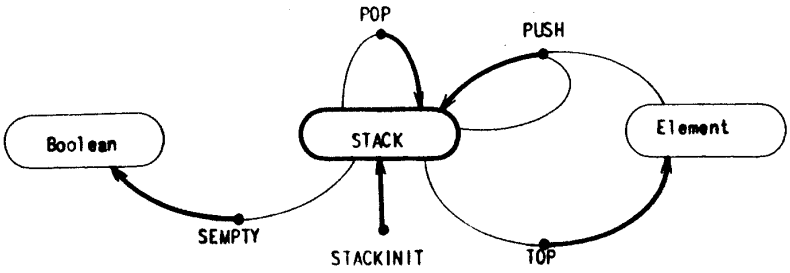
Neprázdný zásobník má na vrcholu jednu minci, tlačenou k zarážce vrcholu pružinou. Prázdný zásobník má na vrcholu posuvný píst (na první pohled rozlišitelný od mince) tlačený k vrcholu pružinou. Do zásobníku lze vložit na jeho vrchol novou minci (stlačením horní mince neprázdného, nebo pístu prázdného zásobníku). Bylo-li v zásobníku více než jedna mince, ocitne se na vrcholu zásobníku po vybrání mince ta mince, která byla pod vybíranou mincí. Pro okamžitou potřebu není k dispozici (není přístupná) žádná mince s výjimkou mince na vrcholu. Mince se postupně ze zásobníku vybírají v obráceném pořadí než v jakém byly do zásobníku vkládány. Tuto skutečnost charakterizuje také anglická zkratka LIFO ("Last-in, first-out") česky "poslední tam, první ven"), která se také používá k označení takových struktur. Budeme-li do zásobníku postupně ukládat údaje o uzlových místech cesty určitým prostorem, umožní nám postupné vybírání údajů zpětnou cestu po stejné trase.

Pro ATD zásobník jsou tedy charakteristické tyto operace uvedené v tab.5.4.

Název operace	Stručný význam operace
STACKINIT	Vytvoř prázdný zásobník
PUSH	Vlož nový prvek na vrchol zásobníku (z anglického Push-down = zasuň, zastrč)
POP	Odeber (zruš) prvek na vrcholu zásobníku (z anglického pop-up = vyjmi)
TOP	Získej hodnotu prvku na vrcholu zásobníku
SEMPTY	Dotaz zda je zásobník prázdný

Tab. 5.4. Operace ATD zásobník

Syntaktická specifikace ATD zásobník je vyjádřena diagramem signatury na obr. 5.7.



Obr. 5.7. Diagram signatury ATD zásobník

Pozn.: Při praktickém využívání zásobníku je často výhodnější taková operace POP, která má také vlastnosti operace TOP tzn., že odstraní prvek ze zásobníku a získá jeho hodnotu.(Mince vybírané ze zásobníku většinou nezahazujeme.) Z hlediska formálních manipulací se sémantickou specifikací je však výhodnější uvedený tvar operace.

Sémantická specifikace je uvedena formou dostatečného popisu účinků jednotlivých operací v tab. 5.5. i systémem axiomů v tab. 5.6. V tab. 5.6. představuje S operand typu zásobník (stack) a E operand typu prvek (element).

STACKINIT STACKINIT = < >	TOP TOP (< >) = error TOP (<A>) = A TOP (<A,L>) = A
PUSH PUSH (A, < >) = < A > PUSH (A,<L>) = < AL >	SEMPTY SEMPTY (< >) = true SEMPTY (<A>) = false SEMPTY (<L>) = false
POP POP (< >) = < > POP (<A>) = < > POP (<AL>) = < L >	

Tab. 5.5. Popis operací ATD zásobník

POP (STACKINIT) = STACKINIT	POP (PUSH(E,S))=S
TOP (STACKINIT) = error	SEMPTY (STACKINIT) = true
TOP (PUSH(E,S)) = E	SEMPTY (PUSH(E,S)) = false

Tab. 5.6. Axiomy sémantické specifikace ATD zásobník

ATD zásobník lze z ATD seznam snadno odvodit těmito pravidly :

1. STACKINIT (STACK) = LISTINIT (LIST)

2. PUSH (ELEMENT,STACK) = INSERTFIRST (ELEMENT,LIST); FIRST (LIST)
{prvek na vrcholu je vždy aktivní}

3. POP (STACK) = SUCC (LIST); DELETEFIRST (LIST)

4. TOP (STACK) = COPY (LIST)

5. SEMPTY (STACK) = not ACTIVE (LIST)

5.2.3. Fronta

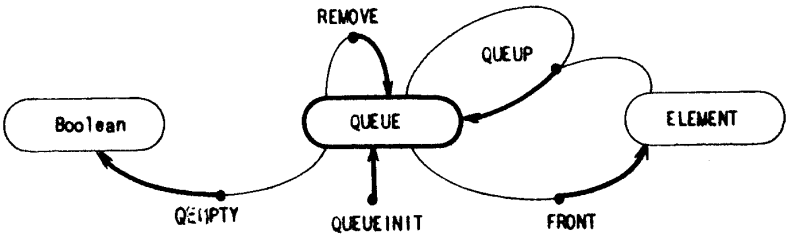
ATD fronta (anglicky queue - čti "kjú") je homogenní, lineární a dynamický typ odvozený od pojmu všeobecně známého z praktického života. Nachází uplatnění především v úlohách z oblasti "hromadné obsluhy", do kterých spadají i některé problémy, které jsou součástí operačních systémů aj. Na rozdíl od zásobníku, označuje se fronta také jako struktura FIFO ("First-in, first-out" - česky "první tam, první ven"). Na rozdíl od zásobníku, který manipuluje s prvky pouze na jednom konci lineární struktury, fronta umožňuje manipulaci s prvky na obou jejich koncích. Na jednom konci se vkládají nové prvky do fronty (konec fronty) a z druhého konce je možné odstraňovat prvky resp. získávat jejich hodnotu (začátek fronty).

Pro ATD fronta jsou charakteristické operace uvedeny v tab. 5.7.

Název operace	Stručný význam operace
QUEUEINIT	Vytvoř prázdnou frontu
QUEUP	Vlož nový prvek na konec fronty
FRONT	Získej hodnotu prvku na začátku fronty
REMOVE	Odstraň prvek ze začátku fronty
QEMPTY	Dotaz, zda je fronta prázdná

Tab. 5.7. Operace ATD fronta

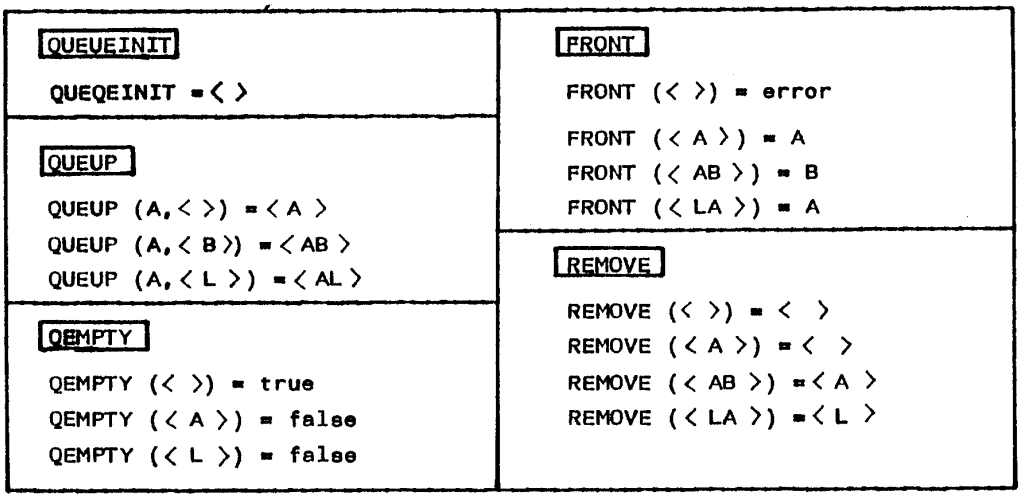
Syntaktická specifikace ATD fronta je vyjádřena diagramem signatury na obr. 5.8.



Obr. 5.8. Diagram signatury ATD fronta

Pozn.: Podobně jako u zásobníku je při praktickém využívání často výhodnější taková operace REMOVE, která má také vlastnosti operace FRONT, tzn., že odstraní prvek ze začátku fronty a získá jeho hodnotu.

Sémantická specifikace je uvedena popisem účinků jednotlivých operací v tab. 5.8. a systémem sémantických axiomů v tab. 5.9. V tab. 5.9. představuje Q operand typu fronta (queue) a E operand typu prvek (element).



Obr. 5.8. Popis operací ATD fronta

QEMPTY (QUEUEINIT(Q)) = true
QEMPTY (QUEUP(E,Q)) = false
FRONT (QUEUEINIT(Q)) = error
FRONT (QUEUP(E,QUEUEINIT(Q)) = E
FRONT (QUEUP(E_1,QUEUP(E_2,Q))) = FRONT(QUEUP(E_2,Q))
REMOVE (QUEUEINIT(Q)) = QUEUEINIT(Q)
REMOVE (QUEUP(E,QUEUEINIT(Q)) = QUEUEINIT(Q)
REMOVE (QUEUP(E_1,QUEUP(E_2,Q))) = QUEUP(E_1,REMOVE(QUEUP(E_2,Q)))

Tab. 5.9. Axiomy sémantické specifikace ATD fronta

ATD fronta lze vyjádřit snadno pomocí ATD seznamu

1. QUEUEINIT(Q) = LISTINIT(L)

2. QUEUP(E,Q) = if ACTIVE (L) then { neprázdná fronta }
begin POSTINSERT(E,L); Ssucc(L)
end { poslední je vždy aktivní }
else { prázdná fronta }
begin INSERTFIRST (E,L); FIRST (L)
end

3. FRONT(Q) = COPYFIRST(L)

4. REMOVE(Q) = DELETEFIRST(L) { Odstraňuje se vždy první }

5. QEMPTY(Q) = not ACTIVE (L)

kde Q .... fronta (Queue)
L .... seznam (List)
E .... prvek (Element)

5.2.3.1. Oboustranně ukončená fronta

ATD oboustranně ukončená fronta (DEQUE - Double Ended Queue) je takový typ fronty, který umožňuje vkládání i vybírání prvků fronty na jejich obou koncích. I tento problém má řadu praktických aplikací (rozřazování a seřazování vagónů, hra Domino aj.).

Pro ATD oboustranně ukončená fronta jsou charakteristické operace uvedeny v tab. 5.10.

Název operace	Stručný význam operace
DEQUEINIT	Vytvoř prázdnou frontu
QUEUEFIRST	Vlož nový prvek na začátek fronty
QUEUELAST	Vlož nový prvek na konec fronty
REMOVEFIRST	Odstraň prvek ze začátku fronty
REMOVELAST	Odstraň prvek z konce fronty
DCOPYFIRST	Získej hodnotu prvního prvku
DCOPYLAST	Získej hodnotu posledního prvku
DEMPTY	Dotaz, zda je fronta prázdná

Tab. 5.10. Operace ATD oboustranně ukončená fronta

Protože popsané operace jsou velmi podobné operacím dříve uvedených ATD, ponecháme jejich specifikaci čtenáři.

5.2.4. Pole

ATD pole (angl. array) je homogenní, lineární datovou strukturou. Složkám (prvkům, elementům) pole jsou přiřazeny navzájem různé hodnoty indexů. Nad množinou hodnot indexů musí existovat uspořádání. Složky pole jsou lineárně uspořádány podle vzrůstající hodnoty indexů přiřazených jednotlivým složkám.

Datový typ indexu může být libovolným ordinálním typem, datový typ prvku pole je libovolný.

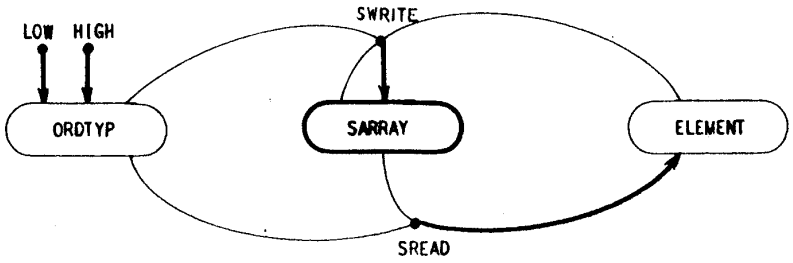
Přístup k prvkům pole je určen udáním hodnoty indexu a není závislý na přístupu k prvku jinému. Proto říkáme, že pole je strukturou s přímým nebo náhodným přístupem (random access).

Počet prvků pole může být pevně určen definicí datového typu nebo se může měnit v době zpracování. V prvním případě nazýváme pole statickým ve druhém dynamickým. Statické pole je známo z jazyka PASCAL. Pokud existuje datový typ ORDTYP libovolného ordinálního typu, můžeme zobrazit diagram signatury statického pole (SARRAY) viz obr. 5.9.

Typ indexu ORDTYP vyváží generátory LOW a HIGH určující rozsah indexů pole. Uvažujeme-li pole tak, jak je známé z PASCALu, nejsou nad ním nutné žádné další operace.

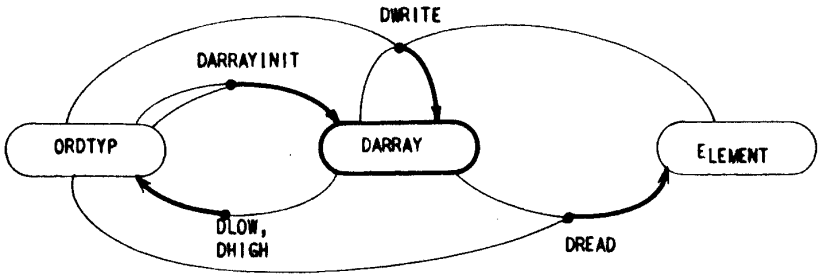
Sémantika operací READ a WRITE je dána sémantikou operací přístupu k prvkům pole známou z PASCALu.

Pro dynamické pole je typická operace inicializace DARRAYINIT určující rozsah hodnot indexů. Množina všech povolených hodnot indexů (z nichž je operací



Obr. 5.9. Diagram signatury statického pole

inicializace vybrán jistý interval) je obdobně jako u statického pole dána datovým typem indexu ORDTYP. Diagram signatury ATD dynamické pole je na obr. 5.10.



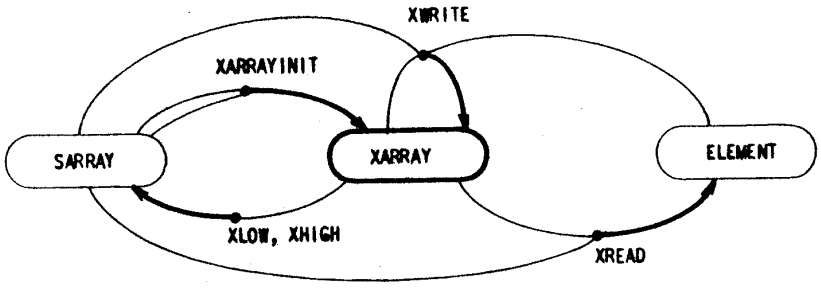
Obr. 5.10. Diagram signatury dynamického pole

Operace LOW a HIGH v tomto případě určují inicializovanou minimální a maximální hodnotu indexu. Sémantiku operací dynamického pole lze určit těmito axiomy :

DREAD (DARRAYINIT(M,N),I)=error
DREAD(DWRITE(A,I,D),J) = if (I=J) then D else DREAD (A,J)
DWRITE(A,I,D) = if (I ≥ DLOW(A)) ∧ (DHIGH(A) ≥ I) then
DWRITE(A,I,D)
else
error
DLOW(DARRAYINIT(M,N))=M
DHIGH(DARRAYINIT(M,N))=N
DLOW(DWRITE(A,I,D))=DLOW(A)
DHIGH(DWRITE(A,I,D))=DHIGH(A)

Obě doposud uvažované varianty pole jsou indexovány hodnotami ordinálního typu. Připustíme-li možnost indexace pole vektorem hodnot ordinálního typu (tj. statickým polem, jehož typ ELEMENT je ordinálního typu), pak takový datový typ nazýváme vícerozměrným polem. Počet složek vektoru pro indexaci určuje počet rozměrů pole. Diagram signatury vícerozměrného pole je na obr. 5.11.

Existuje samozřejmě i vícerozměrná varianta statického pole. Významy operací nad vícerozměrným polem jsou obdobné významům operací jednorozměrného pole. Pokud bychom definovali operace = a ≥ nad vektorem indexů, zůstala by axiomatická definice shodná s axiomatickou definicí dynamického jednorozměrného pole.



Obr. 5.11. Diagram signatury vícerozměrného pole

5.2.5. Tabulka

Pro ATD tabulka (angl. look-up table) by se měl raději používat název "vyhledávací tabulka", protože význam názvu "tabulka" bývá často zaměňován s významem nějakého druhu dvourozměrného pole (matice). ATD tabulka je homogenní, obecně dynamická struktura, jejíž funkce i operace jsou odvozeny od objektu, kterému v praxi nejčastěji říkáme "kartotéka". Každá položka tabulky (kartotéky) obsahuje složku, která plní funkci tzv. klíče. Hodnotou klíče lze jednoznačně identifikovat každou položku; zásadní vlastností tabulky tedy je, že obsahuje položky, hodnoty jejichž klíčů jsou navzájem různé.

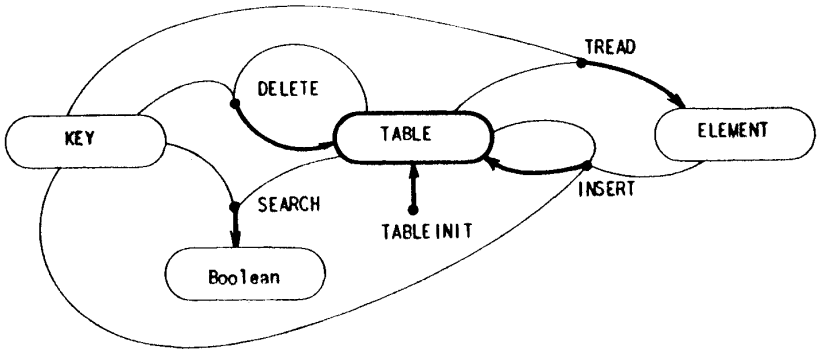
Pro ATD tabulka jsou v tab. 5.11. uvedeny charakteristické operace

Název operace	Stručný význam operace
TABLEINIT	Vytvoř prázdnou tabulku
INSERT	Vlož nový prvek s daným klíčem do tabulky; Je-li již v tabulce prvek s tímto klíčem, nahraď jej (přepiš) novým prvkem
TREAD	Získej hodnotu prvku, jehož klíč se rovná zadané hodnotě
DELETE	Vyřaď z tabulky prvek, jehož klíč se rovná zadané hodnotě
SEARCH	Dotaz, zda tabulka obsahuje prvek, jehož klíč se rovná zadané hodnotě

Tab. 5.11. Operace ATD tabulka

Syntaktická specifikace ATD tabulka je vyjádřena diagramem signatury na obr. 5.12.

Axiomy sémantické specifikace ATD tabulka jsou uvedeny v tab. 5.12. Symbol T představuje operand typu tabulka, K operand typu klíč (key) a E operand typu prvek.



Obr. 5.12. Diagram signatury ATD tabulka

1.	SEARCH (K,**TABLEINIT(T)**) = **false**
2.	SEARCH (K,DELETE(K,T)) = false
3.	SEARCH (K,INSERT(K,E,T)) = true
4.	TREAD (K,**TABLEINIT(T)**) = **error**
5.	TREAD (K,DELETE(K,T)) = error
6.	TREAD (K,INSERT(K,E,T)) = E
7.	TREAD (K,INSERT(K,E_1,INSERT(K,E_2,T))) = E_1
8.	DELETE (K,**TABLEINIT(T)**) = **TABLEINIT(T)**
9.	DELETE (K,INSERT(K,E,INIT) = **TABLEINIT(T)**
10.	DELETE (K,INSERT(K,E,T) = T
11.	DELETE (K,T) = if SEARCH(K,T) then DELETE (K,T) else T

Tab. 5.12. Sémantická specifikace ATD tabulka

Stručně lze komentovat sémantickou specifikaci těmito slovními pravidly :

1. Operace TABLEINIT vytvoří prázdnou tabulku.
2. Operace čtení prvku (TREAD), který není v tabulce obsažen, má za následek chybu; je-li v tabulce obsažen, získá operace jeho hodnotu.
3. Operace zrušení prvku (DELETE), který není v tabulce obsažen, nezpůsobí žádnou změnu v tabulce; je-li prvek v tabulce obsažen, operace jej z tabulky vyřadí.
4. Operace zápisu prvku do tabulky (INSERT), způsobí zařazení tohoto prvku do tabulky v případě, že tabulka neobsahuje prvek se stejnou hodnotou klíče; obsahuje-li tabulka prvek se stejnou hodnotou klíče, způsobí operace přepsání (nahrazení, aktualizaci) starého prvku novým.
5. Operace SEARCH zjišťuje, zda tabulka obsahuje prvek s danou hodnotou klíče, a podle výsledku má hodnotu true je-li prvek přítomen a false, není-li prvek v tabulce obsažen.

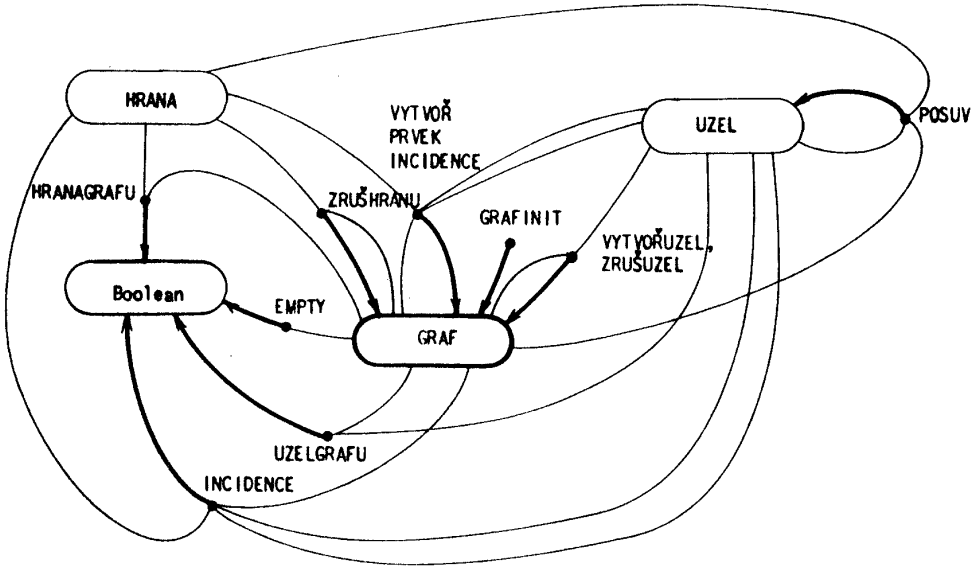
5.2.6. Nelineární struktury

5.2.6.1. Graf

Společnou vlastností všech předcházejících abstraktních datových typů (s výjimkou tabulky) byla lineárnost uspořádání jejich prvků. Obecnou nelineární strukturou je **graf**. Se základními vlastnostmi grafu jsme se seznámili v předmětu Algebry a grafy (viz také [10]).

Grafem se nazývá uspořádaná trojice $G = \langle H, U, \sigma \rangle$, kde H a U jsou libovolné disjunktní množiny a $\sigma: H \rightarrow U \times U$ je libovolné zobrazení nazývané incidence. Prvky množiny H se nazývají hrany (angl. edge), prvky množiny U se nazývají uzly (angl. node). σ je incidenční relace, která každé hraně $h \in H$ přiřazuje neprázdnou, nejvýše dvouprvkovou množinu $\sigma(n) \subset U$ uzlů, které hrana spojuje. Orientovaný graf se od neorientovaného liší tím, že incidenční relace přiřazuje hraně uspořádanou dvojici uzlů.

Pokusme se specifikovat základní operace nad ATD graf diagramem signatury na obr. 5.13.



Obr. 5.13. Diagram signatury ATD graf

Základní operace ATD graf jsou přehledně uvedeny v tab. 5.13.

Název operace	Stručný význam operace
GRAFINIT	Vytvoří se prázdný graf
VYTVOŘ_UZEL	Nový uzel se stane prvkem grafu
ZRUŠ_UZEL	Daný uzel přestane být prvkem grafu
VYTVOŘ_PRVEK_INCIDENCE	Dva dané uzly grafu jsou spojeny hranou
ZRUŠ_HRANU	Zruší se daná hrana grafu
POSUV	Z daného uzlu grafu se po dané hraně grafu získá uzel grafu
EMPTY	Dotaz, zda je graf prázdný
HRANAGRAFU	Dotaz, zda je daná hrana hranou grafu

UZELGRAFU	Dotaz, zda je daný uzel uzlem grafu
INCIDENCE	Dotaz, zda daná hrana inciduje s danými dvěma uzly

Tab. 5.13. Tabulka operací ATD graf

Specifikujme sémantiku uvedených operací těmito slovními pravidly :

1. Operace GRAFINIT vytvoří prázdný graf.
2. Operace VYTVOŘ_UZEL způsobí, že (ohodnocený) uzel daného typu, se stane novým izolovaným uzlem grafu.
3. Operace ZRUŠ_UZEL způsobí, že izolovaný uzel grafu, který je zadán jako operand, přestane být prvkem grafu. Situace, kdy operandem není izolovaný uzel grafu, vede k chybovému stavu.
4. Operace VYTVOŘ_PRVEK_INCIDENCE způsobí, že dva dané uzly grafu jsou spojeny hranou. (Opakování operace nad stejnou dvojicí uzlů vede k vytvoření multigrafu). Jsou-li oba uzly dvojice identické, vytvoří se smyčka.
5. Operace ZRUŠ_HRANU způsobí, že daná hrana grafu přestane být prvkem grafu.
6. Operace POSUV je základní operací pro realizaci orientované cesty grafem. Ze zadaného uzlu grafu se po zadané hraně získá druhý uzel, s nímž tato hrana inciduje. Situace, kdy první uzel není prvkem množiny $\sigma(h)$, kde h je zadaná hrana, vede k chybovému stavu.
7. Operace EMPTY zjišťuje, zda je graf prázdný. Je-li graf prázdný má operace Booleovskou hodnotu true, jinak hodnotu false.
8. Operace HRANAGRAFU zjišťuje, zda je daná hrana hranou grafu. Výsledek má Booleovskou hodnotu.
9. Operace UZELGRAFU zjišťuje, zda je daný uzel uzlem grafu. Výsledek má Booleovskou hodnotu.
10. Operace INCIDENCE zjišťuje, zda daná hrana inciduje s danými dvěma uzly. Výsledek má Booleovskou hodnotu.

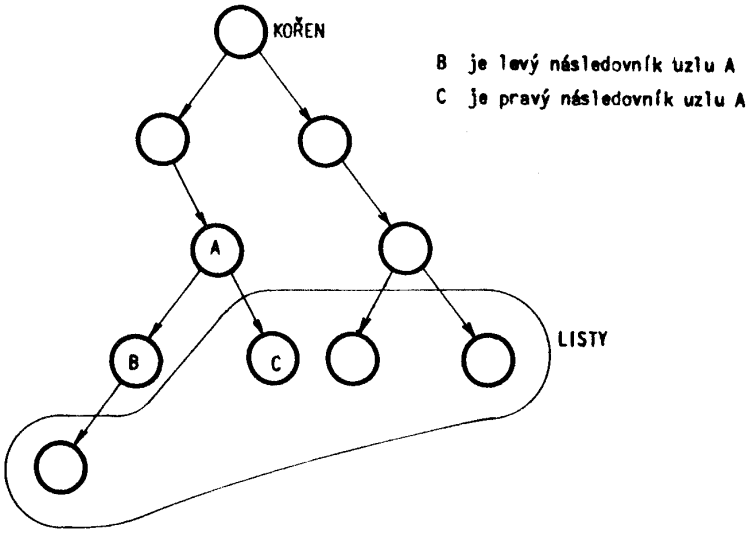
5.2.6.2. Binární strom

ATD binární strom (angl. binary tree) má pro předmět Programovací techniky, ale i pro širší využití v oblasti programování značný význam. S jeho definicí i s některými operacemi jsme se seznámili v předmětu Algebry a grafy (viz také [10]). Zopakujme jen nejdůležitější definice :

1. Strom je acyklický souvislý graf.
2. Kořenový strom je orientovaný strom, ve kterém každá cesta spojující zvláštní uzel, nazvaný kořen, se všemi ostatními uzly, je orientovaná cesta.
3. Binární strom je orientovaný kořenový strom, jehož každý uzel má výstupní stupeň 0 nebo 2 (pro některé aplikace připustíme, že výstupní stupeň může být nejvýše 2).
4. Uspořádaným binárním stromem nazýváme takový binární strom, v němž je každá dvojice $\langle u,u_1 \rangle$ a $\langle u,u_2 \rangle$, vycházejících z uzlu u jistým způsobem uspořádána.

Z definic vyplývá, že binární strom je zvláštním případem orientovaného grafu. Jeden zvláštní uzel-kořen (angl. root) - je uzlem, který představuje "počátek" stromu. Uzly, které mají výstupní stupeň 0 jsou koncové uzly (angl. terminal node) a říká se jim terminální uzly, nebo také "listy". Ostatní uzly se nazývají "běžné" (vnitřní) uzly. Považujeme-li běžný uzel za kořen jistého stromu, pak tento strom je podstromem (angl. branch nebo subtree) stromu, do

něhož náleží tento uzel.

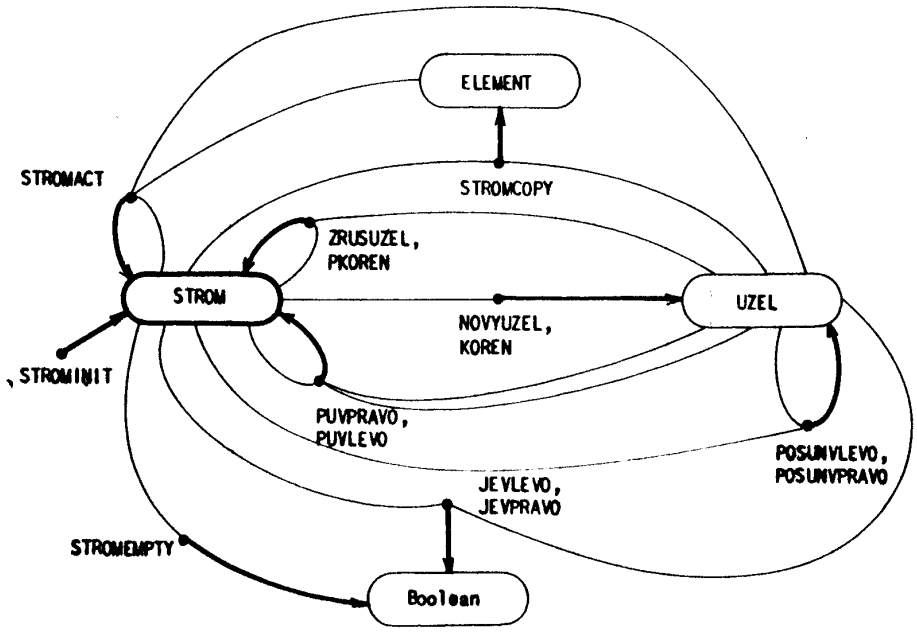


Obr. 5.14. Binární strom

Pro ATD binární strom budeme předpokládat (podobně jako u předchozích lineárních struktur), že uzel stromu může nést hodnotu datového typu ELEMENT. Hrana binárního stromu nemůže být označena. Představuje pouze vazbu mezi uzly.

Pro ATD zvolíme takovou množinu operací, která by umožňovala vytvářet binární strom a dosazovat hodnoty typu ELEMENT do vytvářených uzlů, postupovat binárním stromem od kořene stromu k listům a získávat hodnoty typu ELEMENT z uzlů, případně binární strom rušit.

Syntaktické definice operací nad ATD binární strom jsou znázorněny diagramem signatury na obr. 5.15.



Obr. 5.15. Diagram signatury ATD binární strom

Sémantika operací je stručně shrnuta v tab. 5.14.

Název operace	Stručně popsaný význam
STROMINIT	vytvoření prázdného stromu
STROMEMPTY	predikát vyjadřující skutečnost, že strom je prázdný
NOVYUZEL	vytvoření nového izolovaného uzlu
STROMACT	dosazení hodnoty do uzlu
PUVLEVO	připojení uzlu jako levého následovníka daného uzlu
PUVPRAVO	připojení uzlu jako pravého následovníka daného uzlu
PKOREN	zařazení daného uzlu jako kořene stromu
KOREN	získání kořene stromu
POSUNVLEVO	získání levého následovníka
POSUNVPRAVO	získání pravého následovníka
JEVLEVO	predikát existence levého následovníka
JEVPRAVO	predikát existence pravého následovníka
STROMCOPY	získání hodnoty daného uzlu
ZRUSUZEL	zrušení daného uzlu

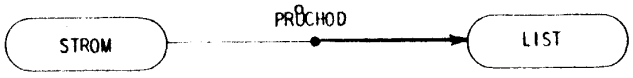
Tab. 5.14. Tabulka operací ATD binární strom

Specifikujme sémantiku uvedených operací těmito slovními pravidly :

1. Operace STROMINIT vytvoří prázdný strom.
2. Operace STROMEMPTY zjišťuje, zda je strom prázdný. Pokud je strom prázdný, má operace hodnotu true, jinak má hodnotu false.
3. Operace NOVYUZEL vytvoří nový izolovaný uzel, který je možné zařadit do stromu operacemi PUVLEVO, PUVPRAVO, PKOREN.
4. Operace STROMACT dosadí určenému uzlu hodnotu datového typu ELEMENT.
5. Operace PUVLEVO,PUVPRAVO připojí uzel jako levého resp. pravého následovníka určeného uzlu.
6. Operace PKOREN zařadí daný uzel jako kořen stromu.
7. Operace KOREN poskytne jako výsledek kořen stromu, pokud existuje. Pokud kořen neexistuje, nastane chyba.
8. Operace POSUNVLEVO resp. POSUNVPRAVO poskytne jako výsledek uzel, jež je levým resp. pravým následovníkem daného uzlu, pokud následovník existuje. Pokud neexistuje nastane chyba.
9. Operace JEVLEVO resp. JEVPRAVO zjišťuje, zda daný uzel má levého resp. pravého následovníka. Pokud následovník existuje, má operace hodnotu true, jinak má hodnotu false.
10. Operace STROMCOPY získá hodnotu typu ELEMENT příslušející danému uzlu.
11. Operace ZRUSUZEL odstraní ze stromu daný uzel, včetně hran z něho vycházejících. Před provedením této operace je tedy nutné vytvořit jiné hrany zachovávající spojitost binárního stromu po zrušení uzlu.

Velmi důležitou operací nad stromem, je tzv. průchod stromem, jehož výsledkem je transformace nelineární stromové struktury na lineární strukturu (seznam).

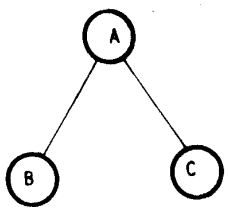
Operaci můžeme znázornit diagramem signatury na obr. 5.16. Pod pojmem "průchod" rozumíme libovolnou posloupnost všech uzlů stromu takovou, v níž se každý uzel stromu vyskytuje pouze jednou. Ze všech možných navzájem různých průchodů jsou významné některé systematické průchody. Jsou to průchody typu preorder, inorder



Obr. 5.16. Diagram signatury operace průchod

a postorder. S těmito průchody jsme se seznámili již v předmětu Algebry a grafy (viz také [10]). Předpona názvu těchto průchodů (pre = před, in = v (mezi), post = po) naznačuje pořadí, ve kterém se zpracovává (prochází) kořen stromu ve vztahu k levému a pravému podstromu, přitom se předpokládá, že vzájemné pořadí podstromů je vždy zleva doprava. Jednotlivé průchody jsou znázorněny na tříuzlovém binárním stromu na obr. 5.17.

preorder = A, B, C
inorder = B, A, C
postorder = B, C, A



Obr. 5.17. Průchody stromem

5.3. Strojové prostředky implementace abstraktních datových struktur

Na abstraktní rovině řešíme daný problém abstraktním programem vytvářeným z abstraktních řídicích a datových struktur. Abychom mohli takový program řešit na konkrétním počítači, musíme se zabývat realizací (implementací) řídicích i datových struktur reálnými prostředky konkrétního počítače. Většina vyšších programovacích jazyků řeší problém implementace abstraktních řídicích struktur i některých vybraných datových abstrakcí nabídkou různě širokého sortimentu standardních prostředků jazyka.

Z praktických důvodů se budeme zajímat o prostředky jazyků pro implementaci datových struktur ve třech rovinách :

- strojově orientované jazyky
- tradiční vyšší programovací jazyk (Fortran, Algol 60)
- moderní vyšší programovací jazyk (Pascal)

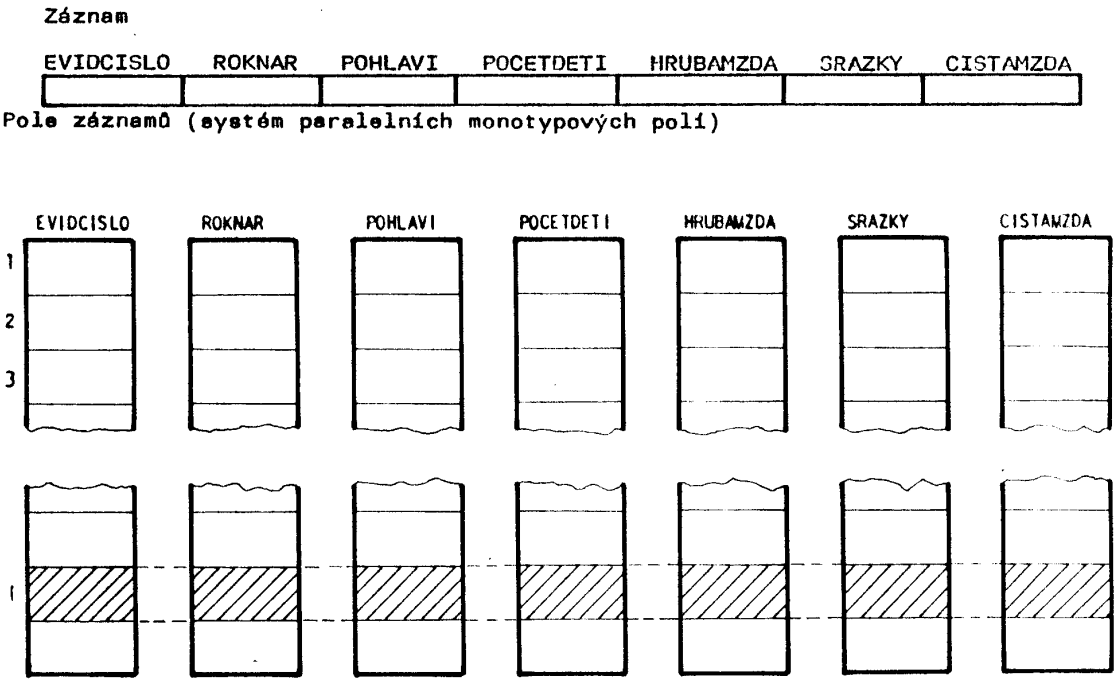
5.3.1. Strojově orientované jazyky

Na úrovni JSI je základním prostředkem pro práci s údaji adresa paměťového místa. Paměť jako celek, i každý její vnitřní souvislý celek, se jeví jako pole, jehož indexy jsou adresy po sobě jdoucích prvků. Záznam (record) lze na úrovni JSI vytvářet z N-tice po sobě jdoucích paměťových míst, kde N je součet počtů paměťových míst potřebných pro jednotlivé složky záznamu. Některé JSI mají standardní prostředky pro symbolickou identifikaci složek záznamu (viz pseudosekce v JSI JSEP [9]). Pole o k záznamech se pak implementuje polem k x N paměťových míst, ve kterém je uloženo k N-tic a každá N-tice představuje jeden záznam. Zvýšení "indexu" prvku o jednu představuje zvýšení adresového ukazatele o N (bytů či slov ap.).

Zřetězený seznam lze vytvářet ze záznamů, jejichž jednou ze složek je adresa další položky (záznamu) seznamu. Přístup k další položce seznamu se řeší mechanismem nepřímé adresace (JSI ADT), nebo mechanismem instrukcí využívajících pro adresaci jednoho nebo více registrů (JSI JSEP).

5.3.2. Tradiční vyšší programovací jazyk

Jediným prostředkem pro vytváření abstraktních datových struktur je v jazycích jako je Fortran resp. Algol 60 je pole. Pole záznamů, kde záznam sestává ze složek různých typů (integer, real, Boolean, complex aj.), lze vytvářet pomocí systému paralelních monotypových polí, kde i-tý prvek (záznam) pole sestává z řezu systémem paralelních polí na indexu i (z i-tých prvků všech polí systému). Situaci znázorňuje obr. 5.18. Jistá složka (např. ROKNAR) je pro všechny záznamy uložena v jednom poli, jehož identifikátor je vlastně identifikátorem složky záznamu (i-tý záznam je na obr. 5.18. tvořen vyšrafovanými políčky)



Obr. 5.18. Pole záznamů

Zřetězený seznam lze v těchto jazycích vytvářet opět pouze pomocí polí. Všechny položky seznamu jsou uloženy opět v systému paralelních monotypových polí, je však k němu připojeno jedno pole typu integer pro každý ukazatel v záznamu. Ukazatelem je index prvku, na který ukazatel odkazuje. Funkci hodnoty nil může zastávat libovolná hodnota vně intervalu vyhrazeného pro index (např. Ø nebo záporná hodnota). Průchod takto zkonstruovaným seznamem má pak tvar :

```
  :
DALSI : array [1..N] of Ø..N;  {Pole ukazatelů}
INDEX,PRVNI : Ø..N;
  :
    INDEX := PRVNI;
    while INDEX≠Ø do INDEX := DALSI [INDEX] ;
  :
```

5.3.3. Moderní vyšší programovací jazyk

Moderní vyšší programovací jazyk obsahuje řadu prostředků pro výstavbu a použití vyšších statických i dynamických datových struktur.

Programovací jazyk Pascal definuje několik standardních statických abstraktních datových struktur (pole, záznam, soubor, množina) a pro výstavbu dynamických struktur disponuje mechanismem dynamického přidělování paměti ovládaným standardními procedurami new a dispose.

5.3.4. Principy dynamického přidělování paměti

V předcházejících odstavcích jsme se seznámili s několika abstraktními datovými strukturami, které označujeme jako dynamické, čímž jsme rozuměli především skutečnost, že počet jejich komponent se může v průběhu zpracování měnit. Většinu těchto struktur lze vytvářet (implementovat) v jazyce Pascal pomocí tzv. ukazatelů, které vytvářejí vazby mezi jednotlivými komponentami (položkami, uzly) struktur. Komponenty se v průběhu zpracování generují standardní procedurou "new". Vyvstává otázka, jaký mechanismus se skrývá za přidělováním paměti nově generovaným objektům pomocí procedury new a jak lze řešit generování dynamických struktur v jazycích, které nemají proceduru new, nebo podobný prostředek, jako vestavěnou vlastnost jazyka ?

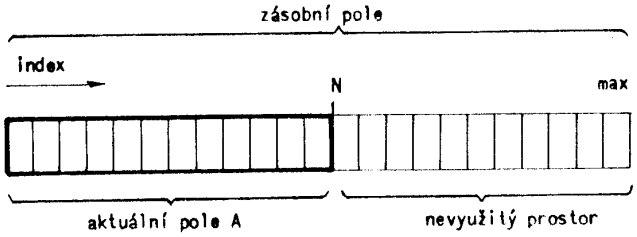
5.3.4.1. Příklad pseudodynamického pole

V řadě programovacích jazyků se zdá být citelným nedostatkem, že pole nelze deklarovat tak, aby se hranice indexu (resp. hranice typu interval) mohly určit až v průběhu zpracování programu. Princip řešení tohoto problému je charakteristický i pro jiné datové struktury a pro složitější mechanismy dynamického přidělování paměti.

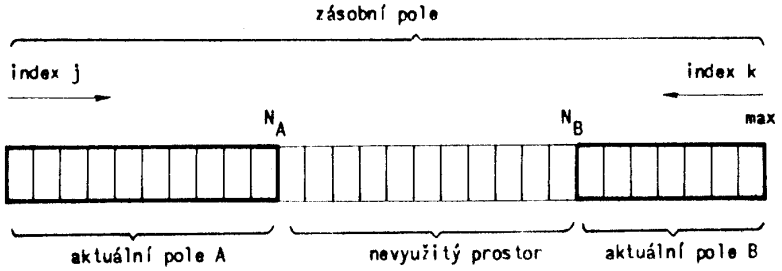
Je-li potřebné pole, které v různých případech řešení (a nebo v různých stadiích téhož řešení) má různý počet prvků, vyhradí se tak dostatečně velké pole, aby jeho velikost vyhověla i prostorově nejnáročnějšímu případu. Má-li aktuální pole

méně prvků, než je vyhrazený "zásobní" prostor, zůstává (horní) část pole nevyužita. Velikost aktuálního pole je dána aktuálním počtem prvků a takové pole se může chovat "dynamicky", pokud požadovaný počet prvků pole nepřekročí počet daný vyhrazeným zásobním prostorem. Požadavek na vyšší počet prvků vede jednoznačně k chybovému stavu.

Tento mechanismus lze vylepšit, je-li zapotřebí dvou "dynamických" polí téhož typu. Vyhradí-li se pro obě pole prostor jednoho zásobního pole, s počtem prvků daným součtem maximálních počtů prvků obou požadovaných polí, lze jedno pole umístit zleva v zásobním poli a druhé zprava v zásobním poli. Index j prvního požadovaného pole se bude shodovat s indexem i zásobního pole (i=j pro j=1,2,..., max). Pro index k druhého požadovaného pole (uloženého v zásobním poli zprava) a pro index i zásobního pole bude platit i=max+1-K (pro K=1,2,...,max; kde max je velikost zásobního pole max = max_1 + max_2, a max_1, max_2 jsou maximální odhady velikostí obou polí). Při práci s takto vytvořenými "dynamickými" poli se zvýší statistická naděje, že aktuální požadavky obou polí nepřekročí současně možnosti vyhrazeného prostoru. Tím se sníží pravděpodobnost chybového stavu způsobeného nedostatečnou kapacitou zásobního pole. Oba případy ilustrují obr. 5.19. a 5.20.



Obr. 5.19. Znázornění jednoho pseudodynamického pole



Obr. 5.20. Znázornění dvojice pseudodynamických polí

Z tohoto jednoduchého příkladu lze vyvodit zásady, které platí i pro složitější mechanismy dynamického přidělování paměti :

1. Paměťový zásobní prostor, ze kterého se čerpá při generování dynamických struktur je pro dané řešení pevně stanoven a musí být dostatečně velký tak, aby pokryl potřeby konkrétní aplikace.
2. Vyčerpání zásobního paměťového prostoru při konkrétním řešení vede k chybovému stavu, kterému lze při novém řešení zabránit dostatečným zvětšením tohoto

prostoru.

3. Slouží-li zásobní paměťový prostor více dynamickým strukturám současně a jeho velikost je stanovena z maximálních prostorových požadavků jednotlivých struktur, sníží se pravděpodobnost vzniku chybového stavu v důsledku vyčerpání zásobní paměti.
4. Dynamické přidělování paměti má vždy za důsledek snížení využití paměťového prostoru.

Z těchto zásad vyplývá, že dynamičnost reálných datových struktur má své meze, jejichž příčinou je konečnost paměti počítače. Tyto meze mohou být stanoveny programem, překladačem nebo operačním systémem a mají vždy hranici, která závisí na kapacitě využitelné paměti konkrétního počítače.

5.3.4.2. Rozdělení metod dynamického přidělování paměti

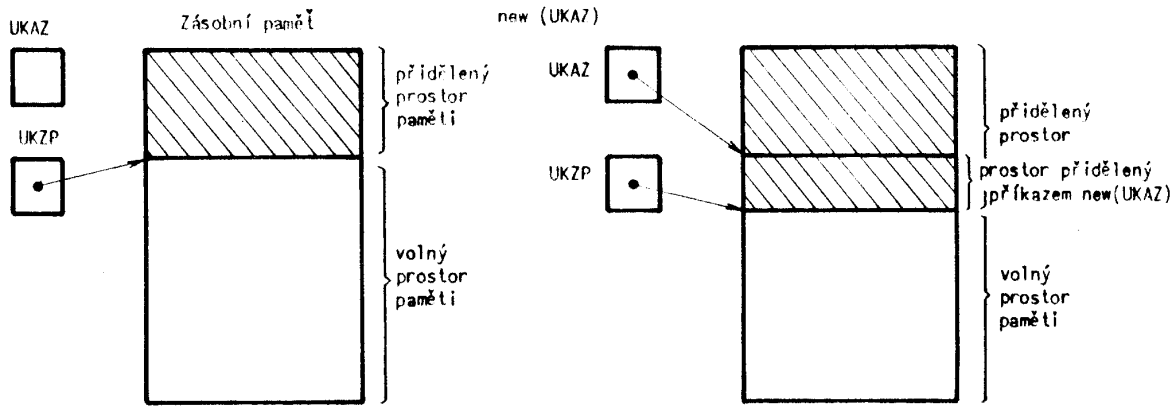
Pro dynamické struktury je charakteristické nejen to, že vzniká požadavek na vznik nových komponent těchto struktur, ale také to, že již nepotřebné (neaktuální) komponenty "zanikají" a nebudou se dále používat (viz mechanismus příkazu "dispose" v Pascalu). Metody dynamického přidělování paměti pak můžeme rozdělit podle toho, zda prostor zaujímaný již neaktuálními objekty může či nemůže být "vrácen" do zásobní paměti k novému použití (k regeneraci zásobní paměti). V případě vrácení neaktuálních objektů můžeme metody dále rozdělit podle toho, zda se prvky vrací do zásobní paměti explicitně (zvláštním příkazem či procedurou jazyka, který vrátí označený objekt do zásobní paměti) nebo implicitně (automatickým vyhledáváním neaktuálních objektů a vrácením jejich prostoru do zásobní paměti, které se uplatní až v okamžiku potřeby). Podle toho rozlišíme tři základní přístupy :

1. Dynamické přidělování paměti bez regenerace zásobní paměti
2. Dynamické přidělování paměti s regenerací zásobní paměti programově ovládaným vracením neaktuálních prvků.
3. Dynamické přidělování paměti s regenerací zásobní paměti s automatickým vracením neaktuálních prvků.

5.3.4.3. Dynamické přidělování paměti bez regenerace

Tento mechanismus je velmi jednoduchý a do vyčerpání zásobní paměti vysoce účinný. Zásobní paměť pracuje s ukazatelem, který je nastaven na první adresu nepoužité části zásobní paměti (volné paměti). Při požadavku na přidělení paměti generovanému dynamickému objektu, se adresa ukazatele předá jako výstupní hodnota procedury new (nebo jiné jí odpovídající procedury) a ukazatel se zvýší o hodnotu velikosti přiděleného prostoru. Prostor se přiděluje tak dlouho, dokud se nevyčerpá vyhrazená zásobní paměť. Situaci před a po provedení příkazu typu "new" zobrazuje obr. 5.21.

Jestliže přestane být dynamický objekt aktuální, může v Pascalu "zaniknout" pomocí procedury "dispose". Používá-li se tato metoda přidělování paměti, pak důsledkem této procedury není návrat paměťového prostoru do volné paměti, ale "označení" prostoru, jako prostoru již zaniklého (neaktuálního) objektu. Toto označení lze při snaze odkazovat se na zaniklý objekt využít k ustavení chybového stavu.



Obr. 5.21. Princip dynamického přidělování paměti bez regenerace

5.3.4.4. Dynamické přidělování paměti s programově řízenou regenerací

Podle způsobu, jakým se paměť přiděluje a jakým se vrací neaktuální paměťový prostor do zásobní paměti, můžeme tento způsob přidělování rozčlenit na tři charakteristické podtypy.

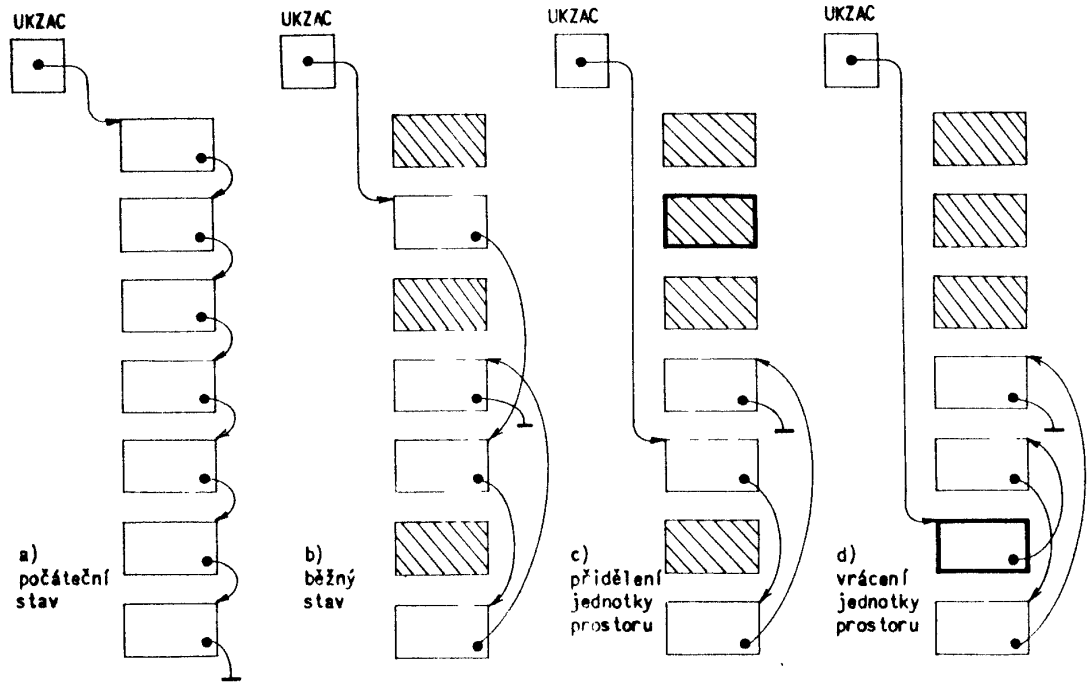
1. Metoda, která rozšiřuje princip přidělování paměti bez regenerace o možnost regenerace zásobní paměti o souvislý úsek paměti přidělený po sobě jdoucími příkazy new, využívá v některých verzích překladače Pascal (Pascal 8000, Pascal EC, FEL Pascal ADT) standardních procedur "mark" a "release".

   Procedura mark (UK) způsobí, že ukazatel UK je nastaven na aktuální konec přidělené paměti. Později užitá procedura release (UK) nastaví ukazatel zásobní paměti zpět na hodnotu pomocného ukazatele UK, čímž způsobí "vrácení" souvislého prostoru mezi adresou UK a adresou ukazatele zásobní paměti v okamžiku procedury "release".

2. Jiný přístup je charakteristický pro metodu, která označuje každou jednotku přidělovaného prostoru dvouhodnotovým příznakem volný/obsazený. Zpočátku je celý prostor "volný". Při přidělení prostoru nastaví metoda jeho příznak na hodnotu "obsazený" a při programovém vracení prostoru k dalšímu použití nastavuje příznak na hodnotu "volný". Při přidělování prostoru novému požadavku se sekvenčně prochází zásobní paměť od začátku, a vyhledává se první volný prostor, který svou velikostí může pokrýt požadovaný prostor. Nevýhodou této metody je poměrně zdlouhavé vyhledávání volného prostoru jako důsledek požadavku typu "new" a následná možnost "rozdrobení" volné paměti na úseky, které svou velikostí nemusí vyhovět požadavku generování objektu potřebujícího větší paměťový prostor (např. pole), i když v úhrnu je dostatečné množství volného prostoru.

3. Další metoda je vysoce účinná, jestliže všechny požadavky nárokují stejně velký paměťový prostor. Celou zásobní paměť lze zřetězit do seznamu typu zásobník, jehož prvky mají stejnou velikost. Vznikne-li nový požadavek na přidělení paměti, přidělí se paměťový prostor prvku na vrcholu zásobníku a prvek se odstraní ze zásobníku operací typu "POP". Při vracení neaktuálního prostoru se prvek obsahující tento prostor vloží zpět do zásobníku operací typu "PUSH". Tato metoda je časově velmi účinná, vyžaduje však dodatečný pracovní paměťový prostor pro zřetězení prvků do zřetězeného seznamu. Pro svou jednoduchost je zvláště výhodná v konkrétních aplikacích řešených v jazycích nevybavených vestavěnou

možností dynamického přidělování paměti.



Obr. 5.22. Princip zřetězeného zásobníku zásobní paměti

Na obr. 5.22. jsou čtyři stavy zřetězeného zásobníku zásobní paměti. Vyčárkovaný obdélník představuje aktuální (přidělenou) jednotku paměti, prázdný obdélník představuje volnou jednotku paměti. Stav ad a. představuje počáteční stav zásobníku, stav ad b. představuje stav dosažený v průběhu řešení. Dojde-li v tomto stavu k novému požadavku, přidělí se prostor tučně orámovaného obdélníku a vznikne stav ad c. Dojde-li v tomto stavu k deaktivaci tučně orámovaného obdélníku, začlení se na vrchol zásobníku a vznikne stav ad d.

5.3.4.5. Dynamické přidělování paměti s automatickou regenerací

Zatímco v předcházejících metodách byl za vracení neaktuálních položek dynamických datových struktur zodpovědný programátor, metody z této skupiny musí umět automaticky rozlišit, které položky zásobní paměti v daném stavu výpočtu jsou a které nejsou aktuální. Algoritmus této metody prochází všechny dynamické proměnné dostupné pomocí ukazatelů v daném stavu programu a označí je jako aktuální. Všechny neoznačené položky zásobní paměti jsou tedy využitelné k dalšímu použití. V další fázi této metody se roztroušené paměťové prostory označené jako neaktuální kondenzují do souvislého volného paměťového prostoru, s čímž souvisí také přesun aktuálních položek na jiné místo zásobní paměti a tudíž i změna hodnot všech ukazatelů, které na přesunuté aktuální položky ukazují. Značkovací fázi metody, která připomíná "sbírání smetí" v paměti se proto také říká anglickým názvem "garbage collector" (sběrač smetí).

Principiálně pracuje tato metoda tak, že postupně přiděluje volnou paměť

ze zásobní paměti až do okamžiku jejího vyčerpání. Vznikne-li další požadavek na přidělení paměti, vyvolá se automaticky složitá i časově náročná procedura regenerace paměti sestávající ze značkovací a kondenzační fáze.

Při ladění programu, který využívá této metody, je třeba dávat bedlivý pozor zejména v případech, kdy program může vyžadovat větší volnou paměť, než jaká je vůbec k dispozici. To se může při ladění projevit opakovaným a stejně marným, ale časově nákladným vyvoláváním procedury regenerace. Metoda vyžaduje důsledné udržování aktuálních hodnot všech ukazatelů (ačkoliv je často pohodlnější nedefinovat ukazatelové složky komponent struktur, jichž program vůbec nevyužívá). Tato metoda také není vhodná pro prostředí pracující v reálném čase, protože i když procedura regenerace nemusí být volána často, spotřebuje značné množství času a nelze vždy stanovit okamžik, kdy k jejímu vyvolání dojde. Metoda "garbage collector" patří k nejdůmyslnějším metodám dynamického přidělování paměti a je nezbytná ve velkých programových systémech s rozsáhlými dynamickými strukturami.

5.4. IMPLEMENTACE TYPICKÝCH ATD

5.4.1. Způsob implementace

V předchozích odstavcích jsme se seznámili s některými typickými ATD, přičemž popis obsahoval také jejich specifikace dohodnutým formalismem. Vzhledem k tomu, že byly popsány prostředky implementace ATD, můžeme nyní přistoupit k popisu implementace vybraných ATD v Pascalu s využitím popsaných implementačních prostředků.

Množina ATD implementovaných v této podkapitole je totožná s množinou ATD definovanou v kap. 5.2. U těch ATD, u nichž je způsob implementace již ustálen , případně existuje více známých metod (např. u pole) budeme uvádět i několik variant implementace téhož ATD.

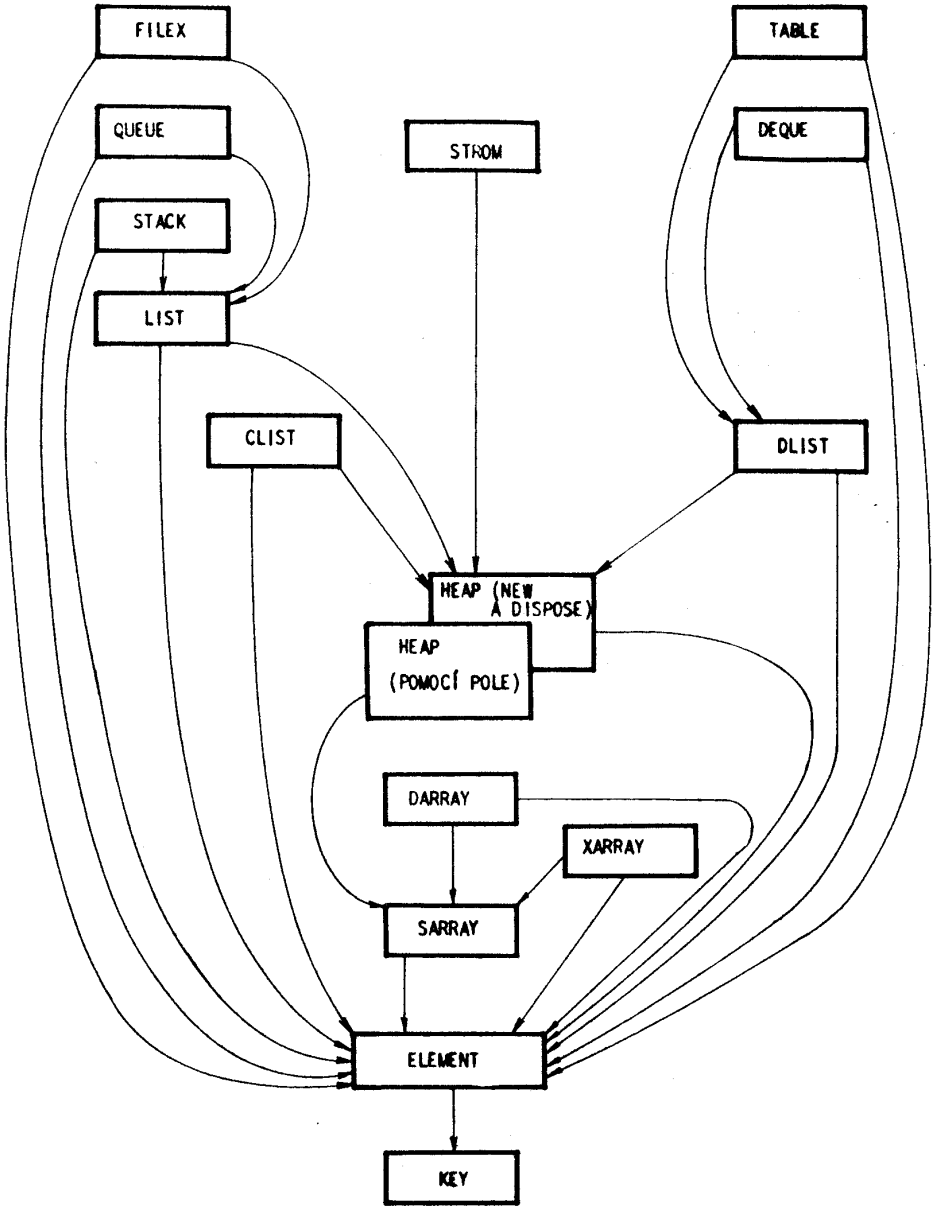
Implementaci popsané množiny ATD budeme specifikovat jako problém, jehož řešení popíšeme nejprve ve velkém. Tím určíme možnost vzájemného dovozu a vývozu datových typů a operací jednotlivých ATD. ATD uvedené v popisu ve velkém budeme pak zapisovat jako generické moduly.

Pro popis ve velkém užijeme orientovaného grafu. Uzly grafu reprezentují jednotlivé ATD, které chceme implementovat. Orientované hrany vyjadřují možnost dovozu a vývozu datových typů a operací z/do modulu popisujícího implementaci. Orientovaná hrana vyjadřuje skutečnost, že vyvážené datové typy a operace ATD (uzlu), do něhož hrana vchází jsou dováženy do ATD (uzlu), z něhož hrana vychází. Pro ATD z kap. 5.2. budeme užívat tyto zkratky :

lineární seznam	LIST
obousměrný seznam	DLIST
soubor	FILEX
kruhový seznam	CLIST
zásobník	STACK
fronta	QUEUE
oboustranně ukončená fronta	DEQUE
statické pole	SARRAY
dynamické pole	DARRAY

vícerozměrné pole	XARRAY
tabulka	TABLE
binární strom	STROM
klíč	KEY
dynamické přidělování paměti	HEAP
element struktury	ELEMENT

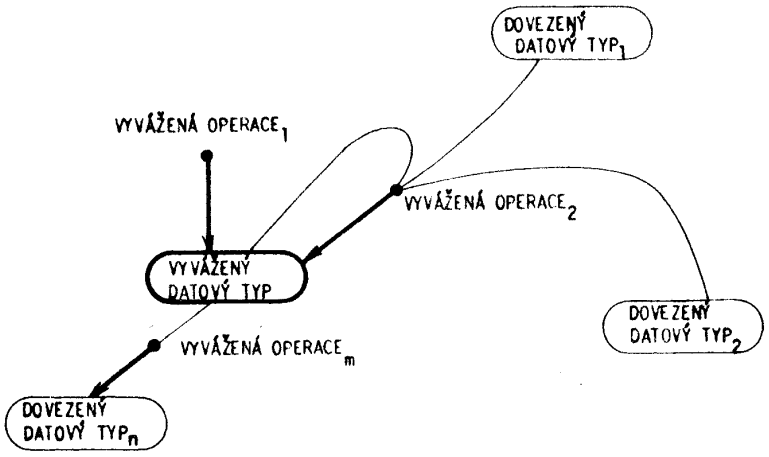
Potom graf popisu ve velkém lze vyjádřit obr. 5.23.



Obr. 5.23. Popis problému ve velkém

Z vlastností popisu ve velkém plyne, že musí jít o acyklický graf. Je zde tedy možné zvolit postupnou implementaci modulů (ATD) směrem shora-dolů nebo zdola-nahoru, případně jinou. V rámci této kapitoly zvolíme popis zdola-nahoru což vede k situaci, že dovezené objekty jsou vždy předem implementovány.

Při implementaci jednotlivých uzlů grafu (ATD) vyjdeme z jejich specifikace. Syntaktická specifikace ATD ve formě definice zobrazení, případně diagramu signatury (viz obr. 5.24.) určuje beze zbytku vývoz modulu a určuje některé objekty dovozu.



Obr. 5.24. Diagram signatury

Vyvážený datový typ je určen silným oválem v diagramu signatury a silnými tečkami vyvážené veřejné operace modulu. Slabými ovály jsou označeny dovážené datové typy. Jsou to typy těch parametrů vyvážených operací, které nejsou definovaného vyváženého typu. Minimálně tyto typy musí být modulem dováženy. Mimo to může modul dovážet další objekty využité při implementaci vyváženého typu a operací nad ním. Algoritmy vyvážených operací musí vyhovovat sémantické části definice ATD.

Implementaci jednotlivých ATD budeme zapisovat ve tvaru vhodném pro zpracování překladačem PASCALu-EC :

(1)

```
program  < jméno definovaného ATD >;
```

---

(2)

```
{ import }
¤include < jméno ATD, z něhož dovážíme jím exportované objekty >;
¤include                                    ⋮
    ⋮
¤include
```

---

(3)

```
{ export }
  const
      <definice vyvážených konstant>
```

```
(deklarace hlaviček vyvážených procedur a funkcí s direktivou oper)
 procedure (jméno definovaného ATD); module;
```

④ soukromé objekty definovaného ATD

```
⑤ {implementace}
implementace bloků vyvážených procedur a funkcí
```

```
begin end.   {prázdné tělo programu}
```

Pokud některá část není pro implementaci ATD potřebná, je možno ji vynechat (mimo ① a ⑤).

Vyvážený typ dat jsme zapisovali ve tvaru záznamu obsahujícího dvě části

```
TYP = record
        :
        veřejná část typu
        :
        PRIVATE :   :
                    soukromá část typu
                    :
      end;
```

Vyvážené typy dat budou obvykle plně abstraktní, tj. veřejná část typu bude prázdná. V tomto případě budeme definovat typ pouze jeho soukromou částí

```
TYP = :
      soukromá část typu
      :
```

Potom například namísto definice

```
TYP = record
        PRIVATE: record
            X,Y:integer;
            end
        end;
```

budeme zapisovat přímo

```
TYP = record
      X,Y:integer;
      end;
```

PŘÍKLAD :

Implementaci ATD POSINT celých kladných čísel z kap. 5.2. zapíšeme při dodržení shora popsaných konvencí takto :

```
program POSINT;
 {import}
    {dováží se objekty ATD integer a Boolean poskytované přímo jazykem PASCAL,
     proto uvedeme popis dovážených objektů pouze formou poznámky
     #include Boolean;
          type
               Boolean;
     #include integer;
          const
               maxint; mimo to je dovážena konstanta 1
          type
               integer;
          dovážené operace jsou + (sčítání) a relace = (rovnost)}
 {export}
          const
               JEDNA=1;
          type
               POSINT=JEDNA..maxint;
          function ADD(X,Y:POSINT):POSINT; oper;
          function SUCC(X:POSINT):POSINT; oper;
          function JEJEDNA(X:POSINT):Boolean; oper;
          procedure POSINT; module;
  { ④ vynecháno }
        {implementace}
      function ADD;
      begin
             ADD:=X+Y;
             end; {ADD}
      function SUCC;
      begin
             SUCC:=ADD(JEDNA,X);
             end; {SUCC}
      function JEJEDNA;
      begin
              JEJEDNA:=X=JEDNA;
             end; {JEJEDNA}
      begin
             end.
```

V dalších příkladech implementace nebudeme již dovoz objektů ATD poskytovaných přímo PASCALem (obdobně jako v tomto příkladu objekty ATD Boolean a integer) explicitně uvádět.

Dříve, než přistoupíme k implementaci shora specifikovaných ATD definujme modul ELEMENT, vyvážející datový typ ELEMENT, využívaný v mnoha specifikacích ATD jako typ obsahu položky ve struktuře. Implementace tohoto typu není pro nás podstatná, proto u tohoto typu budeme specifikovat pouze vyvážený datový typ bez vnitřní struktury a bez operací nad ním.

```
program ELEMENT;
 {import}
     :
 {export}
      type
         ELEMENT= ..... ;
      :
begin end.  {ELEMENT}
```

5.4.2. Pole

Nejjednodušší variantou pole pro implementaci bude statické jednorozměrné pole. Tento datový typ je poskytován přímo jazykem Pascal, proto je implementace triviální. Využívá se pouze pascalovských operací indexace pole.

```
program SARRAY;
 {implementace jednorozměrného statického pole}
 {import}
 {include ELEMENT; {typ složky pole}
 {include ORDTYP;  {typ indexu}
 {export}
  type
       SARRAY=array [ORDTYP] of ELEMENT;
  {množina operací odpovídá specifikaci ATD pole z kap. 5.2.4.}
procedure SWRITE (var X:SARRAY; Y:ORDTYP; Z:ELEMENT); oper;
procedure SREAD (var X:SARRAY; Y:ORDTYP; var Z:ELEMENT); oper;
procedure SARRAY; module;
 {implementace}
procedure SWRITE;
begin
     X[Y]:=Z;
     end; {SWRITE}
procedure SREAD;
begin
     Z:=X[Y];
     end;  {SREAD}
begin end.  {SARRAY}
```

Vzhledem k tomu, že datový typ SARRAY je standardním typem Pascalu, nebudeme jeho dovoz dále explicitně uvádět a budeme využívat přímo pascalovských zápisů definice typu a operací SWRITE a SREAD zápisem identifikátoru pole a indexu v hranatých závorkách.

ATD pole se většinou v paměti počítače reprezentují sekvencí po sobě jdoucích paměťových míst. Jestliže jeden prvek pole zaujímá ℓ po sobě jdoucích paměťových míst, pak i-tý prvek pole A: array [1..N] of ELEMENT, které je v paměti uloženo od adresy A, začíná na adrese $A+(i-1)*\ell$.

Statické pole můžeme využít pro implementaci dynamického pole. Pokud existuje statické pole A:array [1..N] of ELEMENT, přičemž N je natolik velké, že počet složek dynamického pole

B: array [low..high] of ELEMENT(tj. high-low+1) nepřesáhne při libovolné inicializaci hodnotu N, můžeme libovolnou složku pole B zobrazit do pole A podle vztahu

$$B[M] = A[M-low+1]$$

Tohoto mechanizmu využijeme pro následující implementaci dynamického jednorozměrného pole.

Nechť existuje ordinální typ ORDTYP dovážený při implementaci typu dynamické pole jako typ indexu.

Při implementaci budeme dynamické pole modelovat následujícím pascalovským typem

```
const
    N=...;  {maximální počet položek modelované paměti}
type
    DARRAY=record
        LOW,HIGH:ORDTYP; {minimální a maximální index}
        A:array [1..N] of ELEMENT; {statické pole pro dynamické
                                    přidělování prostoru}
        end;  {DARRAY}
```

Datový typ array [1..N] of ELEMENT reprezentuje souvislou oblast paměti s buňkami typu ELEMENT adresovatelnou hodnotami 1..N. Implementace ATD dynamické jednorozměrné pole má pak tvar

```
program DARRAY;
{implementace dynamického jednorozměrného pole}
{import}
{include ELEMENT;
{include ORDTYP;
{export}
    const
        N=10000;
    type
        DARRAY=record
            LOW,HIGH:ORDTYP;
            A: array [1..N] of ELEMENT;
            end;
procedure DARRAYINIT (var X:DARRAY; Y,Z:ORDTYP); oper;
function DLOW (var X: DARRAY): ORDTYP; oper;
function DHIGH (var X:DARRAY): ORDTYP; oper;
procedure DWRITE (var X:DARRAY; M:ORDTYP; Z:ELEMENT); oper;
procedure DREAD (var X:DARRAY; M:ORDTYP; var Z:ELEMENT); oper;
procedure DARRAY; module;
{implementace}
procedure DARRAYINIT;
begin
  with X do begin
    if   Y<= Z   then begin {LOW <= HIGH}
      LOW:=Y;
      HIGH:=Z;
      if (ord(HIGH)-ord(LOW)+1) > N then {příliš velké pole}
        HALT;
      end
```

```
        else
            HALT;
    end; { DARRAYINIT }
function DLOW;
begin
    DLOW:=X.LOW;
    end; { DLOW }
function DHIGH;
begin
    DHIGH:=X.HIGH;
    end; { DHIGH }
procedure DWRITE;
begin
    with X do
        if (M>=LOW) and (M<=HIGH) then
            A [ ord(M)-ord(LOW)+1 ]:=Z
        else { chyba }
            HALT;
    end; { DWRITE }
procedure DREAD;
begin
    with X do
        if (M>=LOW) and (M<=HIGH) then
            Z:=A[ord(M)-ord(LOW)+1]
        else
            HALT;
    end; { DREAD }
begin end. { DARRAY }
```

Nyní přistupme k implementaci vícerozměrného pole.

Statické vícerozměrné pole je poskytováno přímo PASCALem. Zaměřme se proto na obecný popis metod zpřístupnění prvku vícerozměrného pole a při implementaci na dynamické vícerozměrné pole.

Nechť je pole B dvojrozměrné a má např. K řádků a L sloupců. Pak lze toto pole zobrazit v sekvenci po sobě jdoucích paměťových míst jako vektor řádků (tzv. uložení po řádcích, ve kterém se "rychleji" mění druhý index) nebo jako vektor sloupců (tzv. uložení po sloupcích, ve kterém se "rychleji" mění první index). Pravidlo zobrazení lze rozšířit i na n-rozměrné pole. Je-li pro zobrazení do sekvence charakteristické, že nejrychleji se mění nejpravější (poslední) index, říkáme, že jde o "uložení pole po řádcích"; mění-li se nejrychleji nejlevější (první) index, říkáme, že jde o "uložení pole po sloupcích". Jazyky jako Algol 60, Algol 68, PL/I a Pascal používají uložení po řádcích, zatím co ve Fortranu je charakteristické uložení po sloupcích. Tuto skutečnost je třeba mít na zřeteli zejména při spolupráci modulů zapsaných v různých jazycích.

Podle způsobu, kterým se z ℓ indexů ℓ-rozměrného pole získá jeden index (adresa) jemu odpovídajícího prvku v jednorozměrném poli rozlišujeme několik zpřístupňovacích metod, které se navzájem liší rychlostí přístupu k prvku pole, dodatečnými nároky na paměťový prostor či mírou kontrol legálnosti zadaných indexů.

Definujme nyní záhlaví modulu reprezentujícího L-rozměrné pole. L-rozměrný index budeme modelovat jednorozměrným statickým polem

```
type INDEX = array [1..L] of ORDTYP;  {obecný L-rozměrný index}
```

Záhlaví modulu může mít tento tvar

```
program XARRAY;
 {implementace L-rozměrného dynamického pole}
 {import}
 {include ELEMENT;   {typ položky}
 {include ORDTYP;    {typ indexu}
 {include DARRAY;    {jednorozměrné dynamické pole}
 {export}
     const
         L=...; {počet rozměrů}
     type
         XARRAY=...; {vícerozměrné pole}
         INDEX=array [1..L] of ORDTYP; {ℓ-rozměrný index}
procedure XARRAYINIT(var X:XARRAY; Y,Z:INDEX); oper;
procedure XLOW (var X:XARRAY; var Y:INDEX); oper;
procedure XHIGH (var X:XARRAY; var Y:INDEX); oper;
procedure XWRITE (var X:XARRAY; M:INDEX; Z:ELEMENT); oper;
procedure XREAD (var X:XARRAY; M:INDEX; var Z:ELEMENT); oper;
procedure XARRAY; module;
```

Implementační část modulu (tj. popis typu XARRAY a bloků operací) bude uvedena u jednotlivých metod zpřístupnění prvku pole.

5.4.2.1. Zpřístupnění prvku vícerozměrného pole mapovací funkcí

Nechť je dáno ℓ-rozměrné pole B:array [$low_1..high_1, low_2..high_2, ..., low_\ell ... high_\ell$] of ELEMENT

Pak prvek tohoto pole, jehož zápis má tvar $B[j_1, j_2, ... j_\ell]$ bude zobrazen (mapován) do jednorozměrného pole A s indexem :

$$1 + \sum_{m=1} (j_m - low_m) * D_m \qquad (5.1.)$$

a tedy

$$B[j_1, j_2, ..., j_\ell] \rightarrow A[1 + \sum_{m=1} (j_m - low_m) * D_m]$$

kde :

$$D_{\ell+1} = 1$$

$$D_m = (high_m - low_m + 1) * D_m$$

Vztahu (5.1.) se říká mapovací funkce a hodnota této funkce se musí vyčíslovat v průběhu výpočtu při každé referenci na indexovanou proměnnou. Jak je z tvaru funkce vidět, může být vyčíslení zejména u vícedimenzionálních polí značně časově náročné.

Implementační část modulu vícerozměrného pole využívající mapovacích funkcí může mít tvar :

```
type
     XARRAY=record
         LOW,HIGH:array[1..L] of ORDTYP;  {dolní a horní hranice indexů}
         A: DARRAY; {jednorozměrné dynamické pole}
         end;
function D (var X:XARRAY; M:1..L): 1..N;
 {mapovací funkce}
begin
     if M=1 then
        D:=1
     else with X do
        D:=(ord(HIGH[M-1])-ord(LOW[M-1])+1) * D(X,M-1);
     end; {D}
function SINDEXEM (var X:XARRAY; J:INDEX):1..N;
var
     SUM:1..N;
     M:1..L;
begin
     with X do begin
          SUM:=1;
          for M:=1 to L do
              SUM:=SUM+ (ord(J[M])-ord(LOW[M]))*D(X,M);
          end;
end; {SINDEXEM}
 {implementace}
procedure XARRAYINIT;
var
     SUM:1..N;
begin
     with X do begin
         LOW:=Y;
         HIGH:=Z;
         SUM:=SINDEXEM(X,HIGH);  {počet položek jednorozměrného pole =
                                  maximální možný index}
         DARRAYINIT(A,1,SUM);
         end;
     end; {XARRAYINIT}
procedure XLOW;
begin
     Y:=X.LOW
     end; {XLOW}
procedure XHIGH;
begin
     Y:=X.HIGH;
     end;  {XHIGH}
procedure XWRITE;
begin
     DWRITE(X.A,SINDEXEM(X,M),Z);
     end;  {XWRITE}
```

```
procedure XREAD;
begin
    DREAD(X.A,SINDEXEM(X,M),Z);
    end; { XREAD }
begin end. { XARRAY }
```

5.4.2.2. Zpřístupnění prvku vícerozměrného pole s využitím informačního vektoru

Ze vztahu (5.1.) je zřejmé, že členy low_m a D_m jsou nezávislé na momentálních hodnotách indexů indexované proměnné a že jsou spočitatelné v okamžiku zpracování deklarace pole (definice typu pole). Vztah lze upravit na tvar

$$1 + \sum_{m=1}^{\ell} j_m D_m - \sum_{m=1}^{\ell} low_m . D_m \qquad (5.2.)$$

Pak je tedy při zpracování deklarace pole možné vytvořit pomocný informační vektor (angl. dope vector) v němž se uloží všechny hodnoty D_m a $\sum_{m=1}^{\ell} low_m . D_m$. Jejich výpočet se provede jednou a to v době inicializace. Vyčíslení mapovací funkce podle vztahu (5.2.) je s použitím informačního vektoru rychlejší, za cenu dodatečné paměti potřebné pro uložení informačního vektoru. Protože řada jazyků vyžaduje i různé kontroly správnosti zápisu indexované proměnné (legální počet indexů, legální hodnota daného indexu) může informační vektor obsahovat i jinou pomocnou informaci. Typický informační vektor obsahuje podle [5] tyto položky :

1. Počet dimenzí pole (ℓ)
2. Dolní a horní hranice pro každou dimenzi
3. Celkový počet položek pole
4. D_m pro $m = 1,2, \ldots, \ell$
5. $\sum_{m=1}^{\ell} low_m . D_m$
6. Adresa prvního prvku pole

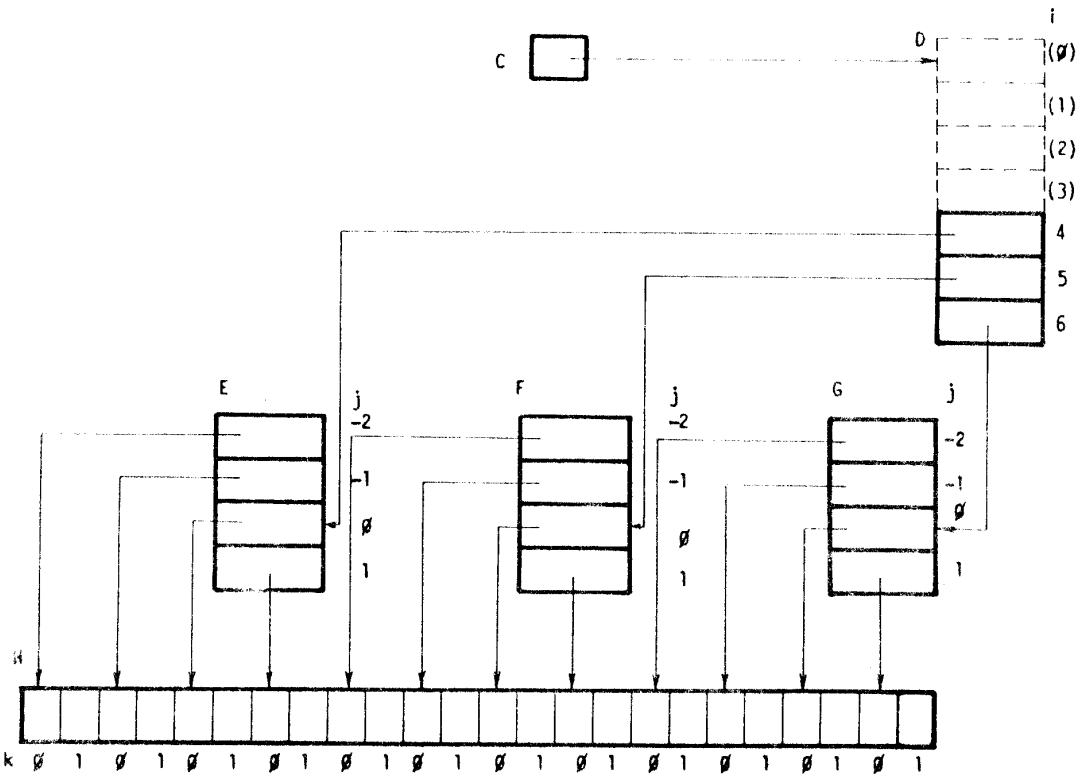
Předává-li se pak v podprogramech (procedurách) parametr typu pole, předává se často adresa informačního vektoru, který obsahuje všechny důležité informace o poli včetně jeho adresy. V rámci cvičení necháváme na čtenáři implementovat vícerozměrné dynamické pole s využitím popsaného informačního vektoru. Záhlaví modulu by mělo být shodné s předchozí implementací mapovacími funkcemi.

5.4.2.3. Zpřístupnění prvku vícerozměrného pole s využitím Iliffových vektorů

Zpřístupnění prvku pole lze za cenu dalších požadavků na paměťový prostor zrychlit použitím tzv. Iliffových vektorů. Jejich princip vysvětlíme na příkladě [5] pole definovaného zápisem :

```
B:array [4..6, -2..1, 0..1] of ELEMENT
```

Struktura Iliffových vektorů i s polem H v němž je zobrazeno pole B je na obr. 5.25. Použijeme-li zápis ⟨ M ⟩ ve smyslu "obsah adresy M", pak K obsahu prvku B [i,j,k] se dostaneme pomocí výrazu ⟨⟨⟨ C ⟩ +i ⟩ +j ⟩ +k.

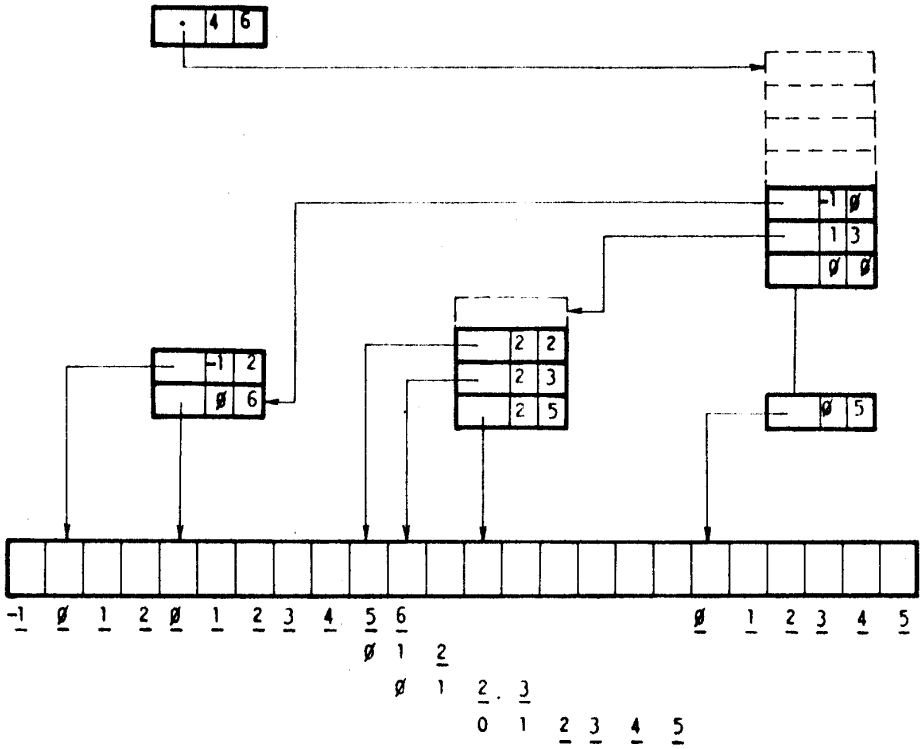


Obr. 5.25. Systém Iliffových vektorů

V obr. 5.25. představuje C zpřístupňovací adresu k celému systému Iliffových vektorů pro pole B. Vyčárkované pole vektoru D odpovídají neexistujícím hodnotám indexu i mezi Ø (která odpovídá prvnímu prvku vektoru D) a dolní hranicí první dimenze. Ukazatel vždy ukazuje na "nulový" index dalšího vektoru. K dalšímu ukazateli se dostáváme přičtením dalšího indexu indexované proměnné k adrese ukazující na "nultý" index Iliffova vektoru. Z celého principu je zřejmé, že přístup k prvku pole pomocí Iliffových vektorů se realizuje pouze pomocí aditivních operací a nevyžaduje žádného násobení, což je základem rychlosti přístupové metody. Na druhé straně je však kromě 24 prvků vlastního pole H zapotřebí 16 paměťových míst o velikosti potřebné pro uchování adresy paměti. Metoda Iliffových vektorů je nejúčinnější tehdy, když rozsahy jednotlivých dimenzí vzrůstají od první dimenze k poslední a nejméně účinnou je v případě, kdy první dimenze má největší rozsah, a dále rozsahy klesají.

Prvky Iliffových vektorů lze doplnit o dvojici hodnot vymezujících povolený rozsah přičítaného indexu a tím umožnit kontrolu přípustnosti hodnoty daného indexu. V obr. 5.25. by pak Iliffovy vektory E,F i G byly v případě pravoúhlého pole doplněny všechny o stejnou čtveřici dvojic hodnot, což se zdá být neekonomické. Této myšlenky lze však využít pro zpřístupnění jen určitých prvků pole. Na obr. 5.26. je uveden příklad (viz [5]) systému Iliffových vektorů zpřístupňujících pouze tyto prvky trojrozměrného pole :

B[4,-1,-1], B[4,-1,Ø], B[4,-1,1], B[4,-1,2],
B[4,Ø,Ø], B[4,Ø,1], B[4,Ø,2], B[4,Ø,3], B[4,Ø,4], B[4,Ø,5], B[4,Ø,6]
B[5,1,2]
B[5,2,2], B[5,2,3]
B[5,3,2], B[5,3,3], B[5,3,4], B[5,3,5]
B[6,Ø,Ø], B[6,Ø,1], B[6,Ø,2], B[6,Ø,3], B[6,Ø,4], B[6,Ø,5]



Obr. 5.26. Systém Iliffových vektorů pro zapamatování a zpřístupnění pouze jistých prvků pole

5.4.2.4. Trojúhelníková matice

V řadě aplikací se pracuje s maticí NxN, jejíž všechny prvky nad (pod) diagonálou jsou nulové, nebo zcela nedefinované (a nepoužívané). Takovou matici lze uložit tak, že v paměti uchováváme pouze definované (používané) prvky. Matice A s prvky

$$A_{11}$$
$$A_{21}\ A_{22}$$
$$A_{31}\ A_{32}\ A_{33}$$
$$\vdots$$
$$A_{N1}\ A_{N2}\ A_{N3}\ \cdots\cdots\ A_{NN}$$

Vyžaduje pro uložení svých prvků $\frac{N}{2}(N+1)$ paměťových jednotek. Zavedením pomocných mapovacích funkcí f_1 a f_2 získáme zobrazení

$$A[j,k] \rightarrow B[f_1(j) + f_2(k)]$$

kde pro trojúhelníkovou matici $f_1(j) = \frac{j(j-1)}{2}$ a $f_2(k) = k$

Pak tedy platí zobrazení

$$A[j,k] \rightarrow B[\frac{j(j-1)}{2} + k]$$

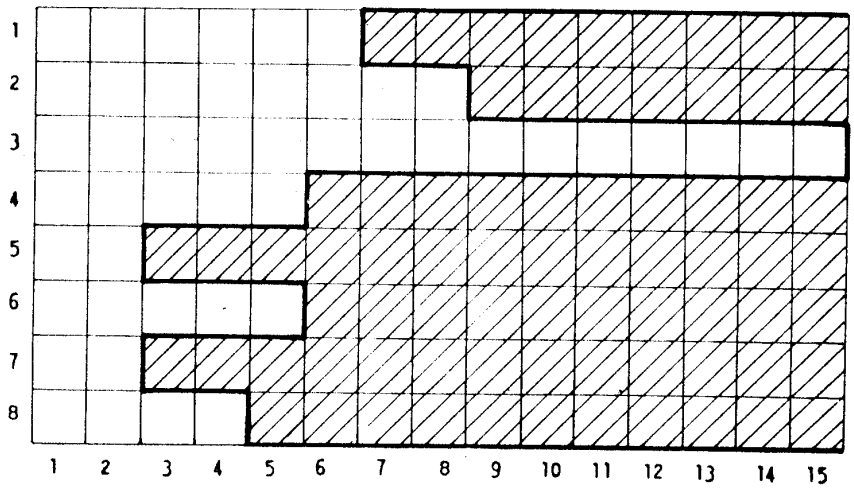
Je-li nutné zrychlit přístup k prvkům trojúhelníkové matice, je možné použít tabelárně zapamatovaných funkcí f_1 a f_2.

Mohou se vyskytnout případy, kdy v dané aplikaci se pracuje se dvěmi stejnými trojúhelníkovými maticemi (nebo se sudým počtem takových matic). Tento případ lze snadno vyřešit zavedením obdélníkové matice C:array [1..N, 1..(N+1)] a použitím zobrazení

A [j,k] → C [j,k]
B [j,k] → C [k,j+1]

5.4.2.5. Matice s nestejně dlouhými řádky

V některých aplikacích se setkáme s maticí, na konci jejichž řádků jsou nedefinované (nepoužívané) prvky. Taková matice je znázorněna na obr. 5.27. a anglicky se jí říká "rag table" nebo "jagged table".



Obr. 5.27. Matice s nestejně dlouhými řádky

Matice na obr. 5.27. má 15 sloupců a 8 řádků a tedy 120 prvků. Pouze 47 z nich je aktivních. Je-li účelné ukládat v paměti pouze definované (používané) prvky matice, lze aktivní prvky uložit do vektoru M o délce rovné počtu aktivních prvků, a jednotlivé prvky zpřístupňovat pomocí přístupového vektoru PV, který má tolik prvků, kolik má řádků matice s nestejně dlouhými řádky. Hodnotou i-tého prvku přístupového vektoru je pak součet počtu definovaných prvků řádků 1 až i-1. Pak lze zavést zobrazení

A[i,j] → M[PV[i]+ j]

Přístupový vektor k matici na obr. 5.27. by měl hodnoty

Ø	6	14	29	34	36	41	43
1	2	3	4	5	6	7	8

Pak prvek A[4,4] → M[PV[4]+ 4] = M[29+4] = M[33]

5.4.2.6. Řídké pole

V řadě zejména numerických aplikací se vyskytuje případ, kdy pole (nejčastěji matice) má počet nenulových prvků významně menší, než počet prvků nulových. Taková pole se nazývají řídká pole. Je-li úspora paměti vyžadovaná pro nulové prvky významnější než čas potřebný pro přístup k jednomu prvku pole, lze řídké pole implementovat na principu vyhledávací tabulky. Tabulka obsahuje pouze nenulové prvky a funkci vyhledávacího klíče prvku plní indexy prvku pole.

Pozn.: "Implicitní hodnotu" řídkého pole samozřejmě nemusí být vždy nula. Řídké pole můžeme vytvořit v každém případě, kdy jistá hodnota svým výskytem v poli významně dominuje.

Operace ATD pole se při této implementaci redukují na operace INIT, WRITE a READ (viz odst. 4.2.4.) a operace DEFINED není v tomto případě použitelná.

Pak lze operace nad řídkým polem definovat pomocí operací nad ATD tabulka takto :

```
INIT (ARRAY) = TABLEINIT (TABLE)

XWRITE (ARRAY, I, ELEMENT) =
        if ELEMENT ≠ Ø then INSERT (TABLE, I, ELEMENT)
                       else
           if SEARCH (TABLE, I) then DELETE (TABLE, I)
XREAD (ARRAY, I, ELEMENT) =
        if SEARCH (TABLE, I) then TREAD (TABLE, I, ELEMENT)
                             else ELEMENT := Ø
```

Což lze slovy komentovat takto :

Zapisuje-li se nenulový prvek, vloží se do tabulky operací INSERT.
Zapisuje-li se nulový prvek, je nutné zjistit, zda byl před zápisem nenulový (tzn. že je v tabulce nalezen) a pak se z tabulky vyřadí (DELETE).
Není-li při čtení prvek v tabulce nalezen, znamená to, že je nulový.

5.4.2.7. Poznámka k úsporným uložením některých polí

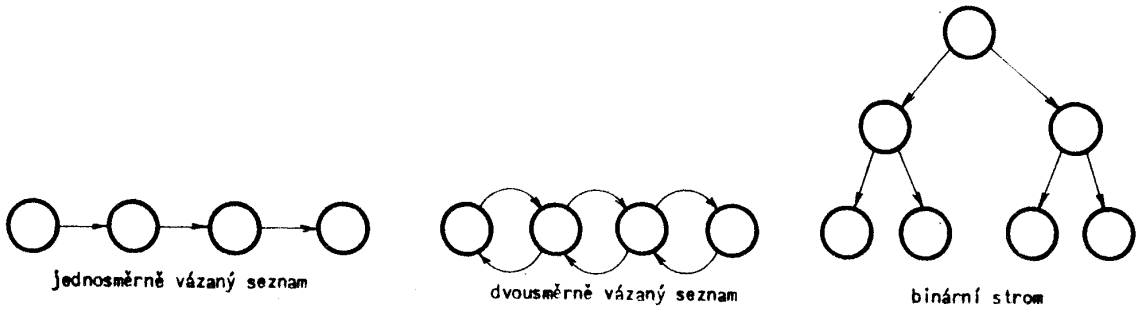
V případě trojúhelníkové matice, matice s nestejně dlouhými řádky a řídkého pole přistupujeme k paměťové úsporné implementaci vždy po důkladné analýze paměťových možností použitého počítače, skutečné úspory paměti a proudloužení přístupové doby k jednotlivým prvkům pole. Obvyklá tendence vývoje počítačů sleduje vždy postupné zvyšování kapacity paměti v každé nové generaci počítačů a proto se často nahlíží na paměťově úsporné implementace datových struktur řešené programově jako na nouzové řešení.

Vzhledem k prostorovým možnostem není možné uvádět implementace modulů reprezentujících všechny varianty implementace polí. Čtenář by si je však měl provést v rámci cvičení.

5.4.3. Dynamické přidělování paměti

Pokračujeme-li v řešení problému zapsaného ve velkém zdola-nahoru, musíme nyní přistoupit k implementaci modulu dynamického přidělování paměti (HEAP). Specifikace tohoto modulu nebyla uvedena v kap. 5.2., jelikož se nejedná o typický

ATD. Proveďme ji proto nyní s ohledem na potřebné vlastnosti vyvážených objektů pro implementaci ostatních ATD. Z popisu ve velkém plyne, že vyvážený datový typ z operace modulu HEAP bude využit při implementaci jednosměrně i dvousměrně vázaného seznamu a stromu (viz obr. 5.28.)



Obr. 5.28. Schematické znázornění seznamů a binárního stromu

Tyto dynamické struktury jsou budovány z položek (zobrazených na obr. 5.28. kružnicemi). Každá tato položka (ITEM) má obsah typu ELEMENT, přičemž mezi položkami existuje jedna, dvě případně u grafu obecně n-vazeb. Položky s vazbami je tedy nutné dynamicky vytvářet a rušit. Z hlediska dynamického přidělování položek navrhneme vyvážené objekty modulu HEAP takto :

- konstanty HEAPN ... maximální počet možných vazeb položky vyváženého typu HEAPITEM
  HEAPMAX ... maximální možný počet současně vytvořených položek
- typy HEAPPOINT ... zpřístupňující typ - typ vazby
  HEAPITEM .... typ položky s vnitřní složkou typu ELEMENT a HEAPN - vazbami
  HEAPSPACE ... prostor pro dynamické přidělování položek
- operace

  procedure HEAPINIT (var X:HEAPSPACE);
  operace inicializující dynamické přidělování položek typu HEAPITEM z prostoru HEAPSPACE
  function NEWX (var X:HEAPSPACE): HEAPPOINT;
  vytvoří dynamicky novou položku typu HEAPITEM a zpřístupní ji hodnotou typu HEAPPOINT
  procedure DISPOSEX (var X:HEAPSPACE; Y:HEAPPOINT);
  zruší dynamicky vytvořenou položku zpřístupněnou hodnotou Y typu HEAPPOINT
  procedure HEAPINS (var X:HEAPSPACE; Y:HEAPPOINT; Z:ELEMENT);
  uloží hodnotu proměnné Z do dynamicky vytvořené položky zpřístupněné hodnotou Y
  procedure HEAPREC (var X:HEAPSPACE; Y:HEAPPOINT; var Z:ELEMENT);
  získá hodnotu z dynamicky vytvořené položky zpřístupněné hodnotou Y a uloží ji do proměnné Z
  procedure LINKINS (var X:HEAPSPACE; Y:HEAPPOINT; NUM:1..HEAPN; Z:HEAPPOINT);
  uloží hodnotu Z jako NUM-tou vazbu položky zpřístupněné hodnotou Y.
  function LINKREC (var X:HEAPSPACE; Y:HEAPPOINT; NUM:1..HEAPN): HEAPPOINT;
  získá hodnotu NUM-té vazby položky zpřístupněné hodnotou Y

```
function HEAPNIL:HEAPPOINT;
vytvoří hodnotu "nil" zpřístupňujícího typu
```

Po specifikaci vyvážených objektů můžeme přistoupit k implementaci typů a bloků operací. Vytvoříme dvě varianty implementace. První z nich ukazuje možnost realizace modulu HEAP bez užití datového typu ukazatel a prostředků dynamického přidělování paměti new a dispose. Tuto variantu uvádíme, abychom ukázali možnost realizace všech popsaných ATD i v jazycích, které přímo prostředky dynamického přidělování paměti neposkytují. Druhá varianta je pak obvyklou implementací problému dynamického přidělování paměti v Pascalu.

V následujících implementacích je pak možné volbou varianty dováženého modulu dynamického přidělování paměti HEAP možné určit, zda užijeme vlastního prostoru pro dynamické přidělování, či (pokud je k dispozici) užijeme prostředků poskytovaných implementačním jazykem.

```
program HEAP;
 {dynamické přidělování položek typu HEAPITEM s HEAPN-možnými vazbami;
  varianta využívající paměťový prostor reprezentovaný statickým jednorozměrným
  polem}
 {import}
     type
        ELEMENT: {typ obsahu položky}
 {export}
     const
        HEAPN=1; {počet vazeb položky}
        HEAPMAX=2ØØ; {maximální počet vytvořených položek}
     type
        HEAPPOINT=Ø..HEAPMAX;  {zpřístupňující typ}
        HEAPITEM=record
              CONT:ELEMENT; {obsah}
              LINK:array[1..HEAPN] of HEAPPOINT; {vazby}
              end;
        HEAPSPACE=array [1..HEAPMAX] of {prostor pro přidělování}
                  record
                     ALLOC:Boolean;  {příznak přidělení}
                     ITEM:HEAPITEM; {obsah}
                     end;
     procedure HEAPINIT (var X:HEAPSPACE); oper;
      {inicializace prostředků dynamického přidělování položek}
     function NEWX (var X:HEAPSPACE):HEAPPOINT; oper;
       {přidělení položky}
     procedure DISPOSEX (var X:HEAPSPACE; Y:HEAPPOINT); oper;
       {zrušení položky}
     procedure HEAPINS (var X:HEAPSPACE; Y:HEAPPOINT; Z:ELEMENT); oper;
       {vložení hodnoty Z do dynamicky přidělené položky}
     procedure HEAPREC (var X:HEAPSPACE; Y:HEAPPOINT; var Z:ELEMENT); oper;
       {získání hodnoty dynamicky přidělené položky}
     procedure LINKINS (var X:HEAPSPACE; Y:HEAPPOINT;
                        NUM:1..HEAPN;  Z:HEAPPOINT); oper;
      {uložení NUM-té vazby Z do položky označené Y}
```

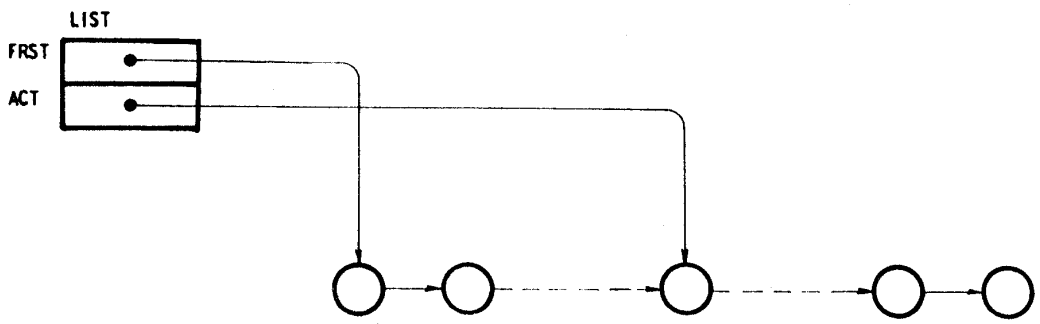
```
function LINKREC (var X:HEAPSPACE; Y:HEAPPOINT;
                  NUM:1..HEAPN):HEAPPOINT; oper;
    {získání NUM-té vazby z položky označené Y}
function HEAPNIL:HEAPPOINT; oper;
    {získání ukazatele "nil"}
procedure HEAP; module;
 {implementace}
procedure HEAPINIT;
 {inicializace přidělování paměti}
var I:HEAPPOINT;
begin
   for I:=1 to HEAPMAX do X[I].ALLOC:=false;
   end; {HEAPINIT}
function NEWX;
{přidělení nové proměnné}
var I:integer;
   POKRACUJ:Boolean;
begin
   I:=1; POKRACUJ:=true;
   while POKRACUJ do begin
        if X[I].ALLOC then
           I:=I+1
        else
           POKRACUJ:=false;
        if I > HEAPMAX then
           POKRACUJ:=false;
        end;
   if I <= HEAPMAX then begin
        NEWX:=I;
        X[I].ALLOC:=true;
        end
   else {chyba}
        NEWX:=Ø;
   end; {NEWX}
procedure DISPOSEX;
 {zrušení proměnné}
begin
   X[Y].ALLOC:=false;
   end; {DISPOSEX}
procedure HEAPINS;
 {vložení proměnné označené Y}
begin
   X[Y].ITEM.CONT:=Z;
   end; {HEAPINS}
procedure HEAPREC;
 {získání hodnoty proměnné označené Y}
begin
   Z:=X[Y].ITEM.CONT;
   end; {HEAPREC}
```

```
procedure LINKINS;
 {vložení NUM-té vazby}
begin
   X[Y].ITEM.LINK[NUM]:=Z;
   end; {LINKINS}
function LINKREC;
 {získání NUM-té vazby}
begin
  LINKREC:=X[Y].ITEM.LINK[NUM];
   end; {LINKREC}
function HEAPNIL;
begin
   HEAPNIL:=Ø;
   end; {HEAPNIL}
begin
   end; {HEAP}
program HEAP;
 {dynamické přidělování položek typu HEAPITEM s HEAP N-možnými vazbami;
  varianta využívající paměťový prostor ovládaný proceduremi new a dispose}
 {import}
        type
           ELEMENT; {typ obsahu položky}
 {export}
        const
           HEAPN=1;  {počet vazeb položky}
           HEAPMAX=maxint;  {v této variantě nemá smysl, je uvedena jen
                             kvůli kompatibitě vývozu s předchozí variantou}
        type
           HEAPPOINT=↑HEAPITEM;  {zpřístupňující typ}
           HEAPITEM=record
              CONT:ELEMENT; {obsah}
              LINK:array[1..HEAPN] of HEAPPOINT; {vazby}
              end;
           HEADSPACE=(NIC);  {v této variantě nemá smysl, je uvedena jen
                              kvůli kompatibilitě s předchozí variantou}
        hlavičky procedur a funkcí jsou totožné s předchozí variantou
 {implementace}
procedure HEAPINIT;
begin end;  {prázdná}
function NEWX;  var POM:HEAPPOINT;
begin
   new (POM); NEWX:=POM;
   end;  {NEWX}
procedure DISPOSEX;
begin
   dispose (Y);
   end; {DISPOSEX}
procedure HEAPINS;
begin
   Y↑.CONT:=Z
   end;  {HEAPINS}
```

```
procedure HEAPREC;
begin
   Z:=Y↑.CONT;
   end; {HEAPREC}
procedure LINKINS;
begin
   Y↑.LINK[NUM]:=Z;
   end {LINKINS}
function LINKREC;
begin
   LINKREC:=Y↑.LINK[NUM];
   end; {LINKREC}
function HEAPNIL;
begin
   HEAPNIL:=nil;
   end;  {HEAPNIL}
begin
   end. {HEAP}
```

5.4.4. Lineární seznam

Při implementaci lineárního seznamu vyjdeme ze specifikace ATD z odst. 5.2.1. Datový typ LIST (lineární seznam) obsahuje složky ACT a FRST zpřístupňující v daném okamžiku vždy první a aktuální položku lineárního seznamu (viz obr. 5.29.) a položku SPACE reprezentující prostor pro dynamické přidělování položek.



Obr. 5.29. Implementace lineárního seznamu LIST

Algoritmy bloků operací musí odpovídat sémantické specifikaci ATD.

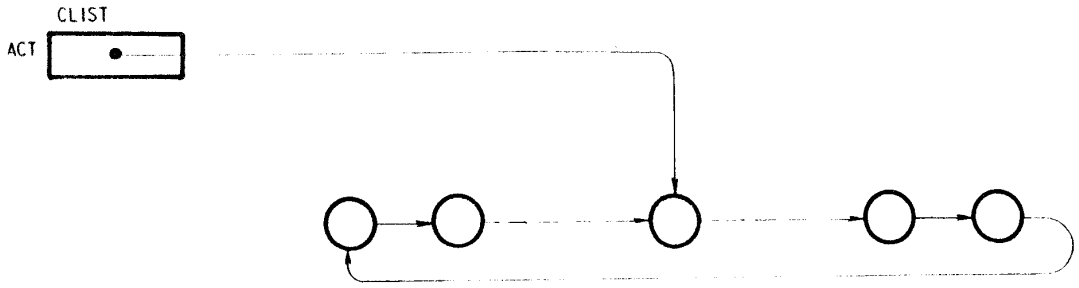
```
program LIST;
 {implementace ATD jednosměrně vázaný seznam}
 {import}
      ¤include ELEMENT; {dovoz typu elementu seznamu}
      ¤include HEAP;    {dynamické přidělování položek}
 {export}
       type
          LIST=record  {lineární seznam}
               FRST,ACT:HEAPPOINT;  {první a aktuální položka}
               SPACE:HEAPSPACE;  {prostor pro přidělování položek}
               end;
```

```
 {operace podle specifikaci ATD}
procedure LISTINIT (var X:LIST); oper;
procedure FIRST (var X:LIST); oper;
procedure SUCC (var X:LIST); oper;
procedure POSTDELETE (var X:LIST); oper;
procedure DELETEFIRST (var X:LIST); oper;
procedure COPY (var X:LIST; var Y:ELEMENT); oper;
procedure COPYFIRST (var X:LIST; var Y:ELEMENT); oper;
procedure POSTINSERT (var X:LIST; Y:ELEMENT); oper;
procedure INSERTFIRST (var X:LIST; Y:ELEMENT); oper;
procedure ACTUALIZE (var X:LIST; Y:ELEMENT); oper;
function ACTIVE (var X:LIST):Boolean; oper;
procedure LIST; module;
 {implementace}
procedure LISTINIT;
begin
   with X do begin
            HEAPINIT (SPACE);
            FRST:=HEAPNIL;
            ACT:=HEAPNIL;
            end;
   end; {LISTINIT}
procedure FIRST;
begin
   with X do
            ACT:=FRST;
   end; {FIRST}
procedure SUCC;
begin
   with X do
            if ACT<>HEAPNIL then
               ACT:=LINKREC(SPACE,ACT,1);
   end; {SUCC}
procedure POSTDELETE;
var
   POM:HEAPPOINT;
begin
   with X do
        if ACT<>HEAPNIL then begin
              POM:=LINKREC(SPACE,ACT,1);  {POM zpřístupní rušenou položku}
              if POM<>HEAPNIL then begin
                  {přemístění vazby}
                   LINKINS(SPACE,ACT,1,LINKREC(SPACE,POM,1));
                   DISPOSEX(SPACE,POM); {rušení položky}
                   end;
              end;
        end; {POSTDELETE}
procedure DELETEFIRST;
var
   POM:HEAPPOINT;
```

```
begin
   with X do
        if FRST<>HEAPNIL then begin
           if FRST=ACT then
              ACT:=HEAPNIL;
           POM:=FRST; { POM zpřístupní rušenou položku }
           FRST:=LINKREC(SPACE,FRST,1); { přemístění vazby }
           DISPOSEX(SPACE,POM); { rušení }
           end;
   end; { DELETEFIRST };
procedure COPY;
begin
   with X do
        if ACT<>HEAPNIL then
           HEAPREC(SPACE,ACT,Y) { získání hodnoty aktuální položky }
        else
            {error } HALT;
   end; { COPY }
procedure COPYFIRST;
begin
   with X do
        if FRST<>HEAPNIL then
            HEAPREC(SPACE,FRST,Y)
        else  { chyba }
            HALT;
   end;  { COPYFIRST }
procedure POSTINSERT;
var
   POM:HEAPPOINT;
begin
   with X do
        if ACT<>HEAPNIL then begin
           POM:=NEWX(SPACE); { POM zpřístupní novou položku }
           HEAPINS(SPACE,POM,Y); { uložení hodnoty nové položky }
           LINKINS(SPACE,POM,1,LINKREC(SPACE,ACT,1));
           LINKINS(SPACE,ACT,1,POM); { přemístění vazeb }
           end;
   end; { POSTINSERT }
procedure INSERTFIRST;
var
   POM:HEAPPOINT;
begin
   with X do begin
        POM:=NEWX(SPACE); { POM zpřístupní novou položku }
        HEAPINS(SPACE,POM,Y); { uložení hodnoty nové položky }
        LINKINS(SPACE,POM,1,FRST); { vytvoření vazby }
        FRST:=POM;
        end;
   end; { INSERTFIRST }
```

```
procedure ACTUALIZE;
begin
   with X do
        if ACT<>HEAPNIL then
              HEAPINS(SPACE,ACT,Y); { vložení nové hodnoty }
   end; { ACTUALIZE }
function ACTIVE;
begin
   ACTIVE:=X.ACT<>HEAPNIL;
   end; { ACTIVE }
begin
   end. { LIST }
```

Kruhový seznam CLIST (obr. 5.30.) lze implementovat obdobným způsobem jako lineární seznam. Operace pracující s první položkou nebudou však implementovány, neboť



Obr. 5.30. Kruhový seznam

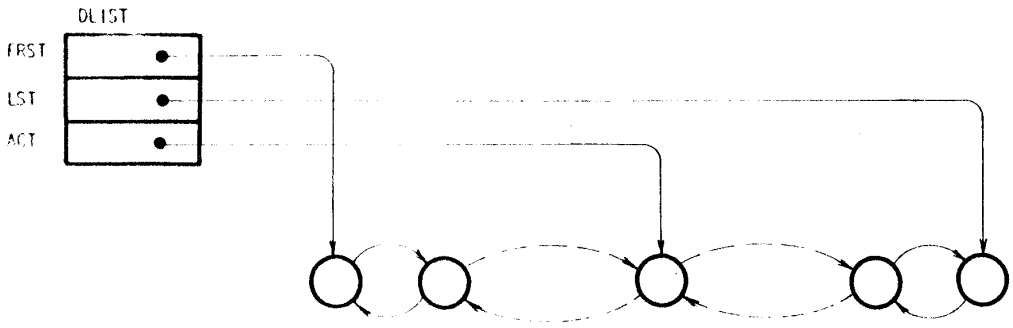
u kruhového seznamu nemá smysl první položku uvažovat. V seznamu bez aktuální položky (v prázdném seznamu) způsobí operace POSTINSERT vložení počáteční položky.

```
program CLIST;
 { implementace ATD kruhový jednosměrný seznam }
 { import }
 ¤include ELEMENT;
 ¤include HEAP;
 { export }
type
     CLIST=record  { kruhový seznam }
             ACT:HEAPPOINT;
             SPACE:HEAPSPACE;
             end; { CLIST }
procedure CLISTINIT (var X:CLIST); oper;
procedure POSTDELETE (var X:CLIST); oper;
procedure SUCC (var X:CLIST); oper;
procedure COPY (var X:CLIST; var Y:ELEMENT); oper;
procedure POSTINSERT (var X:CLIST; Y:ELEMENT); oper;
procedure ACTUALIZE (var X:CLIST; Y:ELEMENT); oper;
function ACTIVE (var X:CLIST):Boolean; oper;
procedure CLIST; module;
 { implementace }
```

```
procedure CLISTINIT;
begin
   with X do begin
        HEAPINIT(SPACE);
        ACT:=HEAPNIL;
        end;
   end; {CLISTINIT}
procedure SUCC; {totožná s operací lineárního seznamu}
procedure POSTDELETE;
var
   POM:HEAPPOINT;
begin
   with X do
        if ACT<>HEAPNIL then begin
           POM:=LINKRES(SPACE,ACT,1);
           if POM=ACT then   {poslední položka seznamu}
              ACT:=HEAPNIL
           else
             {přemístění vazby}
              LINKINS(SPACE,ACT,1,LINKREC(SPACE,POM,1));
           DISPOSEX(SPACE,POM);
           end;
   end; {POSTDELETE}
procedure COPY; {totožná s operací lineárního seznamu}
procedure POSTINSERT;
var
   POM:HEAPPOINT;
begin
   with X do begin
        POM:=NEWX(SPACE);
        HEAPINS(SPACE,POM,Y);
        if ACT=HEAPNIL then begin  {počáteční-první položka seznamu}
           ACT:=POM;
           LINKINS(SPACE,POM,1,POM);
           end
        else begin
           LINKINS(SPACE,POM,1,LINKREC(SPACE,ACT,1));
           LINKINS(SPACE,ACT,1,POM);
           end;
        end;
   end; {POSTINSERT}
procedure ACTUALIZE;{totožná s operací lineárního seznamu}
function ACTIVE;    {totožná s operací lineárního seznamu}
begin end. {CLIST}
```

5.4.5. Obousměrný seznam

Implementace ATD obousměrný seznam vyžaduje dynamické vytváření položek se dvěma vazbami (viz obr. 5.31.). Datový typ DLIST (obousměrný seznam) obsahuje mimo



Obr. 5.31. Implementace obousměrného seznamu DLIST

složky obdobné lineárnímu seznamu (ACT,FRST a SPACE) i složku zpřístupňující poslední položku seznamu (LST).

Záhlaví (popis dovozu a vývozu) implementace ATD obousměrný seznam je variantou záhlaví pro lineární seznam.

```
program DLIST;
 {implementace ATD obousměrně vázaný seznam}
 {import}
 {include ELEMENT; {typ obsahu položky}
 {include HEAP;    {modul dynamického přidělování položek se 2 vazbami}
 {export}
type
   DLIST=record  {obousměrný seznam}
      FRST,LST,ACT:HEAPPOINT;  {první,poslední,aktuální položka}
      SPACE:HEAPSPACE;
      end; {DLIST}
procedure LISTINIT (var X:DLIST); oper;
procedure FIRST (var X:DLIST); oper;
procedure LAST (var X:DLIST); oper;
procedure SUCC (var X:DLIST); oper;
procedure PRED (var X:DLIST); oper;
procedure PREDELETE (var X:DLIST); oper;
procedure POSTDELETE (var X:DLIST); oper;
procedure DELETEFIRST (var X:DLIST); oper;
procedure DELETELAST (var X:DLIST); oper;
procedure COPY (var X:DLIST; var Y:ELEMENT); oper;
procedure COPYFIRST (var X:DLIST; var Y:ELEMENT); oper;
procedure COPYLAST (var X:DLIST; var Y:ELEMENT); oper;
procedure POSTINSERT (var X:DLIST; Y:ELEMENT); oper;
procedure PREINSERT (var X:DLIST; Y:ELEMENT); oper;
procedure INSERTFIRST (var X:DLIST; Y:ELEMENT); oper;
procedure INSERTLAST (var X:DLIST; Y:ELEMENT); oper;
procedure ACTUALIZE (var X:DLIST; Y:ELEMENT); oper;
function ACTIVE (var X:DLIST):Boolean; oper;
procedure DLIST; module;
```

Implementace bloků operací ATD obousměrný seznam je velmi podobná implementaci operací ATD LIST. Z těchto důvodů ji ponecháváme čtenářům jako cvičení.

Zvláštní případ implementace obousměrného seznamu uvádí WIRTH [4]. Předpo-

kládejme implementaci datového typu DLIST statickým polem

```
type
     DLIST=array [1..MAX] of record
                DATA:ELEMENT;
                POMUK:1..MAX;
                end; { DLIST }
```

Předpokládejme dále, že tři po sobě jdoucí prvky seznamu jsou uloženy na indexech proměnné POLE typu DLIST I_{k-1} , I_k a I_{k+1}.

Pak nechť POLE $[I_k]$.POMUK = $I_{k+1} - I_{k-1}$. Z toho vyplývá, že ze znalosti indexu předchůdce (I_{k-1}) a ze znalosti indexu jemu následujícího prvku (I_k) lze určit index dalšího prvku vztahem

$$I_{k+1} = I_{k-1} + \text{POLE}[I_k] .\text{POMUK}$$

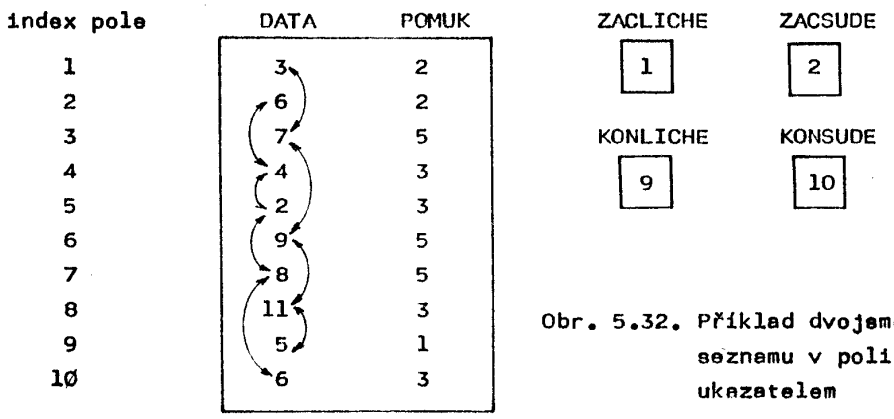
a tím zabezpečit určení následníků při průchodu od začátku ke konci a podobně vztahem

$$I_{k-1} = I_{k+1} - \text{POLE}[I_k] .\text{POMUK}$$

zabezpečit určení předchůdců při průchodu od konce k začátku.

Pro průchod od začátku platí $I_0 = I_1$, čímž se určí index neexistujícího předchůdce prvního prvku. Podobně, jestliže index následníka je vztahem určen tak, že se rovná indexu předchůdce, pak to znamená, že následník neexistuje a došlo se na konec seznamu. Stejná pravidla platí pro průchod od konce k začátku.

Příklad této implementace je uveden na obr. 5.32., kde v poli o 10 prvcích jsou implementovány dva dvojsměrné seznamy. Jeden sestává z lichých a druhý ze sudých čísel v datové složce DATA typu integer.



Obr. 5.32. Příklad dvojsměrného seznamu v poli s jedním ukazatelem

Pak pro průchod seznamem lichých čísel platí :

$I_0 = I_1 = 1$ (číslo 3)
$I_2 = I_0 + \text{POLE}[I_1].\text{POMUK}=1+2=3$ (číslo 7)
$I_3 = I_1 + \text{POLE}[I_2].\text{POMUK}=1+5=6$ (číslo 9)
$I_4 = I_2 + \text{POLE}[I_3].\text{POMUK}=3+5=8$ (číslo 11)
$I_5 = I_3 + \text{POLE}[I_4].\text{POMUK}=6+3=9$ (číslo 5)
$I_6 = I_4 + \text{POLE}[I_5].\text{POMUK}=8+1=9$

Protože $I_5 = I_6$ je prvek na indexu I_6 posledním prvkem seznamu.
Vztahu $I_0 = I_1$ lze použít obdobně pro průchod v opačném směru.

Ukazatele ZACLICHE, ZACSUDE, KONLICHE a KONSUDE v obr. 5.32. jsou ukazatelé na začátky a konce obou seznamů.

5.4.6. Zásobník

Implementaci ATD zásobník můžeme provést několika způsoby. Jednou z nejpoužívanějších metod je implementace zásobníku pomocí lineárního seznamu. Druhou často užívanou metodou je implementace zásobníku pomocí statického jednorozměrného pole. V této variantě je samozřejmě omezen maximální počet položek v zásobníku.

Nejprve uveďme variantu implementace zásobníku lineárním seznamem.

```
program STACK;
 {implementace ATD zásobník lineárním seznamem }
 {import}
 {include ELEMENT;  {typ obsluhové složky zásobníku }
 {include LIST;     {lineární seznam }
 {export}
      type
         STACK=LIST;  {zásobník }
      procedure STACKINIT (var X:STACK); oper;
      procedure PUSH (var X:STACK; Y:ELEMENT); oper;
      procedure POP (var X:STACK); oper;
      procedure TOP (var X:STACK; var Y:ELEMENT); oper;
      function SEMPTY (var X:STACK); Boolean; oper;
      procedure STACK; module;
 {implementace }
procedure STACKINIT;
begin
      LISTINIT(X);
      end; {STACKINIT }
procedure PUSH;
begin
      INSERTFIRST(X,Y);
      FIRST(X);
      end;  {PUSH }
procedure POP;
begin
      SUCC(X);
      DELETEFIRST(X);
      end; {POP }
procedure TOP;
begin
      COPY(X,Y);
      end; {TOP }
function SEMPTY;
begin
      SEMPTY:=not ACTIVE(X);
         end; {SEMPTY }
begin
      end.  {STACK }
```

Implementace zásobníku pomocí statického jednorozměrného pole by měla tento tvar (uvádíme pouze typ STACK a bloky operací).

```
const
     STACKMAX=10000;  { maximální počet položek v zásobníku }
type
     STACK=record
           TOP: 0..STACKMAX; { index vrcholu zásobníku }
             A: array [ 1..STACKMAX ] of ELEMENT;  { vlastní zásobník }
             end; { STACK }
 { implementace }
procedure STACKINIT;
begin
     X.TOP:=0;
     end; { STACKINIT }
procedure PUSH;
begin
     with X do
          if TOP < STACKMAX then begin
              TOP:=TOP+1;
              A[TOP]:=Y;
              end
          else  { chyba }
              HALT;
     end;  { PUSH }
procedure POP;
begin
     with X do
         if TOP > 0 then
             TOP:=TOP-1
         else  { chyba }
             HALT;
     end;  { POP }
procedure TOP;
begin
     with X do
          if TOP > 0 then
              Y:=A[TOP]
          else { chyba }
              HALT;
     end; { TOP }
function SEMPTY;
begin
     SEMPTY:= X.TOP=0
     end {SEMPTY }
begin end. {STACK }
```

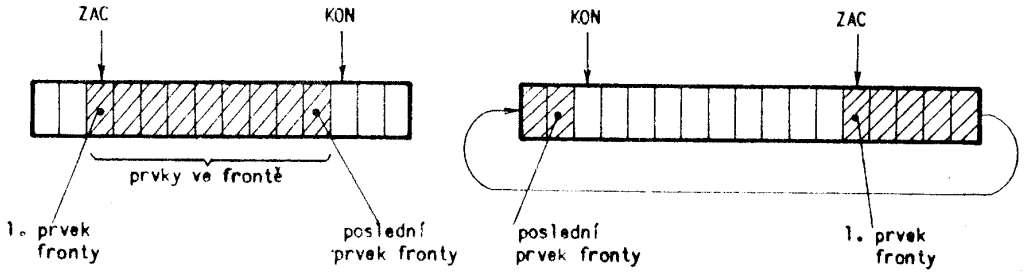
5.4.7. Fronta

Problém implementace fronty je velmi podobný problému implementace zásobníku. Také zde budeme uvažovat dva způsoby implementace a to pomocí lineárního seznamu a jednorozměrného statického pole.

Implementace využívající lineárního seznamu může mít např. tento tvar :

```
program QUEUE;
 {implementace ATD fronta lineárním seznamem}
 {import}
 ¤include ELEMENT;  {typ obsahu položek fronty}
 ¤include LIST;     {lineární seznam}
 {export}
     type
          QUEUE=LIST;
     procedure QUEUEINIT (var X:QUEUE); oper;
     procedure QUEUP (var X:QUEUE; Y:ELEMENT); oper;
     procedure FRONT (var X:QUEUE; var Y:ELEMENT); oper;
     procedure REMOVE (var X:QUEUE); oper;
     function QEMPTY (var X:QUEUE); Boolean; oper;
     procedure QUEUE; module;
      {implementace}
     procedure QUEUEINIT;
     begin
          LISTINIT(X);
          end; {QUEUEINIT}
     procedure QUEUP;
     begin
          if ACTIVE (X) then begin  {neprázdná fronta}
               POSTINSERT (X,Y);
               SUCC (X);
               end
          else begin  {prázdná fronta}
               INSERTFIRST (X,Y);
               FIRST(X);
               end;
          end; {QUEUP}
     procedure FRONT;
     begin
          COPYFIRST(X,Y);
          end; {FRONT}
     procedure REMOVE;
     begin
          DELETEFIRST (X);
          end; {REMOVE}
     function QEMPTY;
     begin
          QEMPTY:=not ACTIVE(X);
          end; {QEMPTY}
     begin
          end. {QUEUE}
```

Pro frontu implementovanou statickým polem se používají dva ukazatelé. Ukazatel ZAC ukazuje na začátek fronty, a tedy na prvek, který lze z fronty odstranit. Ukazatel KON ukazuje na konec fronty, kam se zařazuje nový prvek. Běžné stavy ukazatelů fronty znázorňuje obr. 5.33., stavy ukazatelů při plné frontě obr. 5.34. a stavy ukazatelů při prázdné frontě obr. 5.34. Z obrázku je patrné, že s polem a s ukazateli se pracuje jako s kruhovým seznamem implementovaným polem. Kapacita fronty je o jeden prvek menší, než je počet prvků pole použitého pro implementaci, jak je zřejmé z obr. 5.34.



Obr. 5.33. Běžné stavy fronty

Obr. 5.34. Stavy ukazatelů při plné frontě

Obr. 5.35. Stavy ukazatelů při prázdné frontě

Při vyváženém přidávání a odebírání prvků z fronty se ukazatelé ZAC a KON posunují v poli shodným směrem, přičemž při dosažení konce pole přecházejí opět na začátek (princip kruhového seznamu).

Modul implementace ATD fronta s využitím jednorozměrného statického pole ponecháme čtenářům jako cvičení.

Na závěr tohoto odstavce uveďme ještě implementaci ATD oboustranně ukončená fronta. K implementaci využijeme obousměrně vázaného seznamu. Modul implementace ATD oboustranně ukončená fronta - DEQUE má tento tvar :

```
program DEQUE;
 {implementace ATD oboustranně ukončená fronta obousměrně vázaným seznamem}
 {import}
¤include ELEMENT;  {typ obsahu položky}
¤include DLIST;    {oboustranně vázaný seznam}
```

```
{export}
    type
        DEQUE=LIST;
    procedure DEQUEINIT (var X:DEQUE); oper;
    procedure QUEUEFIRST (var X:DEQUE; Y:ELEMENT); oper;
    procedure QUEUELAST (var X:DEQUE; Y:ELEMENT); oper;
    procedure REMOVEFIRST (var X:DEQUE); oper;
    procedure REMOVELAST (var X:DEQUE); oper;
    procedure DCOPYFIRST (var X:DEQUE; var Y:ELEMENT); oper;
    procedure DCOPYLAST (var X:DEQUE; var Y:ELEMENT); oper;
    function DEMPTY (var X:DEQUE):Boolean; oper;
    procedure DEQUE; module;
{implementace}
    procedure DEQUEINIT;
    begin
        LISTINIT(X);
        end; {DEQUEINIT}
    procedure QUEUEFIRST;
    begin
        INSERTFIRST(X,Y);
        end; {QUEUEFIRST}
    procedure QUEUELAST;
    begin
        INSERTLAST(X,Y);
        end; {QUEUELAST}
    procedure REMOVEFIRST;
    begin
        DELETEFIRST(X);
        end; {REMOVEFIRST}
    procedure REMOVELAST;
    begin
        DELETELAST(X);
        end; {REMOVELAST}
    procedure DCOPYFIRST;
    begin
        COPYFIRST(X);
        end; {DCOPYFIRST}
    procedure DCOPYLAST;
    begin
        COPYLAST(X);
        end; {COPYLAST}
    function DEMPTY;
    begin
        DEMPTY:=not ACTIVE(X)
        end; {DEMPTY}
begin
    end. {DEQUE}
```

5.4.8. Soubor

Jak je uvedeno v odst. 5.2.1.2. lze operace ATD soubor vyjádřit operacemi ATD lineární seznam. Plyne to z podobnosti obou ATD. Hlavní rozdíl mezi těmito ATD by měl být chápán v tom, že položky souboru bývají téměř vždy ukládány do vnější paměti. S tím souvisí dlouhá doba provádění operací nad souborem, protože jejich provedení souvisí ve většině případů s aktivací modulů vstup-výstupního systému a následně i s akcemi na přídavných zařízeních. Pokud je k dispozici dostatečně velká operační paměť, je často výhodné využívat ATD soubor, jehož položky jsou uloženy v operační paměti, protože operace nad ním jsou rychlé.

Následující implementace ukazuje možnost implementace souboru lineárním seznamem. Jeho položky jsou tedy uloženy v dynamicky přidělované části operační paměti. Jména operací a typu využívané pro standardní soubor (file, reset, atd.) mají jako příponu písmeno X (FILEX, RESETX atd.) z důvodů odstranění kolize identifikátorů s klíčovými slovy a identifikátory PASCALu. Typ FILEX (soubor) obsahuje mimo lineární seznam L reprezentující vlastní soubor také složky MODE, EOFFLAG a BUFFER. Složka MODE vyjadřuje stav (modus) souboru (writing =zápis, reading= čtení). Složka EOFFLAG reprezentuje hodnotu predikátu EOF. Složka BUFFER reprezentuje přístupovou proměnnou. Jelikož nelze užít standardního pascalovského zápisu přístupové proměnné ⟨soubor⟩↑, jsme nuceni zavést operace zápisu a čtení hodnoty přístupové proměnné PUTBUF a GETBUF.

```
program FILEX;
 {implementace ATD soubor lineárním seznamem}
 {import}
 ¤include ELEMENT; {typ obsahu položky seznamu}
 ¤include LIST;
 {export}
type
     FILEX=record
          MODE:  (WRITING,READING);  {stav souboru}
          EOFFLAG:Boolean;           {příznak EOF}
          BUFFER:ELEMENT;            {přístupová proměnná}
          L:LIST;                    {seznam}
          end;
procedure REWRITEX (var X:FILEX); oper;
procedure RESETX (var X:FILEX); oper;
procedure PUTX (var X:FILEX); oper;
procedure GETX (var X:FILEX); oper;
function EOFX (var X:FILEX); Boolean; oper;
procedure PUTBUF (var X:FILEX; Y:ELEMENT); oper;
procedure GETBUF (var X:FILEX; var Y:ELEMENT); oper;
procedure FILEX; module;
 {implementace}
procedure REWRITEX;
begin
     with X do begin
          MODE:=WRITTING;
          EOFFLAG:=true;
          LISTINIT(L);
          end;
     end;  {REWRITEX}
```

```
procedure RESETX;
begin
     with X do begin
          MODE:=READING;
          FIRST(L);
          EOFFLAG:=not ACTIVE(L);
          if not EOFFLAG then
               COPY (L,BUFFER);
          end;
     end; { RESETX }
procedure PUTX;
begin
     with X do
          if MODE=WRITTING then
               if not ACTIVE(L) then begin  {dosud neexistuje žádná položka}
                    INSERFIRST(L,BUFFER);
                    FIRST(L);
                    end
               else begin  {zápis na konec seznamu}
                    POSTINSERT(L,BUFFER)
                    SUCC(L);
                    end
          else  {chyba}
                HALT;
     end; {PUTX}
procedure GETX;
begin
     with X do
          if (MODE=READING) and (not EOFFLAG) then begin
             SUCC(L);
             EOFFLAG:=not ACTIVE (L);
             if not EOFFLAG then
                    COPY (L,BUFFER);
             end
          else {chyba}
             HALT;
     end; {GETX}
function EOFX;
begin
     EOFX:=X.EOFFLAG;
     end;  {EOFX }
procedure PUTBUF;
begin
     X.BUFFER:=Y;
     end; {PUTBUF }
procedure GETBUF;
begin
     Y:=X.BUFFER;
     end;  {GETBUF }
begin end.  {FILEX }
```

5.4.9. Tabulka

Variant implementace vyhledávací tabulky je mnoho a algoritmy vyhledávání jsou často využívané a důležité. Z těchto důvodů je problému vyhledávání věnována samostatná kapitola těchto skript. Uveďme proto na tomto místě jen velmi jednoduchou implementaci tabulky obousměrně vázaným seznamem, kde vyhledávání je prováděno postupným srovnáním hledaného klíče s klíči všech položek tabulky (seznamu). Tento způsob vyhledávání je při větším počtu položek tabulky značně pomalý. Bylo by vhodné, aby čtenář po prostudování kapitoly o vyhledávání provedl v rámci cvičení i implementaci efektivnějších algoritmů.

Před implementací ATD tabulka popišme ATD klíč KEY, což je typ klíče tabulky. Předpokládejme, že tento ATD vyváží datový typ KEY a relační operace EQ, NE, GT, LT, GE, LE nad dvěma proměnnými typu KEY s významem $=$, $\neq$, $>$, $<$, $\geqslant$ a $\leqslant$.

```
program KEY;
 {typ klíče tabulky}
 :
 {export}
type
     KEY=....;  {klíč}
function EQ (var X,Y:KEY):Boolean; oper; { = }
function NE (var X,Y:KEY):Boolean; oper; {< >}
function GT (var X,Y:KEY):Boolean; oper; { > }
function LT (var X,Y:KEY):Boolean; oper; { < }
function GE (var X,Y:KEY):Boolean; oper; { >= }
function LE (var X,Y:KEY):Boolean; oper; {<= }
procedure KEY; module;
 {implementace není pro nás podstatná, proto ji neuvádíme}
begin end.  {KEY}
```

Mimo to doplňme další předpoklad o datovém typu ELEMENT. Doposud jsme nepředpokládali žádné vlastnosti tohoto typu. Pro implementaci tabulky předpokládejme, že typ ELEMENT je záznam s položkou k typu KEY.

```
type ELEMENT = record
                 :
               K:KEY; {klíč}
                 :
               end;
```

Toto omezení kladené na typ ELEMENT vyplývá z nutnosti mít k dispozici speciální složku (klíč) potřebnou pro vyhledávání. Implementace ATD tabulka může pak mít např. tento tvar :

```
program TABLE;
 {vyhledávací tabulka implementovaná lineárním seznamem}
 {import}
 ¤include KEY;  {klíč}
 ¤include ELEMENT; {obsah položky tabulky}
 ¤include DLIST; {obousměrně vázaný seznam}
 {export}
type
```

```
     TABLE=DLIST;
procedure TABLEINIT (var X:TABLE); oper;
procedure INSERT (var X:TABLE; Y:ELEMENT); oper;
procedure TREAD (var X:TABLE; Y:KEY; var Z:ELEMENT); oper;
procedure DELETE (var X:TABLE; Y:KEY); oper;
function SEARCH (var X:TABLE; Y:KEY); Boolean; oper;
procedure TABLE; module;
function SEARCH;
var
     FOUND:Boolean;
     A:ELEMENT;
begin
     with X do begin
          FIRST(X);
          FOUND:=false;
          while (not FOUND) and (ACTIVE(X)) do begin { vyhledávání }
               COPY(X,A);
               if EQ(Y,A,K) then
                  FOUND:=true
               else
                  SUCC(X);
               end;
          end;
     SEARCH:=FOUND;
     end; { SEARCH }
procedure TABLEINIT;
begin
     DLISTINIT(X);
     end; { TABLEINIT }
procedure INSERT;
begin
     if SEARCH(X,Y,K) then { prvek s klíčem je již v tabulce }
          ACTUALIZE(X,Y)
     else
          INSERTFIRST(X,Y)
     end; { INSERT }
procedure TREAD;
begin
     if SEARCH(X,Y) then { prvek je v tabulce }
          COPY(X,Z)
     else { není v tabulce-chyba }
          HALT;
     end; { TREAD }
procedure DELETE;
begin
     if SEARCH(X,Y) then begin
          PRED(X); { posun k předchozí položce }
          if ACTIVE(X) then { nebyla to první položka }
               POSTDELETE(X)
          else { zrušit první položku }
                 DELETEFIRST(X);
```

```
    end; {DELETE}
begin end. {TABLE}
```

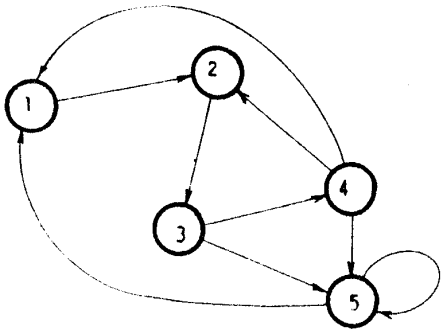
5.4.10. Graf a binární strom

Pro implementaci ATD orientovaný graf lze zvolit buď sekvenční nebo zřetězené umístění uzlů v paměti. Při sekvenčním umístění představuje každý uzel jeden řádek matice a prvky řádku jsou ukazatele na ostatní uzly. (Čtvercová matice umožňuje, aby z každého uzlu vedla jedna orientovaná hrana do každého uzlu). Při zřetězeném umístění představuje každý uzel zřetězený lineární seznam ukazatelů, který má tolik položek, kolik orientovaných hran z uzlu vychází.

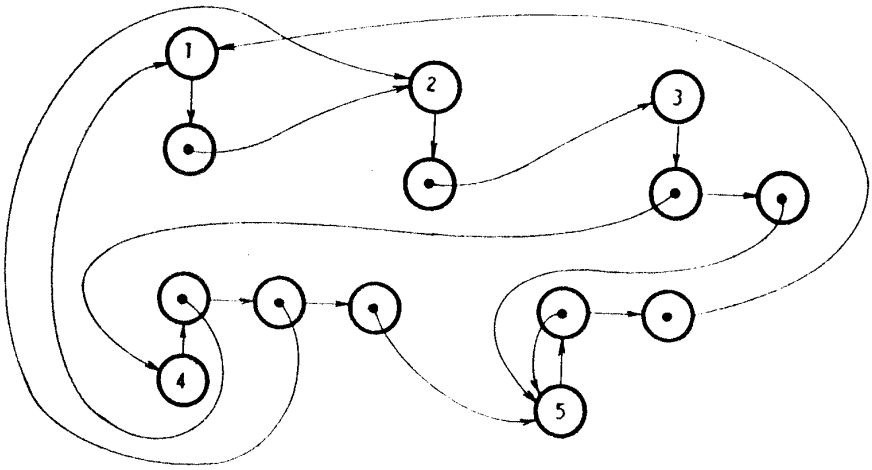
Implementaci sekvenčním umístěním použijeme tehdy, je-li znám pevný nebo maximální počet uzlů a nejsou-li k dispozici pohodlné prostředky dynamického přidělování paměti. Již samotný význam pojmu orientovaný graf a jeho grafické vyjádření napovídá, že typickým zobrazením je struktura dynamicky vytvářených uzlů propojených ukazatelů. Na obr. 5.36. je orientovaný graf o 5 uzlech a na obr. 5.37. a 5.38. je zobrazení této struktury sekvenčním a zřetězeným umístěním v paměti.

	1	2	3	4	5
1	ø	1	ø	ø	ø
2	ø	ø	1	ø	ø
3	ø	ø	ø	1	1
4	1	1	ø	ø	1
5	1	ø	ø	ø	1

Obr. 5.37. Matice zobrazení orientovaného grafu z obr. 5.27.



Obr. 5.36. Orientovaný graf o 5 uzlech



Obr. 5.37. Orientovaný graf z obr. 5.36. implementovaný seznamy

Ze zvolené reprezentace orientovaného grafu pak vyplývá i implementace datového typu a operací popsaných v odst. 5.2.6.1. Implementaci ATD obecný orientovaný graf zde nebudeme uvádět a soustředíme se na častěji využívaný speciální případ binárního stromu. Binární strom je speciálním případem orientovaného grafu v němž počet hran vycházejících z uzlu je omezen maximálně na dvě. Z toho plyne, že není nutné vytvářet seznam ukazatelů na následovníky uzlu, protože jejich maximální počet je předem znám. Proto pro implementaci využijeme stejné dynamické položky se dvěma vazbami, jako pro implementaci ATD obousměrně vázaný seznam. První vazba bude vždy reprezentovat hranu směrující k levému a druhá k pravému následovníku. Významným uzlem binárního stromu je kořen, proto datový typ bude zpřístupňovat tento uzel stromu. Implementace ATD binární strom bude mít pak tento tvar :

```
program STROM;
 {implementace ATD binární strom}
 {import}
 {include ELEMENT;  {obsahová složka uzlu}
 {include HEAP;     {dynamické přidělování položek se dvěma vazbami}
 {export}
type
     UZEL=HEAPPOINT;
     STROM=record
          KOREN:HEAPPOINT;  {kořen}
          SPACE:HEAPSPACE;  {prostor pro přidělování položek}
          end;
procedure STROMINIT (var X:STROM); oper;
function STROMEMPTY (var X:STROM):Boolean; oper;
function NOVYUZEL (var X:STROM):UZEL; oper;
procedure ZRUSUZEL (var X:STROM; Y:UZEL); oper;
procedure PUVPRAVO (var X:STROM; Y,Z:UZEL); oper;
procedure PUVLEVO (var X:STROM; Y,Z:UZEL); oper;
procedure PKOREN (var X:STROM; Y:UZEL); oper;
function KOREN (var X:STROM):UZEL; oper;
function POSUNVLEVO (var X:STROM;Y:UZEL):UZEL; oper;
function POSUNVPRAVO (var X:STROM; Y:UZEL):UZEL; oper;
procedure STROMCOPY (var X:STROM; Y:UZEL; var Z:ELEMENT); oper;
procedure STROMACT (var X:STROM; Y:UZEL; Z:ELEMENT); oper;
function JEVLEVO (var X:STROM; Y:UZEL):Boolean; oper;
function JEVPRAVO (var X:STROM; Y:UZEL):Boolean; oper;
procedure STROM; module;
 {implementace}
procedure STROMINIT;
begin
     with X do begin
          KOREN:=HEAPNIL;
          HEAPINIT(SPACE);
          end;
     end;  {STROMINIT}
```

```
function STROMEMPTY;
begin
     STROMEMPTY:=X.KOREN=HEAPNIL;
     end;  { STROMEMPTY }
function NOVYUZEL;
var POM:HEAPPOINT;
begin
     with X do begin
       POM:=NEWX(SPACE);
       LINKINS(SPACE,POM,1,HEAPNIL);
       LINKINS(SPACE,POM,2,HEAPNIL);
       NOVYUZEL:=POM;
       end;
     end;
procedure ZRUSUZEL;
begin
     DISPOSEX(SPACE,Y);
     end;  { ZRUSUZEL }
procedure PUVPRAVO;
begin
     with X do LINKINS(SPACE,Y,2,Z);
     end;  { PUVPRAVO }
procedure PUVLEVO;
begin
     with X do LINKINS(SPACE,Y,1,2);
     end;  { PUVLEVO }
procedure PKOREN;
begin
     with X do begin
          KOREN:=Y;
          end;
     end;  { PKOREN }
function KOREN;
begin
     if X.KOREN <>HEAPNIL then
          KOREN:=X.KOREN
     else  { chyba }
            HALT
     end;  { KOREN }
function  POSUNVLEVO;
begin
     with X do
          if JEVLEVO(X,Y) then POSUNVLEVO:=LINKREC(SPACE,Y,1)
          else  { chyba }  HALT;
     end;  { POSUNVLEVO }
function  POSUNVPRAVO;
begin
     with X do
       if JEVPRAVO(X,Y) then POSUNVPRAVO:=LINKREC(SPACE,Y,2)
       else  { chyba }  HALT;
     end;  { POSUNVPRAVO }
```

```
procedure STROMCOPY;
begin
     with X do HEAPREC(SPACE,Y,Z);
     end; {STROMCOPY}
procedure STROMACT;
begin
     with X do HEAPINS(SPACE,Y,Z);
     end;  {STROMACT}
function JEVLEVO;
begin
     with X do JEVLEVO:=LINKREC(SPACE,Y,1)<>HEAPNIL;
     end; {JEVLEVO}
function JEVPRAVO;
begin
     with X do JEVPRAVO:=LINKREC(SPACE,Y,2)<>HEAPNIL;
     end; {JEVPRAVO}
begin end.  {STROM}
```

Mimo základní operace nad **binárním** stromem popíšeme nyní také implementaci operace průchodů stromem, tj. transformace binárního stromu na lineární seznam. Budeme předpokládat, že binární strom i seznam mají položky téhož datového typu ELEMENT.

Rekurzivní zápis operací průchodu stromem je velmi jednoduchý. Varianty PREORDER, INORDER a POSTORDER se liší pouze v umístění operací zapisujících do seznamu. Zapíšme proto operaci ORDER se třemi výstupními seznamy PREORDER, INORDER a POSTORDER odpovídajícími jednotlivým typům průchodu.

Definujme nejdříve operaci OUT vytvářející novou položku na konci lineárního seznamu L, přičemž její hodnota je dána hodnotou zadaného uzlu Y binárního stromu X.

```
procedure OUT (var X:STROM; Y:UZEL; var L:LIST);
var
     A:ELEMENT;
begin
     STROMCOPY(X,Y,A);
     if ACTIVE(L) then begin
          POSTINSERT(L,A);
          SUCC(L);
          end
     else begin
          INSERTFIRST(L,A);
          FIRST(L);
          end;
     end; {OUT}
```

S využitím této operace zapíšeme nyní operaci vytvářející všechny tři průchody stromem, přičemž je využito rekurzivního volání vnitřní procedury REKURZE.

```
procedure ORDER (var X:STROM; var PREORDER,INORDER,POSTORDER:LIST);
 {průchody binárním stromem; varianta využívající rekurzivní procedury}
procedure REKURZE (Y:UZEL);
begin
```

```
        {ZAČÁTEK}
        OUT(X,Y,PREORDER);
        if JEVLEVO(X,Y) then
            REKURZE(POSUNVLEVO(X,Y));
        {STŘED}
        OUT(X,Y,INORDER);
        if JEVPRAVO(X,Y) then
           REKURZE(POSUNVPRAVO(X,Y)));
        {KONEC}
        OUT(X,Y,POSTORDER);
        end; {REKURZE}
    begin
        LISTINIT(PREORDER);
        LISTINIT(INORDER);
        LISTINIT(POSTORDER);
        REKURZE(KOREN(X));
        end; {ORDER}
```

Každé volání procedury RECURZE představuje v této variantě implementace průchodů stromem zpracování jednoho uzlu stromu. Zpracování uzlu má 3 fáze nazvané ZAČÁTEK, STŘED a KONEC. Fáze ZAČÁTKU představuje akce prováděné před zpracováním podstromu levého následovníka, fáze STŘED akce prováděné mezi zpracováním podstromů levého a pravého následovníka a fáze KONEC akce provedené po zpracování podstromu pravého následovníka daného uzlu. Typickou akcí začátku je výstup PREORDER, středu výstup INORDER a konce výstup POSTORDER.

Ve fázi ZAČÁTEK se po výstupu PREORDER provádí přechod na levého následovníka daného uzlu, pokud existuje. Přechodem rozumíme započetí fáze ZAČÁTEK pro další uzel (následovníka) a zároveň uložení stavu rozpracovanosti původního uzlu do implicitního zásobníku poskytovaného mechanizmem rekurzivních procedur. Po návratu ze zpracování následovníka pokračuje zpracování uvažovaného uzlu další fází STŘED. Pokud levý následovník neexistuje provádí se pokračování fází STŘED ihned.

Fáze STŘED má obdobnou strukturu. Po výstupu INORDER se provádí přechod na pravého následovníka, pokud existuje. Po jeho zpracování, případně neexistuje-li, provádí se přechod na fázi KONEC.

Ve fázi KONEC je prováděn výstup POSTORDER a návrat k fázi, kterou se pokračuje ve zpracování předchůdce uvažovaného uzlu. Je-li uvažovaným uzlem kořen je operace ORDER ukončena.

Nerekurzivní varianta operace ORDER nemůže využít implicitního zásobníku poskytovaného mechanismem rekurzivních procedur. V něm, jak bylo řečeno se ukládá stav rozpracování uvažovaného uzlu. Tento zásobník musíme nyní explicitně popsat a ovládat. Elementem zásobníku bude datový typ

```
type  ELEMENTSTACK=record
        CO:UZEL;  {uvažovaný uzel}
        KAM:(LEVY,PRAVY);  {stav rozpracování}
        end;
```

Složka CO určuje uvažovaný uzel a složka KAM určuje, zda bylo zpracování uzlu přerušeno po volání levého následníka, tj. před fází STŘED nebo po volání pravého

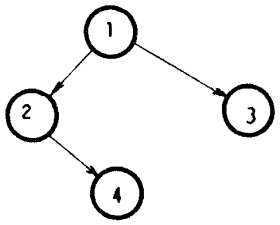
následníka tj. před fází KONEC.

S využitím zásobníku S položek popsaného typu lze implementovat operaci nerekurzivně takto :

```
procedure ORDER (var X:STROM; var PREORDER, INORDER, POSTORDER:LIST);
 {průchody binárním stromem; nerekurzivní varianta }
var
     Y:UZEL;    {uvažovaný uzel }
     RUN:Boolean;
     FAZE:(ZACATEK,STRED,KONEC);
     STE:ELEMENTSTACK; {položka explicitního zásobníku }
     S:STACK;          {explicitní zásobník }
begin
     STACKINIT(S);
     LISTINIT(PREORDER);
     LISTINIT(INORDER);
     LISTINIT(POSTORDER);
     RUN:=true; {inicializace }
     FAZE:=ZACATEK;   {začíná se ve fázi začátek }
     Y:=KOREN(X);     {kořene stromu }
     while RUN do
          case FAZE of
               ZACATEK:begin
                   OUT(X,Y,PREORDER);
                   if JEVLEVO(X,Y) then begin
                        with STE do begin
                             KAM:=LEVY;
                             CO:=Y;
                             end;
                        PUSH(S,STE);
                        Y:=POSUNVLEVO(X,Y);
                        FAZE:=ZACATEK;
                        end
                   else
                        FAZE:=STRED;
                   end;
               STRED:begin
                   OUT(X,Y,INORDER);
                  if JEVPRAVO(X,Y) then begin
                        with STE do begin
                             KAM:=PRAVY;
                             CO:=Y;
                             end;
                        PUSH(S,STE);
                        Y:=POSUNVPRAVO(X,Y);
                        FAZE:=ZACATEK end
                   else
                        FAZE:=KONEC;
                   end;
               KONEC:begin
```

```
                OUT(X,Y,POSTORDER);
                if SEMPTY(S) then
                    RUN:=false
                else begin
                     TOP(S,STE);
                     if STE.KAM=LEVY then
                        FAZE:= STRED
                     else
                        FAZE:=KONEC;
                     Y:=STE.CO;
                     POP(S);
                     end;
                end; {KONEC}
            end; {case}
        end; {ORDER}
```

Na obr. 5.39. je lineární strom se 4 uzly fáze zpracování a stav zásobníku při vytváření přechodů



FÁZE:	1Z	2Z	2S	4Z	4S	4K	2K	1S	3Z	3S	3K	1K
VÝSTUP PREORDER	1	2	-	4	-	-	-	-	3	-	-	-
VÝSTUP INORDER	-	-	2	-	4	-	-	1	-	3	-	-
VÝSTUP POSTORDER	-	-	-	-	-	4	2	-	-	-	3	1
ZÁSOBNÍK	-	1L	1L	1L 2P	1L 2P	1L 2P	1L	-	1P	1P	1P	

Z ... začátek S ... střed K ... konec
L ... levý P ... pravý

Obr. 5.39. Demonstrace nerekurzivního průchodu binárním stromem

5.5. Literatura

[1] Gruska,J., Wiedermann,J., Černý,A.: Typy a štruktúry dat, Referát zborníku semináře SOFSEM 78', Bratislava,1978.

[2] Müller,K. : Programovací jazyky, skriptum FEL ČVUT, Praha, 1981.

[3] Knuth, D.E.: The Art of Computer Programming, Vol.1. Fundamental Algorithms Addison-Wesley Publishing Company, 1968.

[4] Wirth,N.: Algoritmus + Data Structures = Programs Prentice-Hall, 1976.

[5] Hopgood,F.R.A.: Metody kompilovania, Alfa Bratislava, 1973.

[6] Koster,C.H.A.: Visibility and types Sigplan Notices, Vol.11, 1976.

[7] Rábová,Z., Češka,M., Honzík,J., Hruška,T. : Počítače a programování, skriptum FE VUT, SNTL, Praha, 1982.

[8] Češka,M., Rábová,Z., Hruška,T., Máčel,M. : Gramatiky a jazyky skriptum FE VUT, SNTL, Praha, 1981.

[9] Honzík,J., Hruška,T., Zbořil,F. : Strojově orientované jazyky skriptum FE VUT, SNTL Praha 1983

[10] Kolář,J. : Algebra a grafy skriptum FEL ČVUT, Ediční středisko ČVUT Praha, 1982.