TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta strojní **Katedra aplikované kybernetiky**

prof. Ing. Vladimír Věchet, CSc.

ALGORITMY A DATOVÉ STRUKTURY

Zásobníky, fronty a seznamy

Liberec 2004

Recenzoval : prof. RNDr. Ing. Miloslav Košek, CSc.

ISBN 80-7083-828-0

OBSAH :

Motto : Každý program je zastaralý ve chvíli, kdy je vložen do počítače.

Ze zákonů Murphyho

1. ÚVOD

Předkládaný učební text je určen posluchačům IV. ročníku strojní fakulty, oboru Automatizované systémy řízení ve strojírenství. Organicky doplňuje již vydané učební texty pro předmět "Algoritmy a datové struktury", které byly věnovány rekursi, datovým strukturám stromy, grafy a tabulky a též i třídění.

Obsah učebního textu navazuje především na znalosti získané předchozím studiem předmětu "Programovací jazyky a operační systémy" s cílem rozšířit znalosti a používání základních datových typů o složitější datové struktury, jako jsou zásobníky, fronty, seznamy, stromy, grafy a tabulky.

Specifikace abstraktních datových typů se provádí výčtem operací, které generují nebo používají hodnoty takového datového typu. Abstraktní datové typy, jakož i jejich algebraická specifikace jsou předmětem zkoumání teorie programování. Z hlediska našich potřeb vystačíme se zjednodušenou sémantikou operací se zkoumanými datovými typy, nicméně jejich algebraická specifikace (která je ve zkoumaných případech snadno pochopitelná) je uvedena v příloze. Pro hlubší studium této problematiky lze doporučit uvedenou literaturu, kde čtenář najde mnoho dalších odkazů na specializovanou literaturu.

Algoritmy jsou zapisovány v jazyku C, byť jejich uvedení někdy přímo vybízí k zápisu např. v Pascalu. To je z důvodů návaznosti jednotlivých předmětů vyučovaných v oboru Automatizované systémy řízení ve strojírenství a procvičení použití tohoto programovacího jazyka. Při případném použití uváděných fragmentů zdrojových textů programů je jejich úprava s ohledem na konkrétní použití, doplnění standardních hlavičkových souborů atd. věcí čtenáře. V každém případě se předpokládá samostatné řešení příkladů, uvedených na závěr každé kapitoly.

Za pečlivé přečtení rukopisu a korektury textu děkuje autor prof. RNDr. Ing. Miloslavu Koškovi, CSc.

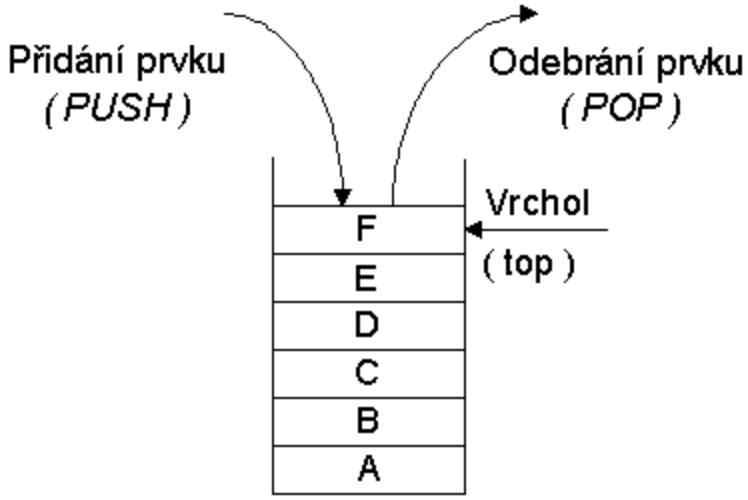
2. ZÁSOBNÍK

2.1 Specifikace zásobníku

Nejjednodušší obecnou datovou strukturou je zásobník (angl. stack) jako posloupnost prvků nějakého typu, přičemž přístup k této posloupnosti je pouze na jednom jejím konci, který se nazývá vrchol (angl. top) zásobníku. Obsah zásobníku lze měnit jen na tomto jednom konci pomocí dvou základních operací (viz obr. 1) :

- přidání prvku na vrchol zásobníku,
- odebrání prvku z vrcholu zásobníku.

Podle toho se zásobník též nazývá **pamětí LIFO** (z anglického last-in, first-out, což znamená "poslední dovnitř", "první ven").

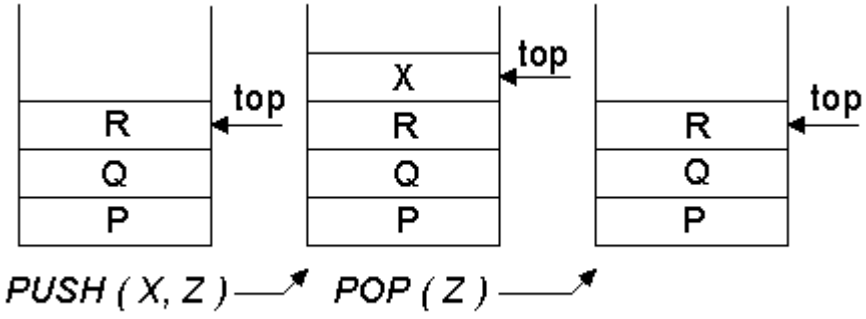


obr. 1

Operace přidání, resp. odebrání prvku ze zásobníku se zpravidla i v naší literatuře označují anglickými názvy *push*, resp. *pop*. Operaci přidání prvku na vrchol zásobníku označíme symbolicky *PUSH (x, Z)* a znamená tedy přidání prvku *x* na vrchol zásobníku *Z*. Naopak pomocí *POP (Z)* označíme odebrání prvku, který je na vrcholu zásobníku *Z* (viz schéma na obr. 2).

Je logické, že operaci odebrání prvku ze zásobníku lze použít jen tehdy, když je zásobník neprázdný, tj. obsahuje alespoň jeden prvek. Je tedy výhodné zavést další operaci s touto datovou strukturou, symbolicky označovanou *EMPTY? (Z)* (empty znamená prázdný) pro rozlišení prázdného a neprázdného zásobníku. Otazník ve jméně znamená, že se ve skutečnosti jedná o predikát. Je-li tedy zásobník *Z* prázdný, vrací predikát *EMPTY? (Z)* hodnotu *true*, jinak hodnotu *false*. Pro čtení prvku na vrcholu zásobníku se zavádí operace *TOP (Z),* která vrací prvek na

vrcholu zásobníku jako funkční hodnotu a konečně operaci *NEW,* pro vytvoření nového zásobníku (nulární operace, generátor typu).



obr. 2

Algebraická specifikace typu zásobník je uvedena v příloze (podrobněji viz např. [2]). U datových struktur specifikovaných jako posloupnosti nějakých prvků (tedy i u front a seznamů) vystačíme z hlediska dalších potřeb se zjednodušenou sémantikou operací. Označme nejprve $< z_1 z_2 \ldots\ldots z_n >$ posloupnost nějakých prvků $z_1, z_2, \ldots .. , z_n$. Pak tedy <> znamená prázdnou posloupnost prvků a potom můžeme zapsat :

$NEW = <>$,

$EMPTY?\ (<>) = true$,

$PUSH\ (z_{n+1}, < z_1\ z_2\ \ldots\ldots\ldots\ldots\ z_n >) = < z_1\ z_2\ \ldots\ldots\ z_n\ z_{n+1} >$

a pro $n \geq 1$

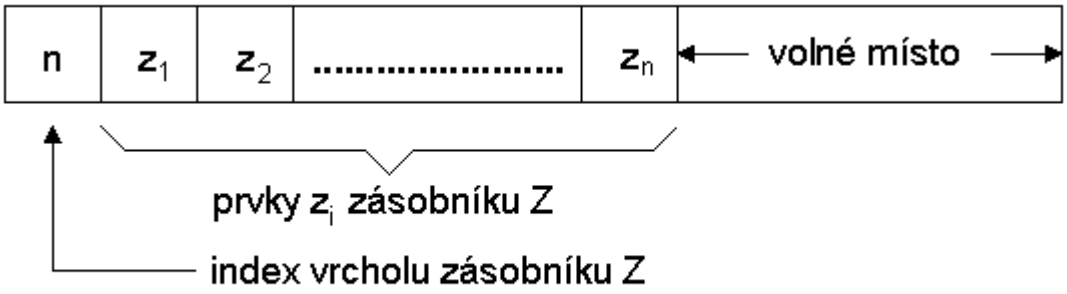
$POP\ (< z_1\ z_2\ \ldots\ldots\ldots\ z_n >) = < z_1\ z_2\ \ldots\ldots\ldots\ z_{n-1} >$,

$TOP\ (< z_1\ z_2\ \ldots\ldots\ldots\ z_n >) = z_n$

$EMPTY?\ (< z_1\ z_2\ \ldots\ldots\ldots\ z_n >) = false$.

2.2 Implementace zásobníku

Vzhledem ke specifikaci zásobníku jako posloupnosti prvků se přímo nabízí implementace zásobníku pomocí pole. Tak např. v jazyku PASCAL je pak možná deklarace záznamu obsahujícího dvě položky : pole, kam se budou ukládat prvky zásobníku a proměnná celočíselného typu, která bude indikovat aktuální polohu vrcholu zásobníku v poli (viz schéma na obr. 3). Procedurální vyjádření uvedených základních operací se zásobníkem je velmi jednoduché (viz např. [6]) a je ponecháno na čtenáři. V této souvislosti je třeba zdůraznit, že je třeba zásobník implementovat vždy v daném kontextu použití, čtenář si vždy sám přizpůsobí publikovanou implementaci svým konkrétním potřebám. To ostatně platí i pro další uváděné datové struktury.



obr. 3

Uvažujme následující implementaci zásobníku pomocí pole uvedenou v [1] :

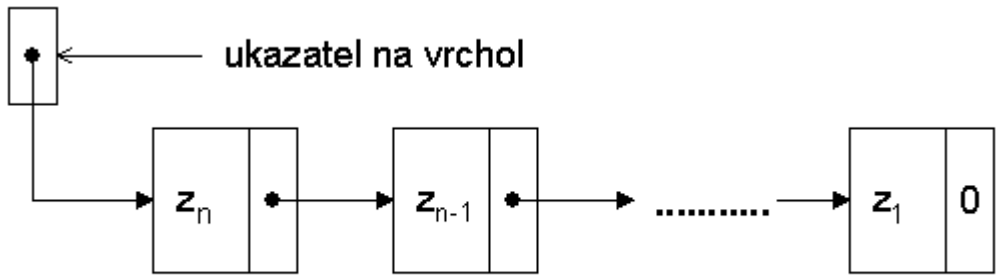
```
#define  MAX  100

int sp = 0   ;  /* "stack pointer" , ukazatel zásobníku */
double hod[MAX]  ;  /* zásobník čísel v pohyblivé řád. čárce */

void clear ( void )  {  sp = 0 ;  }

double push ( double x ) {   /* lze psát i void push .. */
        if ( sp < MAX )  return (  hod[sp++] = x ) ;
        else   {
                    printf ( "Plny zasobnik\n" ) ;
                    clear() ; return ( 0 ) ;
        }
}

double pop ( void ) {
        if ( sp )  return ( hod[--sp] ) ;
        else   {
                    printf ( "Prazdny zasobnik\n" ) ;
                    return ( 0 ) ;
        }
}
```

Především ukazatel zásobníku *sp* neindikuje složku pole *hod,* kde je vrchol zásobníku, ale první volnou složku pole *hod.* Podle toho by si čtenář musel event. zapsat svoji implementaci operace *EMPTY? (Z)*. Dále nechť čtenář uváží následující modifikace :

- V obecnějším zápisu se do zásobníku budou ukládat prvky nějakého typu, bylo by tedy vhodné tento typ definovat zvlášť,
- Ošetření chybových stavů je provedeno tak, že odpovídající zpráva je vynesena na standardní výstup, event. je zásobník znovu

inicializován a do místa volání je vrácena *0*. Ovšem i tato vrácená hodnota může představovat validní výsledek. V každém případě je třeba odpovídající části zápisů funkcí *push* a *pop* zapsat kontextuálně.

- Není implementována operace *TOP (Z),* naopak však funkce *pop* vrací do místa volání hodnotu na vrcholu původního zásobníku, takže tuto operaci lze zapsat i jako *push (pop()).*



obr. 4

Uvedená implementace zásobníku vyžaduje explicitní deklaraci délky zásobníku (konstanta *MAX*). Pokud lze délku zásobníku stanovit přesně, nebo ani není důležitá, lze takový způsob akceptovat. V opačném případě by bylo nutné použít pole o proměnné délce a zvětšovat podle potřeby velikost alokované paměti pomocí standardní funkce *realloc(),* nebo použít jednoduchý lineární spojový seznam (viz schéma na obr. 4). V tom případě jsme omezeni pouze velikostí volné paměti, i když zase na druhé straně je nutné kromě vlastních hodnot prvků zásobníku uchovávat i hodnoty typu ukazatel, což nároky na paměť zvyšuje. Jako příklad uvažujme, že prvky vkládané do zásobníku jsou řetězce. Pak jeden z možných zápisů operací se zásobníkem by mohl vypadat následovně :

```
typedef struct prvek {
        char hod[9] ;
        struct prvek *spoj ;
} PRVEK ;

PRVEK *sp = NULL ;

void clear ( void ) {
        PRVEK *p ;

        while ( sp ) {
                p = sp ; sp = sp->spoj ; free ( p ) ;
        }
}
```

```
void push ( char *x ) {
        PRVEK *p = (PRVEK *) malloc ( sizeof ( PRVEK ) ) ;

        strcpy ( p->hod, x ) ; p->spoj = sp ; sp = p ;
}

char *pop ( void ) {
        PRVEK *p ;
        char *s = (char *) malloc ( 9 ) ;

        p = sp ; strcpy ( s, sp->hod ) ; sp = sp->spoj ; free ( p ) ;
        return ( s ) ;
}
```

2.3 Příklad použití zásobníku

Existuje celá řada aplikací zásobníku. Tak např. aplikace zásobníku představuje softwarový mechanismus podpory rekurse a tomu je věnován celý jeden díl skript pro předmět “Algoritmy a datové struktury“. V dalším textu je jako příklad uvedeno použití zásobníku pro konverzi infixového na postfixový zápis výrazu.

Uvažujme operandy jako písmena a binární aritmetické operátory +, -, *, / se standardní prioritou (tedy multiplikativní operátory mají vyšší prioritu než aditivní a při stejné prioritě se vyhodnocení provádí zleva doprava). Uveďme nejprve příklady infixových zápisů výrazů a jejich postfixových ekvivalentů :

Infix :	Postfix :
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
(A + B) * (C – D)	A B + C D - *
(A + B) * (C + D – E) * F	A B + C D + E - * F *
A / (B + C)	A B C + /
A / B + C	A B / C +

Všimněme si předně, že v postfixovém zápisu (jakož i v prefixovém zápisu) se obejdeme bez závorek, pomocí kterých jsme v infixovém zápisu vyznačovali prioritu provádění operací. Dále si všimněme, že posloupnost (pořadí) operandů v jejich postfixovém zápisu je stejná jako v originálním infixovém zápisu (pokud např. operand X předchází

operand Y v infixovém zápisu, tak je tomu stejně i v postfixovém ekvivalentu),

posloupnost operátorů v postfixovém výrazu udává pořadí jejich aplikace.

Uvažujme nejprve např. infixový výraz A + B * C, jehož postfixový ekvivalent je A B C * +. První operand výrazu A + B * C, tj. A lze umístit bezprostředně do postfixového zápisu. Ovšem operátor + nelze umístit bezprostředně do postfixového výrazu, dříve je třeba vyhledat druhý operand. Umístěme proto zatím tento operátor do zásobníku operátorů. Jakmile je vyhledán druhý operand (B), tak B je umístěno do postfixového výrazu hned za A. Ovšem operátor + nelze umístit do postfixového výrazu za B, protože v infixovém zápisu výrazu je za B operátor **,** který má vyšší prioritu. Zaveďme tedy predikát *prty(op1,op2),* kde argumenty *op1* a *op2* jsou naše operátory. Tato funkce nechť vrátí hodnotu *true*, když operátor *op1* se musí aplikovat dříve než *op2,* tzn. že *op1* má buď vyšší prioritu, nebo při stejné prioritě je v infixovém zápisu vlevo od *op2.* Takže např*. prty('*','+')* , *prty('+','+')* vrací *true,* zatímco *prty('+','*')* vraci *false.*

Zvlášť je třeba zvážit případy, kdy infixový zápis výrazu obsahuje závorky. Každou otevírací závorku umístíme do zásobníku operátorů, což zajistíme konvencí *prty(op,'(')* = *false* pro operátor *op* jiný než uzavírací závorka. Když pak čteme uzavírající závorku, tak všechny operátory až k otevírací závorce musí být odebrány a umístěny do postfixového řetězce. To zajistíme konvencí *prty(op,')')* = *true* pro všechny operátory *op* jiné než *'('*. Když jsou odpovídající operátory odebrány ze zásobníku, je třeba ze zásobníku odebrat i otevírající závorku a eliminovat umístění obou závorek do výstupního řetězce. Uzavírací závorku do zásobníku neumisťujeme. Souhrně zapíšeme pravidla pro závorky následovně :

prty ('(', op) = *false* ,	pro každý operátor *op*,
prty (op, '(') = *false ,*	pro každý operátor *op* $\neq$ *')',*
prty (op, ')') = *true ,*	pro každý operátor *op* $\neq$ *'('* ,
prty (')', '(')	není definováno.

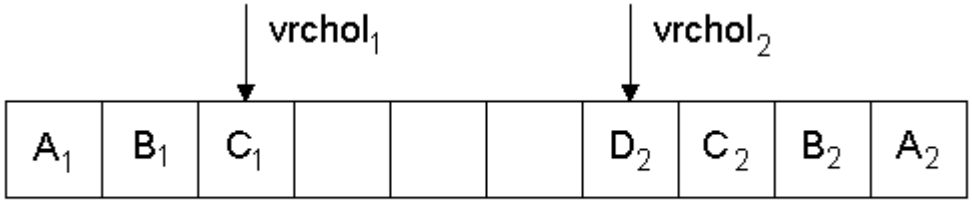
Samozřejmě na závěr zásobník operátorů vyprázdníme. Dále v případě, že přečteme první operátor zleva v infixovém zápisu, tak je zásobník operátorů prázdný a operaci *TOP (Z)* nelze provést. V takovém případě však může dále uvedená funkce se jménem *top* vrátit libovolný operátor, neboť v dále uvedeném zápisu nebude v případě, že zásobník operátorů je prázdný, vrácená hodnota validní. Řešení dané úlohy by mohlo vypadat následovně :

```
#define   MAX   100

int   sp  =  0 ;
char  hod[MAX] ;


void clear ( void )  {
        sp = 0 ;
}

int push ( char x )  {

        if ( sp < MAX )   {
                hod[sp++] = x ;  return ( 0 ) ;
        }
        clear ( ) ;  return ( 1 ) ;
}


char top ( void )  {

        if ( sp ) return ( hod[sp-1] ) ;
        return ( '+' ) ;
}


void   pop  ( void )   {

        if ( sp ) --sp ;
}

int prty ( char op1, char op2 ) {
        int x;

        switch ( op1 ) {
                case '+' : case '-' :
                        x = ( op2 == '+' || op2 == '-' || op2 == ')' ) ? 1 : 0;
                        break ;
                case '*' : case '/' : x = ( op2 == '(' ) ? 0 : 1; break;
                default  : x = 0; break;
        }
        return ( x ) ;
}
```

```
int intopost ( char *infix, char *postfix ) {

        char vrchol ;

        clear () ;
        while ( *infix != '\0' )  {
                if ( *infix > 64 && *infix < 91  ) /* jde o operand */
                        *postfix++ = *infix ;
                else  {
                        while ( sp && prty ( vrchol = top(), *infix ) ) {
                                pop() ; *postfix++ = vrchol ;
                        }
                        if ( !sp || *infix != ')' )  {
                                if ( push ( *infix ) )  return ( 1 ) ;
                        }
                        else    pop () ;
                }
                infix++ ;
        }
        while ( sp )  {
                *postfix++ = top() ; pop () ;
        }
        *postfix = '\0' ;
        return ( 0 ) ;
}
```

2.4 Cvičení

1. Implementujte dva zásobníky pomocí jednoho společného pole tak, aby k případnému přeplnění zásobníku docházelo až po úplném vyčerpání přiděleného paměťového místa a aby nebylo nutné zásobníky v poli přesouvat. Napište svoje funkce pro realizaci základních operací s takovými zásobníky. Návod viz obr. 5.



obr. 5

2. Modifikujte v textu uvedené funkce pro konversi infixového zápisu výrazu na postfixový zápis výrazu pro případ, že vstupní infixový řetězec nemusí obsahovat správný zápis výrazu, tzn. ošetřete případné chyby v infixovém zápisu výrazu.

3. Užitím zásobníku řešte konversi vstupního infixového řetězce na výstupní prefixový řetězec.

   Návod : Použijte řešení konverse infixového na postfixový zápis výrazu s tím, že vstupní infixový řetězec čtete zprava (stejně modifikujte i priority).

4. Uvažujte fiktivní počítač s jedním registrem a možností interpretovat následující instrukce :

LD	*A*	umístění operandu *A* do registru,
ST	*A*	umístění obsahu registru do proměnné A,
AD	*A*	přičtení hodnoty proměnné *A* k obsahu registru,
SB	*A*	odečtení hodnoty proměnné *A* od obsahu registru,
ML	*A*	vynásobení obsahu registru hodnotou proměnné A,
DV	*A*	dělení obsahu registru hodnotou proměnné *A*.

   Napište svoje funkce, které pro daný postfixový řetězec (operandy jsou písmena, operátory jsou binární aritmetické operátory, zápis bez mezer) zapíše do nějakého výstupního souboru posloupnost instrukcí pro výpočet daného postfixového výrazu. Pro dočasné proměnné použijte *TEMPxx*, kde x jsou číslice *0* až *9*. Tak např. pro postfixový výraz *A B C * + D E - /* bude mít posloupnost instrukcí tvar :

   ```
   LD   B
   ML   C
   ST   TEMP01
   LD   A
   AD   TEMP01
   ST   TEMP02
   LD   D
   SB   E
   ST   TEMP03
   LD   TEMP02
   DV   TEMP03
   ST   TEMP04
   ```

Návod :

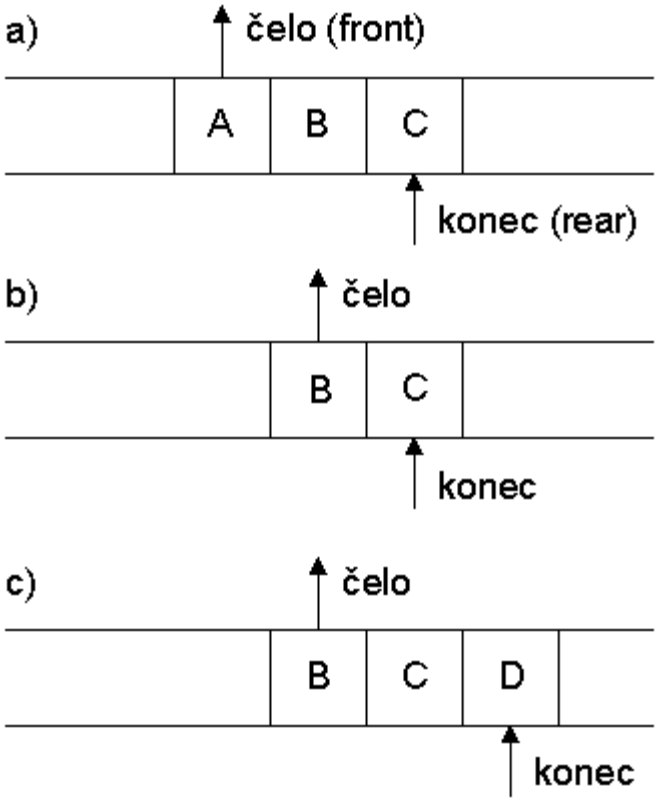
- Pro vyhodnocování postfixového řetězce použijte zásobník operandů.
- Připravte si funkci, které předáte operátor a dva operandy a která do výstupního souboru zapíše odpovídající posloupnost instrukcí.

5. Modifikujte řešení předchozího příkladu pro případ, že operandy ve vstupním postfixovém zápisu jsou identifikátory až s 8 signifikantními znaky. Současně optimalizujte počet dočasných proměnných.

3. FRONTA

3.1 Specifikace fronty

Fronta (angl. queue) je posloupnost prvků stejného typu, která se rozšiřuje přidáním prvku na konec fronty (angl. rear) a zmenšuje odebráním prvku z čela fronty (angl. front). Obr. 6a ilustruje frontu obsahující 3 prvky A, B, C. Situace po odebrání prvku z čela fronty zachycuje obr. 6b a konečně na obr. 6c je znázorněn stav po přidání prvku D na konec fronty. Taková posloupnost je tedy přístupná na obou koncích této posloupnosti.



obr. 6

Algebraická specifikace fronty podle [2] je uvedena v příloze. Nechť $< f_1\ f_2\ \ldots\ldots\ f_n >$ je posloupnost prvků fronty *F*, <> je prázdná posloupnost. Sémantiku operací charakterizujících frontu lze popsat následujícími rovnostmi.

NEW = <> ,
EMPTY? (<>) = *true* ,
INSERT (f_{n+1}, $< f_1\ f_2\ \ldots\ldots f_n >$) = $< f_1\ f_2\ \ldots\ldots f_n\ f_{n+1} >$,

a pro $n \geq 1$

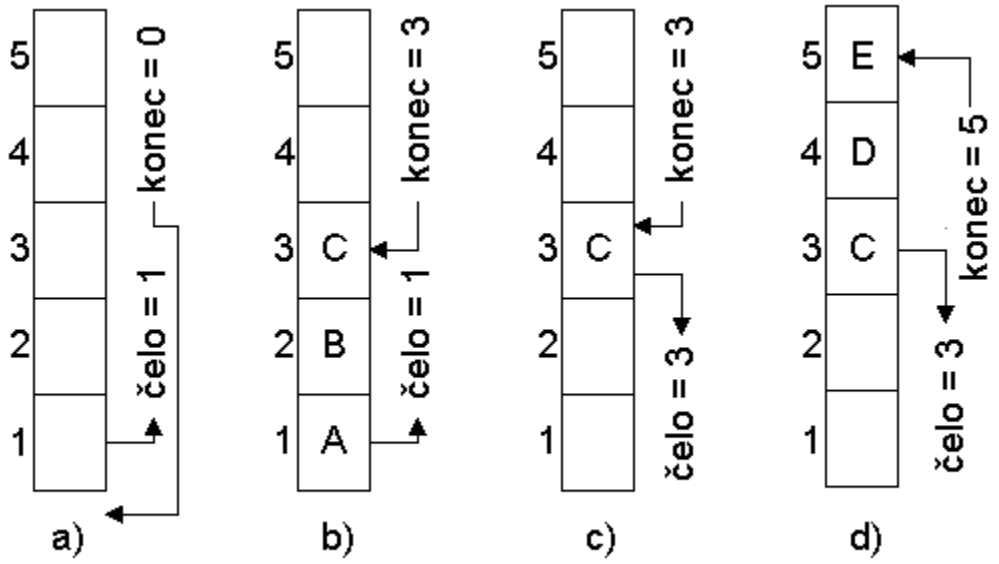
REMOVE ($< f_1\ f_2\ \ldots\ldots f_n >$) = $< f_2\ f_3\ \ldots\ldots f_n >$,
FRONT ($< f_1\ f_2\ \ldots\ldots f_n >$) = f_1 ,

EMPTY? (< f_1 f_2 …….f_n >) = false .

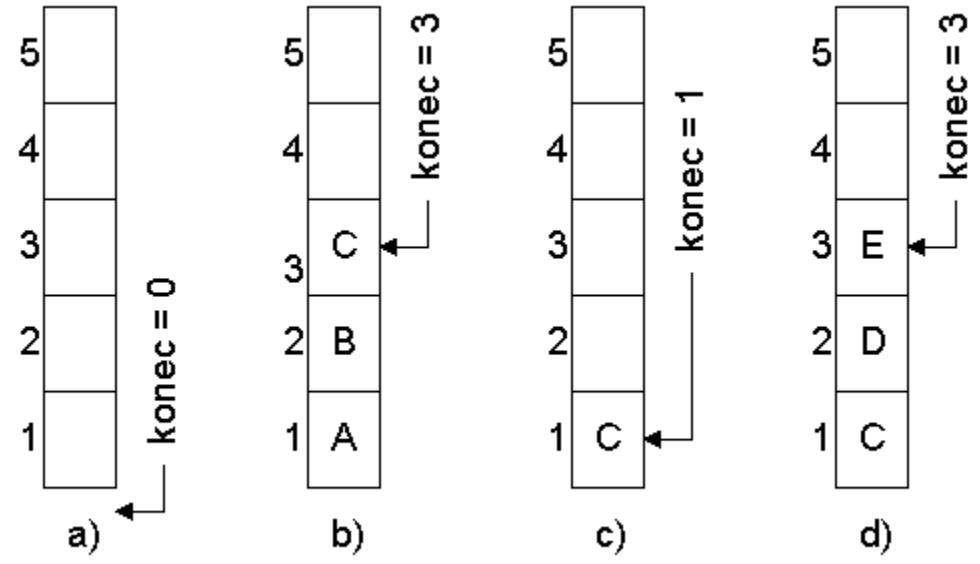
Poznamenejme ještě, že fronta se též nazývá **pamětí FIFO** (z anglického first-in, first-out).

3.2 Implementace fronty

Frontu lze implementovat pomocí pole, kam se budou ukládat prvky fronty a dále užitím dvou proměnných, které budou indikovat polohu čela a konce fronty (viz obr. 7).



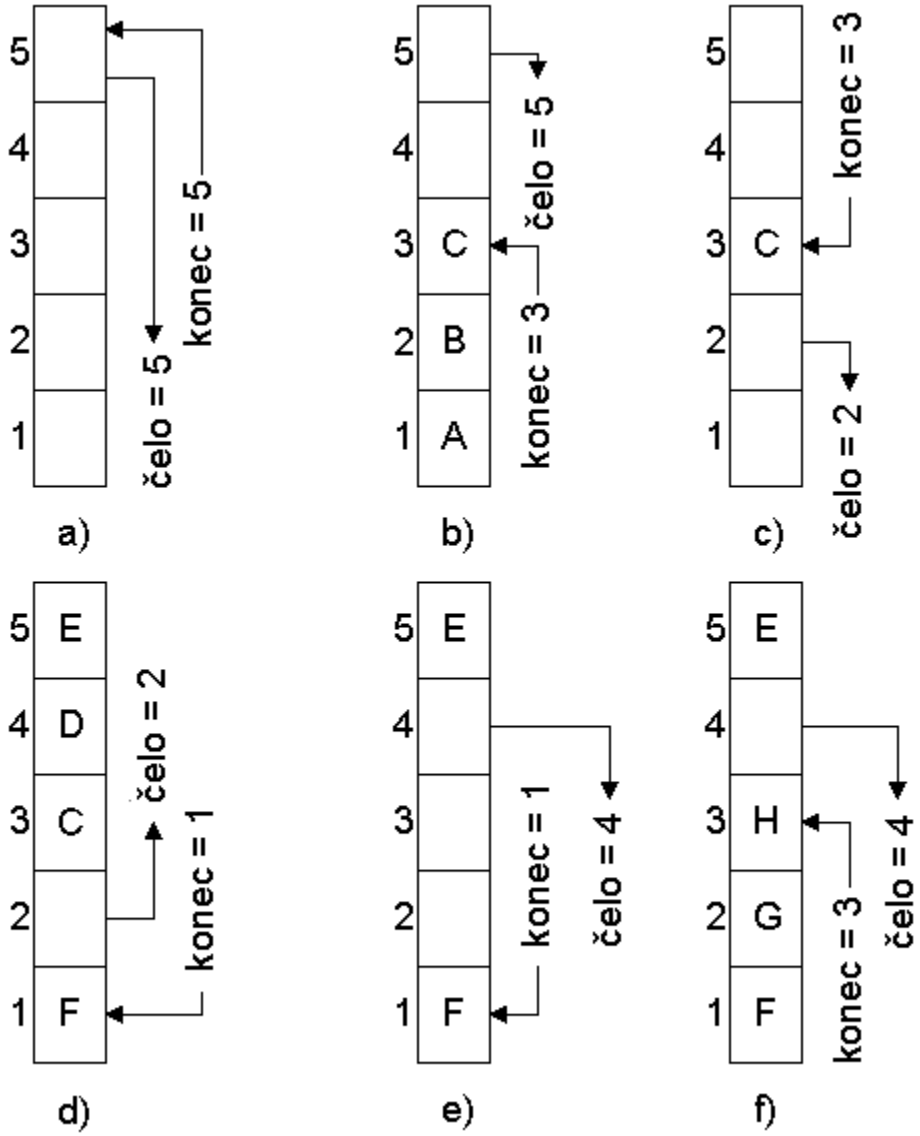
obr. 7



obr. 8

Na počátku provedeme inicializaci fronty pomocí *čelo = 1 , konec = 0* (viz obr. 7a). Pak postupně přidáme prvky A, B a C (obr. 7b),

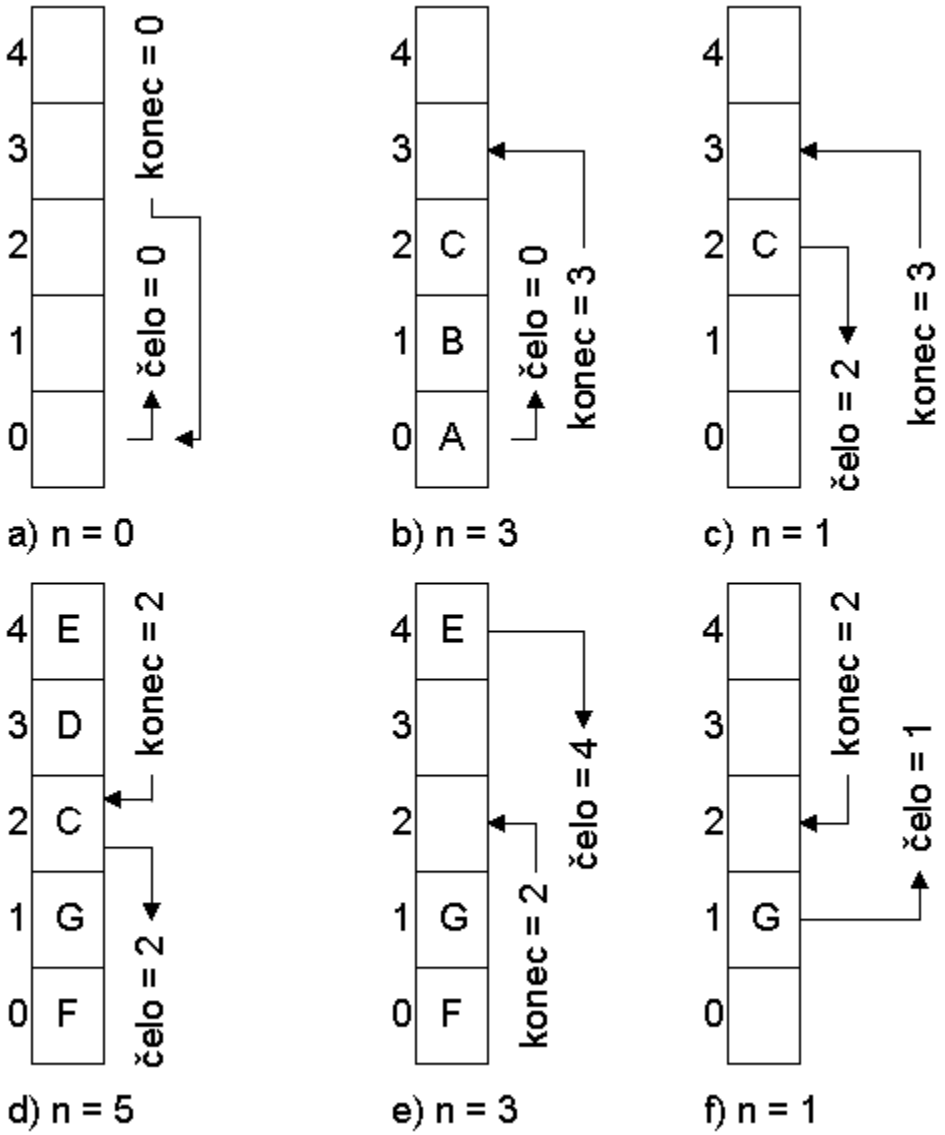
odebereme A a B (obr. 7c) a přidáme D a E (obr. 7d). Při operaci přidání prvku do fronty inkrementujeme proměnnou *konec* a na odpovídající složku pole zapíšeme nový prvek, při odebrání prvku z čela fronty odkládáme hodnotu složky pole s indexem *čelo* a pak tuto proměnnou inkrementujeme. Aktuální počet prvků ve frontě je *konec-čelo+1* a fronta je prázdná, když *konec* $<$ *čelo*. Všimněme si, že v situaci podle obr. 7d již frontu nelze rozšířit, ačkoliv první dvě složky pole jsou volné.



obr. 9

Modifikujme proto řešení našeho příkladu tak, aby čelo fronty bylo stále na první složce pole (viz obr. 8). Pokud je ovšem prvků ve frontě mnoho, tak jejich přesouvání při každém zmenšení fronty by spotřebovalo příliš mnoho času a taková modifikace by byla neefektivní (případně lze tzv. “setřásání“ prvků v poli provádět pouze ve vhodných časových okamžicích). Mnohem výhodnější je tzv. cirkulace prvků fronty

v poli dle příkladu na obr. 9. Inicializaci fronty znázorňuje obr. 9a. V tomto případě však splnění podmínky *čelo* = *konec* signalizuje prázdnou frontu, a proto také v situaci podle obr. 9d nebo 9f nelze již frontu rozšiřovat. Má-li tedy pole *m* složek, může se ve frontě nacházet maximálně *m-1* prvků ! Čtenář jistě sám zapíše procedurální vyjádření základních operací s frontou v jazyku Pascal, při zápisu odpovídajících funkcí v jazyku C musí uvážit, že implicitně dolní index pole je 0 (na předchozích obrázcích jsme složky pole číslovali od 1).



obr. 10

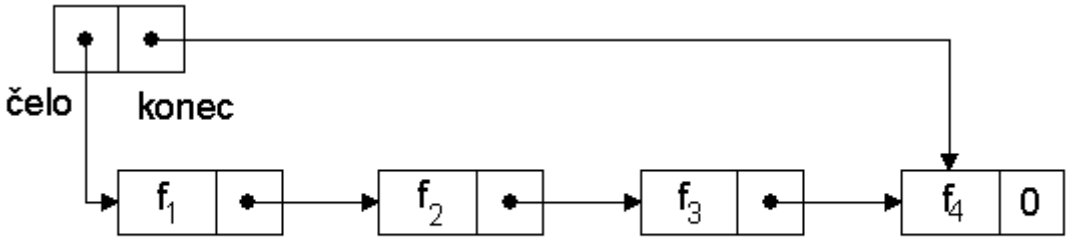
Jiný způsob implementace fronty pomocí pole je na obr. 10. Kromě proměnných (položek záznamu, nebo členů struktur) *čelo* a *konec* zavedeme další proměnnou *n,* jejíž obsah bude udávat okamžitou délku

fronty. Pokud složky pole na obr. 10 číslujeme od 0, tak inkrementaci proměnných *čelo* a *konec* zapíšeme ve tvaru

(čelo+1) mod *m* , *(konec+1)* mod *m* ,

a pak maximální délka fronty je rovna délce deklarovaného pole (tedy *m*). Na druhé straně však musíme navíc deklarovat nějakou celočíselnou proměnnou *n*. Zápis odpovídajících zdrojových textů v symbolickém jazyku je ponechán také na čtenáři.

Při použití spojového seznamu je výhodné spravovat kromě ukazatele na čelo fronty i ukazatel na konec fronty (viz obr. 11). Vyhneme se tím ev. procházení fronty počínaje prvkem na čele, za účelem nalezení ukazatele na konec fronty. Jedna možná implementace fronty by mohla vypadat následovně (TYPPRVKU je nějaký předem definovaný typ) :



obr. 11

```
typedef struct prvek {
        TYPPRVKU     hod ;
        struct prvek    *spoj ;
} PRVEK ;

typedef struct  {
        PRVEK  *celo ;
        PRVEK  *konec ;
} FRONTA ;

FRONTA *init ( void )  {
        FRONTA  *f ;

        f = (FRONTA *) malloc ( sizeof ( FRONTA ) ) ;
        f->celo = NULL ; return ( f ) ;
}

int empty ( FRONTA *f )  {

        if ( f->celo )        return ( 0 ) ;
```

```
        else                return ( 1 ) ;
}

void insert ( FRONTA *f, TYPPRVKU x ) {
        PRVEK  *p ;

        p = (PRVEK *) malloc ( sizeof ( PRVEK ) ) ;
        p->hod = x ; p->spoj = NULL ;
        if ( empty ( f ) )  f->celo = p ;
        else                f->konec->spoj = p ;
        f->konec = p ;
}

TYPPRVKU remove_f ( FRONTA *f ) {
        PRVEK  *p ;
        TYPPRVKU  x ;

        if ( empty ( f ) )
                /* Korekční akce uživatele */
        else  {
                x = f->celo->hod ; p = f->celo ;
                f->celo = f->celo->spoj ; free ( p ) ;
                return ( x ) ;
        }
}

void clear ( FRONTA *f ) {
        PRVEK *p ;

        while ( !empty ( f ) ) {
                p = f->celo ; f->celo = f->celo->spoj ; free ( p ) ;
        }
        free ( f ) ;
}
```

3.3 Použití fronty

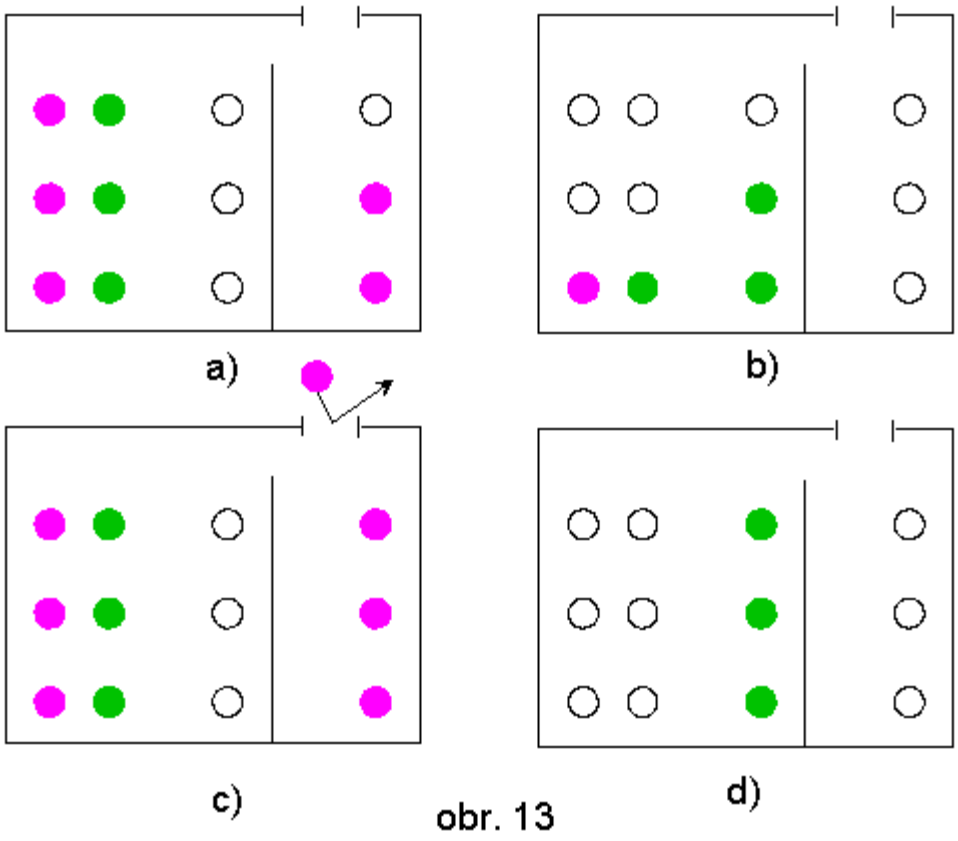
Z hlediska používání je fronta relativně frekventovanou datovou strukturou, kterou používají různé aplikační i systémové programy. Např. z architektury X Window je znám program nazývaný X server, jehož zjednodušená struktura je na obr. 12. Na straně výstupu přijímá X server požadavky X klientů na zobrazení, které řadí do front požadavků. Na

straně vstupu naopak X server řadí vstupní události generované uživatelem do front vstupních událostí.

obr. 12

Jiný prozaičtější příklad je uveden v [2]. Pomocí fronty simulujme řízení obsluhy v holičství, kde zákazníci by měli být obsluhováni v takovém pořadí, v jakém do holičství přišli. Je-li však více holičů volných, tak zákazníka obslouží ten holič, který byl nejdéle bez práce. Obsluha bude řízena pomocí dvou front : do fronty *FH* se budou v případě, že čekárna je prázdná , ukládat identifikační znaky volných holičů a do fronty *FZ* se budou ukládat např. pořadová čísla zákazníků, kteří čekají na obsluhu. Pořadové číslo zákazníka můžeme počítat od počátku pracovního dne. Dále se můžeme domluvit na tom, že v holičství pracují max. tři holiči současně a že ve frontě zákazníků budou také max. tři zákazníci, kteří čekají na obsluhu. To znamená, že když např. přicházející zákazník vidí v čekárně tři jiné zákazníky čekající na obsluhu, tak z holičství odejde (třeba již proto, že si nemá kam sednout).

Ilustrace k tomuto příkladu je na obr. 13. Na obr. 13a čekají dva zákazníci (●) ve frontě *FZ*, naopak na obr. 13b čekají dva holiči (●) ve frontě *FH.* Obr. 13c ilustruje situaci, kdy je plná fronta *FZ* a přichází další zákazník. Obr. 13d ilustruje stav např. na začátku pracovního dne. Zpracování programu simulujícího řízení obsluhy v holičství je ponecháno na cvičení.



obr. 13

3.4 Cvičení

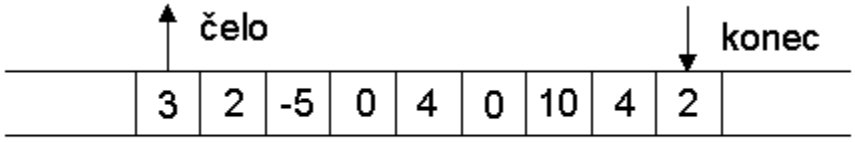
1. Implementujte frontu pomocí pole s cirkulací prvků fronty v poli.

2. Alternativně k variantám uvedeným v textu lze frontu implementovat pomocí pole bez cirkulace prvků v poli i tak, že posouvání prvků fronty se provádí jen v případě, když konec fronty koresponduje s poslední složkou pole. Zapište pro tento případ svoje funkce, realizující základní operace s frontou.

3. Definujte novou datovou strukturu jako posloupnost prvků stejného typu, přičemž tato posloupnost je přístupná jak při přidávání, tak i při odebírání prvku na obou koncích posloupnosti. Tzn. že nový prvek lze přidat buď na levý konec posloupnosti (operace *INSERTLEFT*), nebo

na pravý konec posloupnosti (operace *INSERTRIGHT*). Analogicky při odebírání prvku to budou operace *REMOVELEFT* a *REMOVERIGHT*. Zapište svoje funkce pro vyjádření zmíněných operací s takovou datovou strukturou tak, aby byly funkční i v případě, že posloupnost prvků je prázdná a byly též indikovány případy podtečení a přetečení při implementaci pomocí pole. Ukažte, jak lze tuto datovou strukturu použít pro reprezentaci jak fronty, tak i zásobníku. Oba konce posloupnosti si může čtenář označit i jinak.

4. Implementujte pomocí pole frontu jako posloupnost prvků, které obsahují proměnný počet celých čísel. Nechť např. ve frontě jsou celá čísla 2, -5, 0 a druhý prvek jsou 0, 10, 4, 2. Pak můžeme frontu schematicky znázornit podle obr. 14.



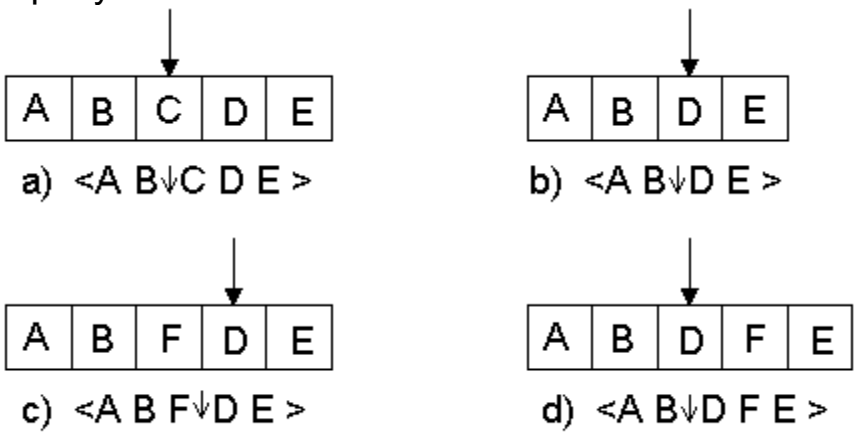
obr. 14

5. Napište svoje funkce pro simulaci řízení obsluhy v holičství podle návodu v textu.

4. SEZNAM

4.1 Specifikace seznamu

Až dosud jsme se zabývali posloupnostmi prvků, u kterých bylo možné aplikovat operace přidání a odebrání prvku pouze na konci nebo koncích této posloupnosti. Seznamem (angl. list) nazýváme posloupnost prvků, u které lze takové operace aplikovat na libovolném místě této posloupnosti. Z tohoto hlediska lze na zásobník i frontu nahlížet jako na speciální případy seznamu.



obr. 15

Seznam budeme interpretovat jako posloupnost prvků s vnitřním ukazatelem, který označuje místo (tj. prvek) aplikace operace. Toto místo budeme v posloupnosti prvků označovat šipkou ↓ (viz obr. 15a). Odebereme-li ze seznamu podle obr. 15a prvek, dostaneme seznam podle obr. 15b. Poloha ukazatele v tomto novém seznamu nechť je zatím naší interní dohodou. Přidáme-li k seznamu podle obr. 15b prvek, mohou nastat dvě situace (viz obr. 15c a 15d), a proto je nutné předem specifikovat, zda prvek se má přidat před, nebo za označený prvek.

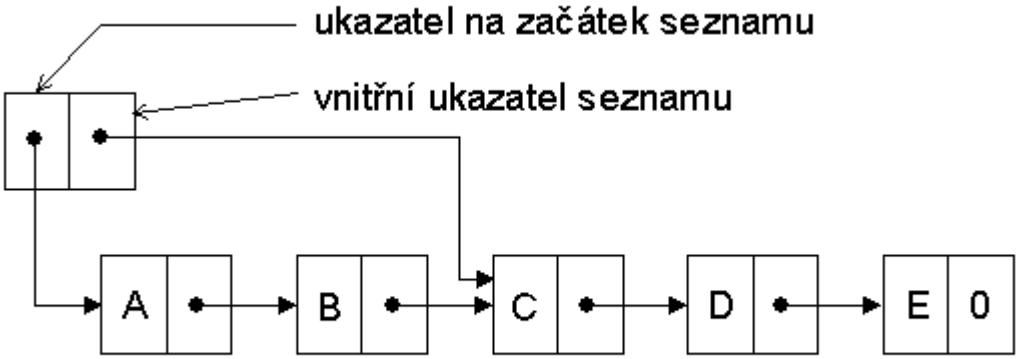
Algebraická specifikace seznamu podle [2] je uvedena v příloze, operace CONS je přidání prvku jako prvního prvku seznamu. Specifikaci seznamu lze provést výčtem následujících elementárních operací, pomocí nichž lze konstruovat složitější operace se seznamy :

1. inicializace seznamu, tj. vytvoření prázdného seznamu, ve kterém přítomnost vnitřního ukazatele vyjádříme zápisem < ↓ > :
   NEW = < ↓ > ,
2. nastavení vnitřního ukazatele na počátek seznamu :
   FIRST (< s_1 s_2 …..↓s_i ……. s_n >) = < ↓s_1 s_2 ……. s_n > ,
3. přidání nového prvku před označený prvek :
   INSERTP (s_{n+1}, < s_1 s_2 ….↓s_i …. s_n >) = < s_1 s_2 …. s_{n+1}↓s_i …. s_n >,
4. přidání nového prvku za označený prvek :
   INSERTN (s_{n+1}, < s_1 s_2 ….↓s_i …. s_n >) = < s_1 s_2 …. ↓s_i s_{n+1}…. s_n >,

5. odebrání prvku ze seznamu :
   $DELETE\ (< s_1\ s_2\ \ldots.\ s_{i-1} \downarrow s_i\ s_{i+1}\ \ldots.\ s_n >) = < s_1\ s_2\ \ldots. s_{i-1} \downarrow s_{i+1}\ \ldots.\ s_n >$,
6. čtení prvku ze seznamu :
   $GETELEM\ (< s_1\ s_2\ \ldots.. \downarrow s_i\ \ldots\ldots.\ s_n >) = s_i$,
7. posun vnitřního ukazatele o jeden prvek vpravo :
   $NEXT\ (< s_1\ s_2\ \ldots.\ \downarrow s_i\ s_{i+1}\ \ldots.\ s_n >) = < s_1\ s_2\ \ldots. s_i \downarrow s_{i+1}\ \ldots.\ s_n >$,
8. rozlišení prázdného a neprázdného seznamu :
   $EMPTY?\ (< \downarrow >) = true$,
   $EMPTY?\ (< s_1\ s_2\ \ldots.. \downarrow s_i\ \ldots\ldots.\ s_n >) = false$, $n \geq 1$,
9. test, zda vnitřní ukazatel ukazuje za poslední prvek seznamu :
   $NULL?\ (< s_1\ s_2\ \ldots..\ldots\ldots.\ s_n \downarrow >) = true$,
   $NULL?\ (< s_1\ s_2\ \ldots.. \downarrow s_i\ \ldots\ldots.\ s_n >) = false$, $n \geq 1$, $i \leq n$.

4.2 Implementace seznamu

Seznamy se implementují převážně jako dynamicky generovaná data (viz grafické znázornění jednoho příkladu na obr. 16). Budou spravovány dva ukazatele – na první prvek seznamu (tj. na začátek seznamu) a vnitřní ukazatel seznamu (tj. na označený prvek seznamu).

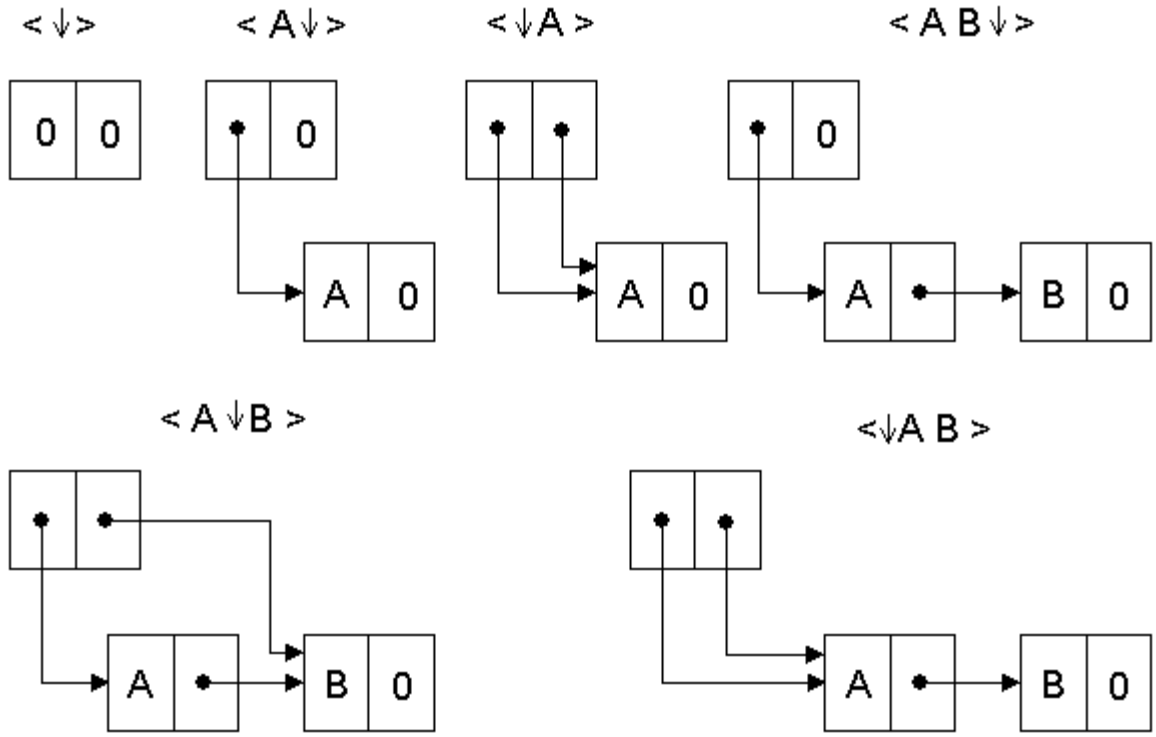


obr. 16

Dále by mohlo být věcí programátora samotného, ať zpracovává seznam podle svých potřeb, tzn. v určitém kontextu. Nakonec ani v daném kontextu nemusí být nezbytně nutné zavést vnitřní ukazatel. Pak ovšem musí programátor definovat žádané operace se seznamem tak, aby výsledný program pracoval efektivně.

V dalším textu jsou uvedeny jen některé obtížnější implementace dříve uvedených elementárních operací se seznamem. Použití takových funkcí (nebo procedur v Pascalu) umožní snadnější a rychlejší ladění programu, ovšem na druhé straně jejich použití může být v daném kontextu neobratné, nebo výsledný program nemusí být právě efektivní. Záleží opět na programátorovi a charakteru konkrétní úlohy, zda si pro

svoji potřebu uvedenou implementaci upraví, nebo použije svůj vlastní mechanismus operací se seznamem. Pro snadnější pochopení dále uváděných funkcí jsou na obr. 17 ilustrovány příklady velmi jednoduchých seznamů.



obr. 17

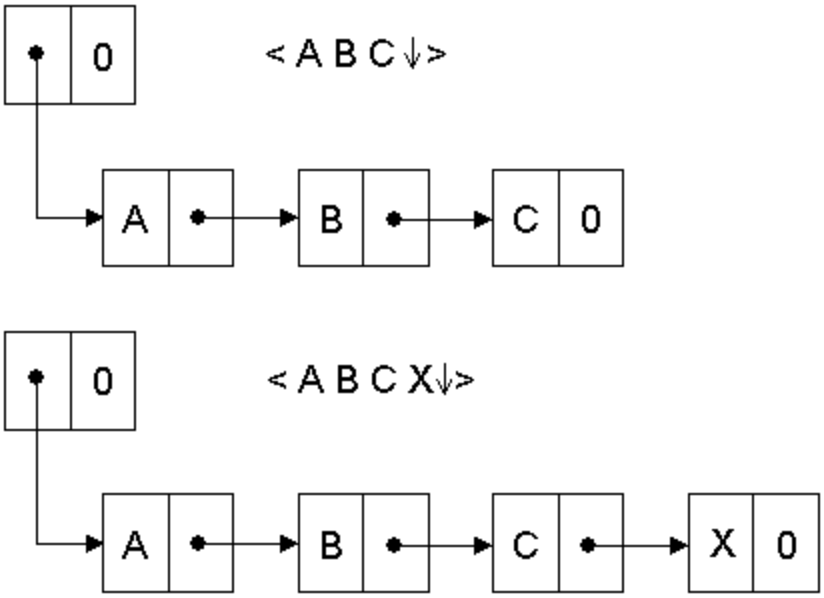
Jako jednu z možností implementace seznamu uvažujme následující typy :

```
typedef struct prvek {
            TYPPRVKU     hod ;
            struct prvek    *spoj ;
} PRVEK ;

typedef struct  {
      PRVEK   *zac ;
      PRVEK   *vu ;
} SEZNAM ;
```

Pak např. operaci *NEW* (tj. vytvoření prázdného seznamu) lze realizovat pomocí této funkce :

```
SEZNAM *init ( void )  {
```

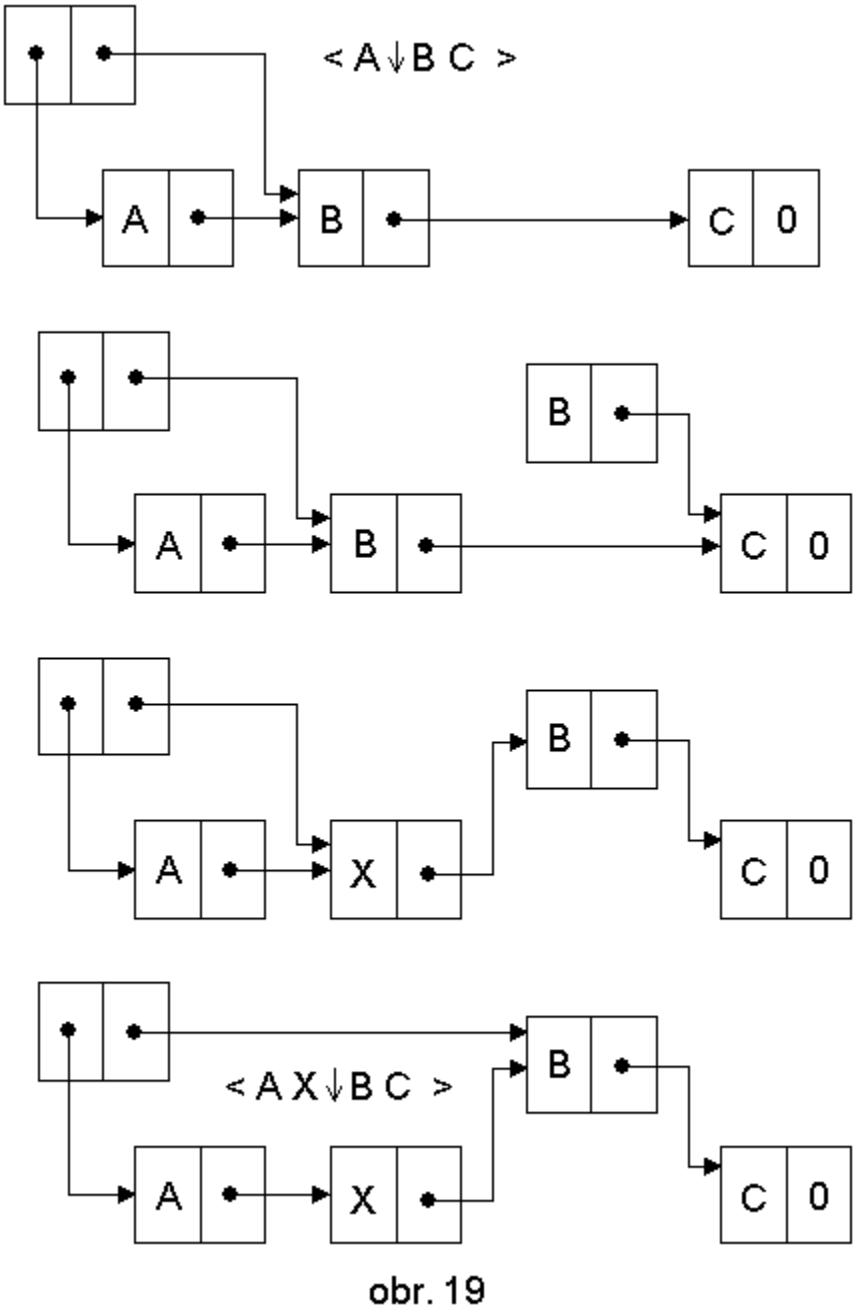
```
        SEZNAM *s = (SEZNAM *) malloc ( sizeof ( SEZNAM ) ) ;

        s->zac = s->vu = NULL ;
        return ( s ) ;
}
```



obr. 18

Rozeberme dále operaci *INSERTP.* Při přidání nového prvku před označený prvek mohou nastat tyto případy. Když je třeba přidat nový prvek na konec neprázdného seznamu (viz obr. 18), tak potřebujeme znát hodnotu ukazatele na poslední prvek seznamu. Tu ovšem v našem případě můžeme určit pouze prohledáním seznamu od začátku. Pokud by se takový případ měl vyskytovat s velkou frekvencí, tak by bylo zřejmě výhodnější spravovat i ukazatel na poslední prvek seznamu. Adekvátně tomu by bylo třeba modifikovat i uvedenou implementaci seznamu (to však je ponecháno na čtenáři samotném). Při přidání nového prvku před označený prvek "někam doprostřed" (ne na konec seznamu) budeme postupovat podle schématu na obr. 19. Do nově vygenerovaného prvku se přesune obsah označeného prvku, kam se poté uloží hodnota nového prvku s ukazatelem na nový prvek. Pak se modifikuje vnitřní ukazatel. Odpovídající funkce by mohla vypadat následovně :

```
void insertp ( SEZNAM  *s, TYPPRVKU  x ) {

        PRVEK  *p ;
        PRVEK  *q = (PRVEK *) malloc ( sizeof ( PRVEK ) ) ;
```

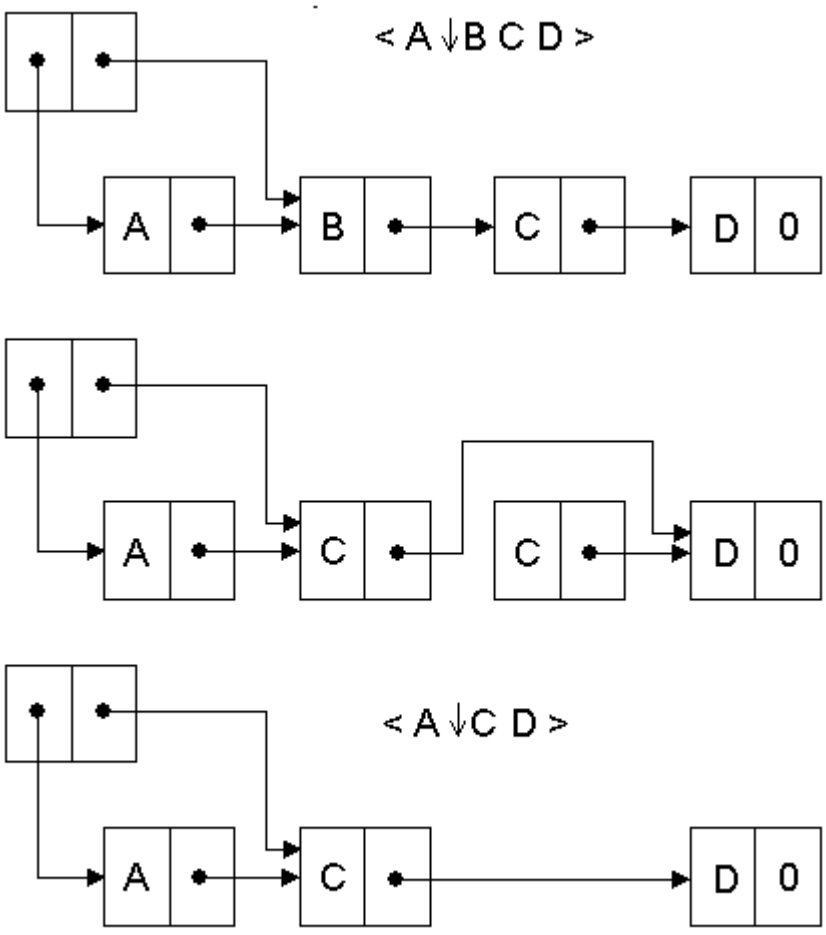


obr. 19

```
if ( ! s->vu  ) {            /* Nový prvek na konec seznamu */
        q->hod = x ; q->spoj = NULL ;
        if ( ! s->zac )  s->zac = q ;
        else {
                p = s->zac ;
                while ( p->spoj ) p = p->spoj ;
                p->spoj = q ;
        }
}
else  {                      /* Nový prvek "doprostřed" */
         *q = *s->vu ;
```

```
                s->vu->hod = x ; s->vu->spoj = q ; s->vu = q ;
        }
}
```

Na obr. 20 je ilustrován v jednotlivých krocích odběr jiného než posledního prvku seznamu. Zvlášť je pak nutné řešit případ odběru posledního prvku seznamu (viz funkce *delete_s*).



obr. 20

```
void delete_s( SEZNAM *s ) {

        PRVEK *p, *q ;

        if ( !s->vu )
                printf ( "\nZe seznamu nelze nic odebrat\n" ) ;
        else {
                if ( !(*s->vu).spoj ) { /* Odběr posledního prvku */
                        if ( s->zac == s->vu ) s->zac = NULL ;
                        else   {
```

```
                        p = s->zac ;
                        while ( p->spoj != s->vu ) p = p->spoj ;
                        p->spoj = NULL ;
                    }
                    q = s->vu ; s->vu = NULL ;
                }
                else  {
                    q = s->vu->spoj ; *s->vu = *s->vu->spoj ;
                }
                free ( q ) ;
        }
}
```

Realizace ostatních operací se seznamem je jednoduchá a je ponechána na čtenáři. Zvlášť je však třeba ošetřit případy, kdy danou operaci nelze provést (např. vložit nový prvek za označený v případě, kdy vnitřní ukazatel je nastaven za poslední prvek seznamu). Užitečná může být i následující doplňující funkce :

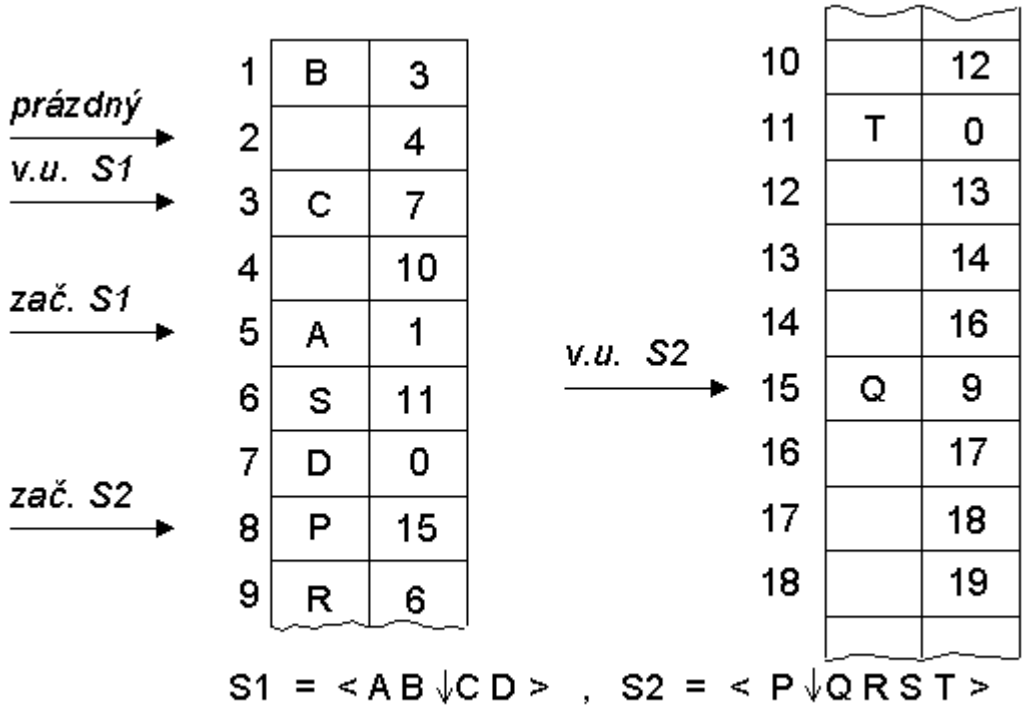
```
void clear ( SEZNAM *s ) {

    first ( s ) ;
    while ( empty ( s ) != NULL ) delete_s ( s ) ;
    free ( s ) ;
}
```

Pomocí uvedených elementárních operací lze relativně snadno řešit i komplexnější úlohy. Např. následující funkce vytvoří v paměti k originálnímu seznamu jeho kopii :

```
SEZNAM *copy_s ( SEZNAM *s1 )  {

    SEZNAM *s2 ;

    s2 = init () ; first ( s1 ) ;
    while ( !null ( s1 ) )  {
        insertp ( s2, getelem ( s1 ) ) ; next ( s1 ) ;
    }
    return ( s2 ) ;
}
```

Zápis takových funkcí pak vypadá sice velmi jednoduše, ovšem nemusí být právě efektivní (viz náš zápis operace *INSERTP*). Případné úpravy jsou ponechány na čtenáři.

Uvedené seznamy lze implementovat i pomocí pole, i když z čistě formálního hlediska programátorské etiky je lepší dříve uvedená implementace (viz příklad na obr.21).

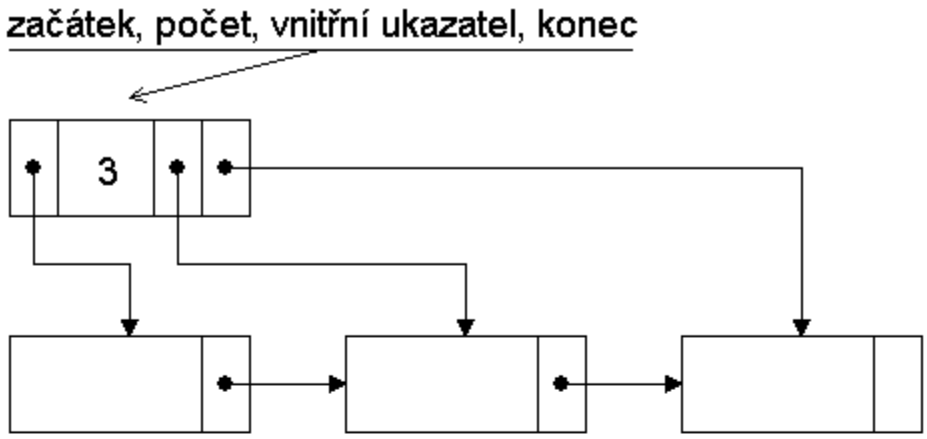


obr. 21

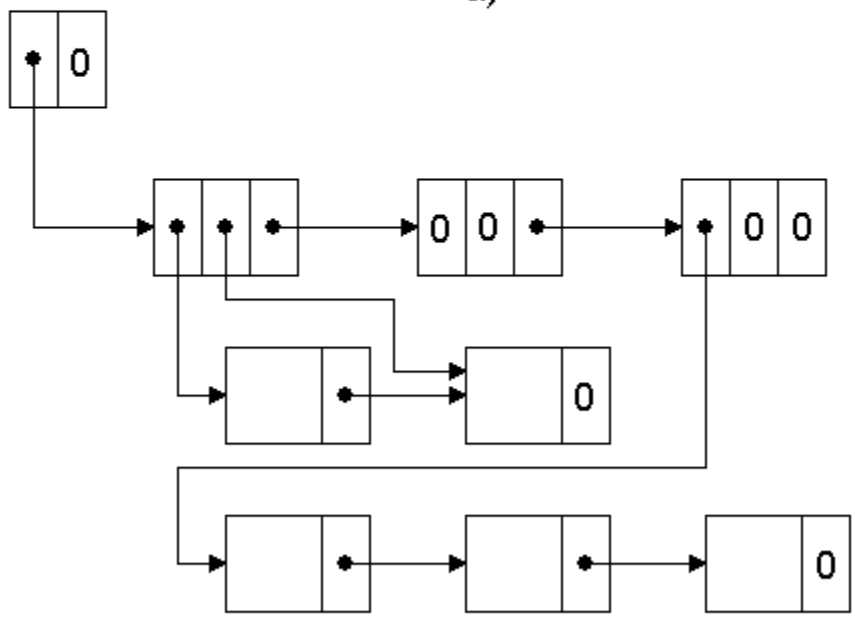
Čísla ve druhém sloupci jako indexy složek pole fungují jako ukazatelé na další prvek seznamu, prázdný ukazatel je reprezentován hodnotou 0. Všimněme si na obr.21 též toho, že kromě seznamů *S1* a *S2* je v poli jakýsi *prázdný* seznam. Prvky tohoto seznamu jsou ty složky pole, které jsou k dispozici pro ev. rozšiřování seznamů. Naopak odebrání nějakého prvku ze seznamu uvolní odpovídající složku pole pro ev. další použití, tzn. vrácení této složky do prázdného seznamu. Na počátku (ještě než začneme nějaký seznam vytvářet) jsou všechny složky pole nevyužité a je tedy třeba inicializovat prázdný seznam. Dále pak ve stejném smyslu a za stejným účelem, jak byly dříve použity standardní funkce pro přidělování a uvolňování paměti, použijeme při implementaci seznamů pomocí pole funkce pro operace s prázdným seznamem – funkci, která odebere prvek z prázdného seznamu a vrátí ukazatel na tento prvek a funkci, která vrací do prázdného seznamu složku pole referencovanou hodnotou argumentu. Zápisy odpovídajících funkcí jsou ponechány na čtenáři.

4.3 Modifikace a použití seznamů

Velmi často bývá užitečné vložit do seznamu jako první prvek tzv. **záhlaví seznamu**, které do vlastního seznamu nepatří, ale nese nějakou globální informaci o seznamu. Tak např. do seznamu celých čísel lze vložit záhlaví, kde uložené celé číslo se rovná aktuálnímu počtu prvků v seznamu. Pak ovšem můžeme snadno určit počet prvků v seznamu i bez toho, že bychom procházeli celým seznamem. Často však bývá rozumnější považovat za záhlaví každou vzhledem k seznamu externí proměnnou, která kromě ukazatele na začátek seznamu a vnitřního ukazatele obsahuje další údaje (např. počet prvků seznamu a ukazatel na poslední prvek seznamu – viz příklad na obr. 22a).



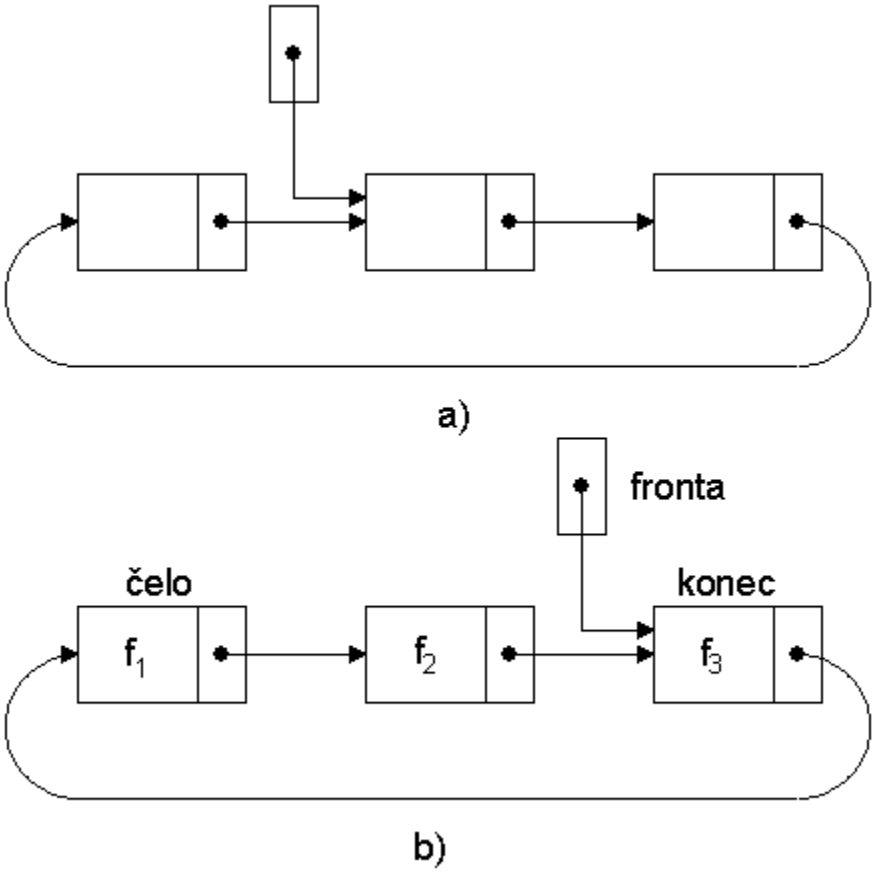
a)



b)

obr. 22

Není důvodu, proč by taková záhlaví seznamů nemohla být prvky nějakého jiného seznamu. Dostáváme tak složitější datové struktury (viz obr. 22b), které se často nazývají jako **multispojové struktury**. Pro takové složitější datové struktury je třeba modifikovat vyjádření operací se seznamem.

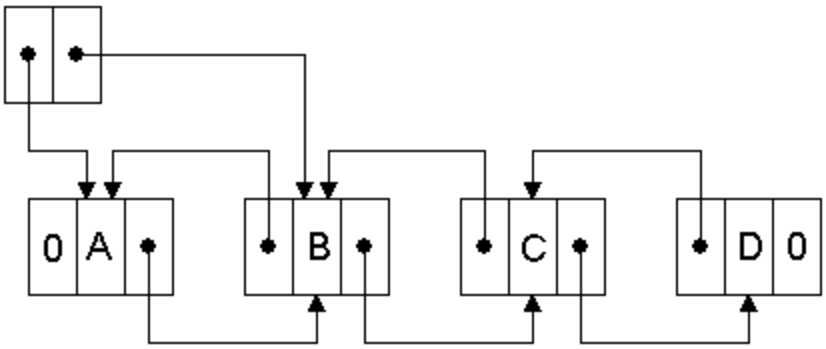


obr. 23

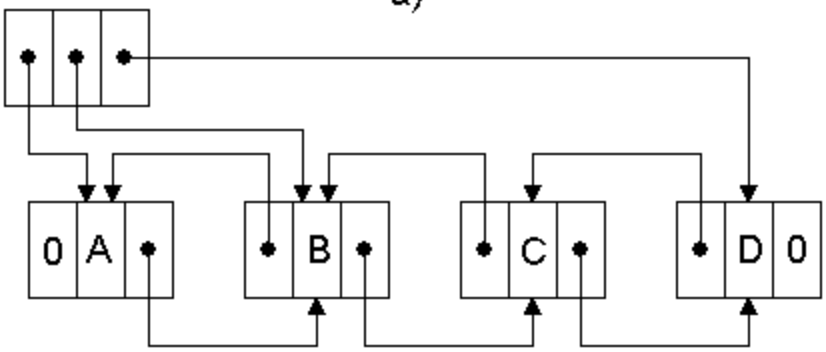
Zvláštní třídou mezi tzv. jednosměrně propojenými seznamy jsou **kruhové seznamy** (angl. circular lists), jejichž principiální schéma je na obr. 23a. Pak lze stěží mluvit o nějakém prvním, nebo posledním prvku seznamu a při praktických aplikacích kruhového seznamu zpravidla vystačíme s jednou, vzhledem k seznamu externí proměnnou, jejíž hodnotou je ukazatel na nějaký prvek seznamu. Jiné by to ovšem bylo v případě, kdy na principu takové datové struktury implementujeme frontu (tam je zřejmě užitečné znát dva ukazatele – na čelo a konec fronty). V případě implementace fronty pomocí kruhového seznamu nám stačí znát ukazatel na konec fronty, neboť pak je nám bezprostředně přístupné i čelo fronty (viz obr. 23b).

Až dosud jsme se zabývali seznamy s tzv. jednosměrným propojením. V některých případech může být výhodné vytvářet **seznamy s obousměrným propojením** (viz příklad na obr. 24a). Pak můžeme výčet elementárních operací rozšířit i o operaci posunu vnitřního

ukazatele o jeden prvek vlevo. Na rozdíl od jednosměrně propojených seznamů lze zde jednoduše provést např. operaci vyjmutí posledního prvku seznamu.

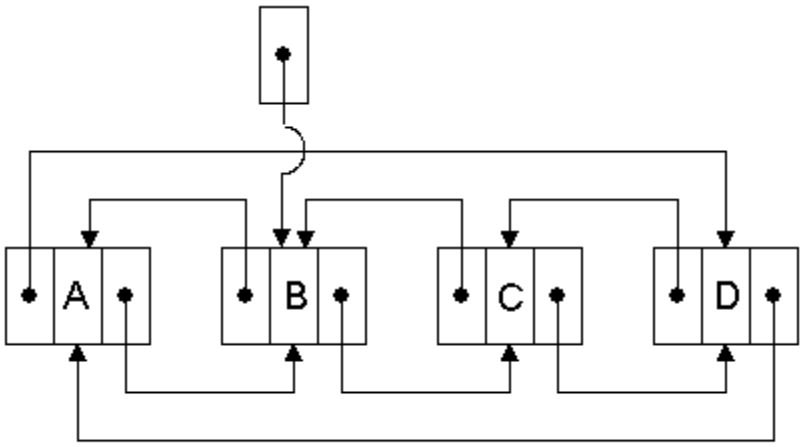


a)



b)

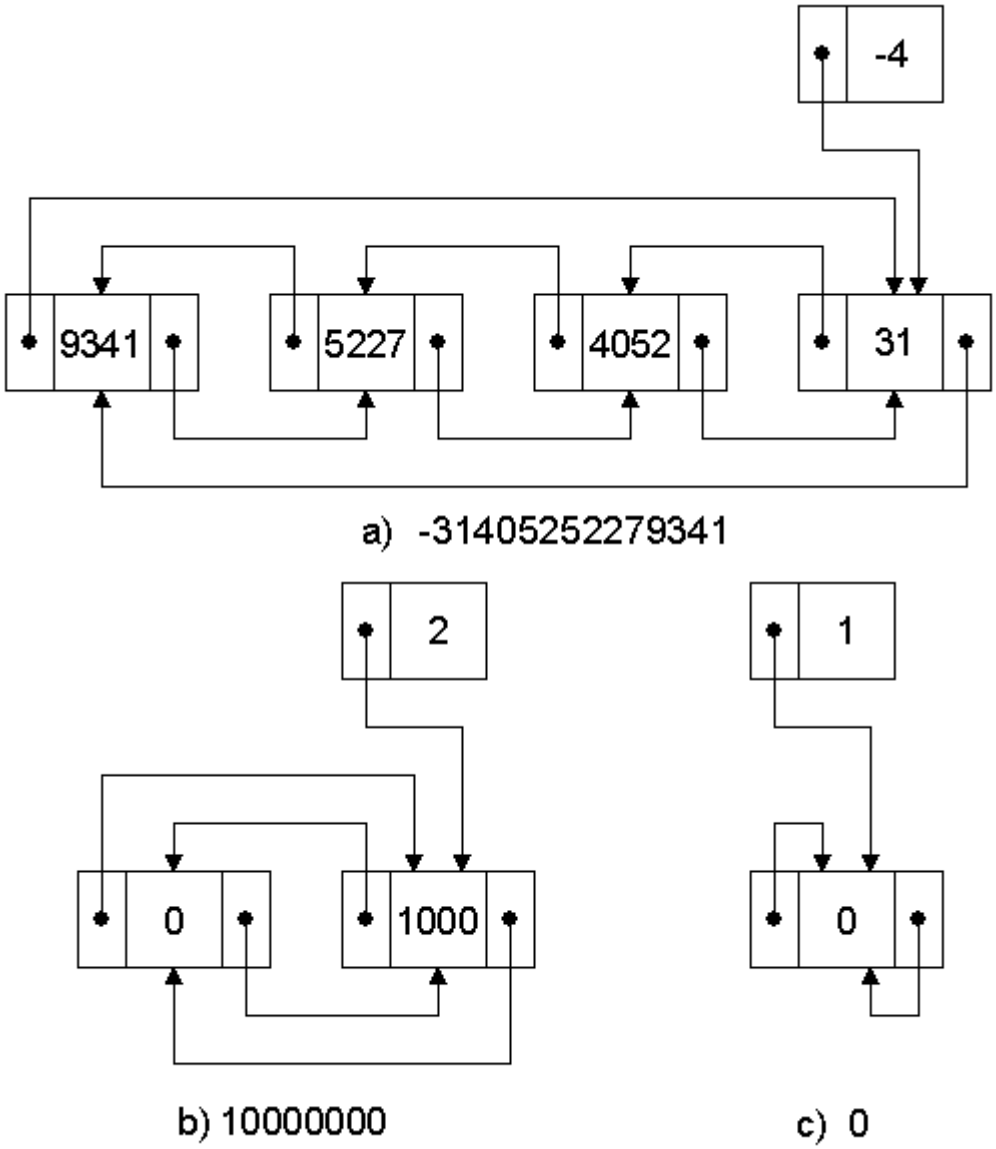
obr. 24



obr. 25

V mnoha konkrétních použitích může být výhodné uchovávat nejen ukazatel na první prvek seznamu, ale též ukazatele na poslední prvek seznamu (viz obr. 24b). Obecně lze říci, že obousměrně propojené seznamy je užitečné konstruovat zvláště v těch případech, kdy budou požadovány průchody v obou směrech. Lze též konstruovat i

obousměrně propojené kruhové seznamy (viz obr. 25), jejichž aplikaci demonstruje následující příklad.



obr. 26

Hardware počítačů dovoluje zobrazení jen konečné množiny celých čísel. V absolutní hodnotě větší (říkejme delší) hodnoty typu integer lze s výhodou reprezentovat pomocí obousměrně propojeného kruhového seznamu (viz příklady na obr. 26a-c). Uvažujme, že v každém prvku takového seznamu bude obsažena max. čtyřmístná hodnota typu integer (a kladná). Jak vyplývá z uvedených příkladů, tak budeme "udržovat" ukazatel na ten prvek seznamu, kde je uložena řádově nejvyšší část čísla. Absolutní hodnota dalšího údaje v záhlaví seznamu pak udává počet prvků v seznamu, zatímco znaménko tohoto údaje znamená znaménko čísla, reprezentovaného daným seznamem. To ovšem znamená, že nejprve budeme konstruovat obousměrně propojený kruhový seznam reprezentující absolutní hodnotu celého čísla a potom opatříme příslušný údaj o počtu prvků znaménkem.

Uvažujme např. následující typy :

```
typedef struct prvek {
        TYPPRVKU      hod ;
        struct prvek     *vlevo, *vpravo ;
} PRVEK ;

typedef struct  {
        PRVEK *kruh ;
        int    pocet ;
} CISLO ;
```

kde TYPPRVKU definujeme jako 16-ti bitové celé číslo (bez znaménka), nebo pokud bychom chtěli do seznamu ukládat více než čtyřmístná čísla, upravíme odpovídajícím způsobem dále uvedené funkce. Zápis funkce, která provede konverzi posloupnosti znaků reprezentují zápis “dlouhé“ hodnoty typu integer na obousměrně propojený kruhový seznam dle příkladů na obr. 26, je ponechán na cvičení. Rovněž tak i jiné užitečné funkce, která provádí opačnou konverzi. V daném kontextu lze pro takovou konstrukci použít funkce pro inicializaci obousměrně propojeného seznamu a pro vložení nového prvku do seznamu na řádově nejvyšší číslice čísla. Uvažujme, že tedy čtenář má připraveny svoje funkce pro realizaci takových akcí, jejichž prototypy budou mít tvar:

```
CISLO initc ( void ) ;
void insertc ( CISLO  *c, TYPPRVKU   x ) ;
```

Ještě dříve, než zapíšeme vyjádření aritmetických operací s takto reprezentovanými celočíselnými hodnotami, tak zapíšeme funkci, které jako argumenty předáme *c1* a c2 typu *CISLO,* a která vrací –1 v případě, že absolutní hodnota *c1* je menší než absolutní hodnota *c2,* v případě rovnosti těchto hodnot vrací 0, jinak vrací 1. Pro takové porovnání lze s výhodou použít údaj o počtu prvků seznamu v záhlaví :

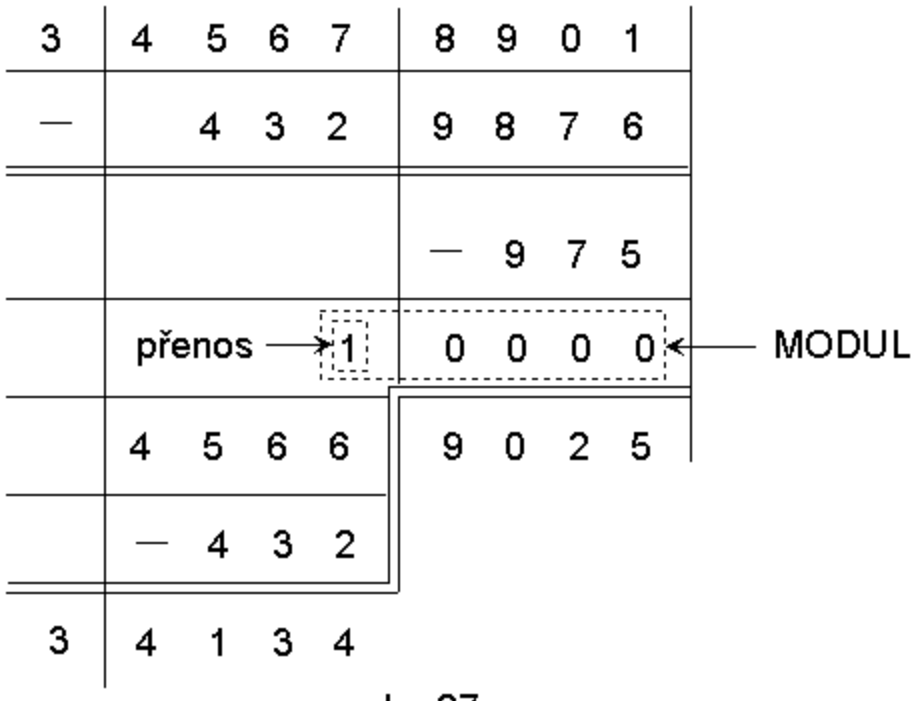
```
int relaceabs ( CISLO c1, CISLO c2 ) {

        PRVEK *p, *q;

        if ( abs ( c1.pocet )  > abs ( c2.pocet ) ) return ( 1 ) ;
        if ( abs ( c1.pocet )  < abs ( c2.pocet ) ) return ( -1 ) ;
```

```
        /*  Zápis c1 i c2 obsahuje stejný počet prvků kruhového seznamu,
            začneme porovnávání od číslic na nejvyšším řádu   */

        p = c1.kruh; q = c2.kruh;
        do  {
              if ( p->hod > q->hod ) return ( 1 ) ;
              if ( p->hod < q->hod ) return ( -1 ) ;
              p = p->vlevo; q = q->vlevo;
         } while ( p != c1.kruh );
         return ( 0 ) ;
}
```

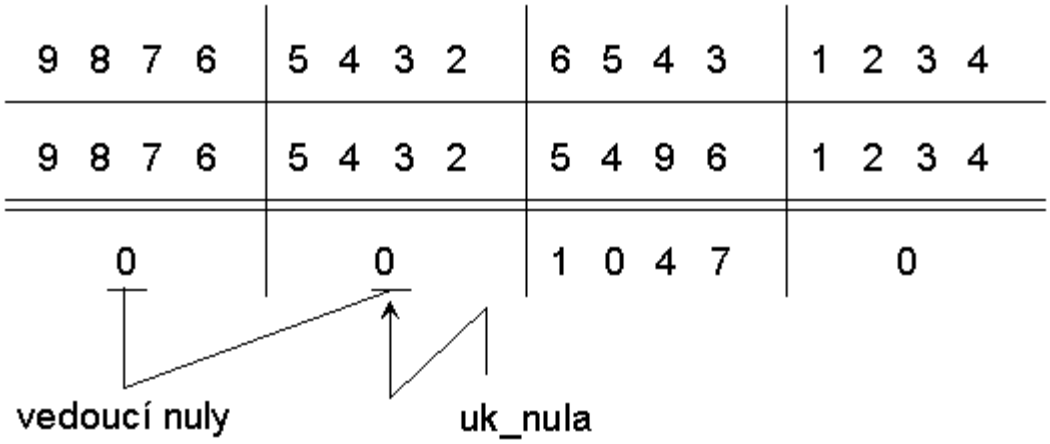
Napišme nejprve funkci se jménem *sumar,* kterou budeme používat pouze v těchto případech :

- absolutní hodnota čísla reprezentovaného seznamem *c1* není menší než absolutní hodnota čísla reprezentovaného seznamem *c2*,
- obě čísla reprezentovaná seznamy *c1* a *c2* mají opačná znaménka.

Od absolutní hodnoty prvního argumentu odečteme absolutní hodnotu druhého argumentu a rozdíl opatříme správným znaménkem. Přitom postupujeme podle všeobecně užívaného algoritmu pro odečítání od řádově nejnižších k řádově vyšším místům, tj. procházíme seznamy *c1* a *c2* doprava (viz obr. 27).



obr. 27



obr. 28

Kromě toho je žádoucí (nikoli však nutné) odstranit z výsledného seznamu tzv. vedoucí (nevýznamné) nuly (viz obr. 28). Za tím účelem je použita lokální proměnná *uk_nula,* odkazující na ten prvek výsledného seznamu, kde se nachází řádově nejnižší vedoucí nula. Funkce *sumar* pak může vypadat následovně :

```
/*  Součet dvou celých čísel c1 a c2, která mají opačná znaménka
    a  abs ( c1 ) >=  abs ( c2 )  */

#define   MODUL  10000

CISLO  sumar ( CISLO c1, CISLO c2 )  {

        CISLO c;
        PRVEK *p, *q, *uk_nula = NULL;
        int prenos = 0, rozdil;

        c = initc () ; p = c1.kruh; q = c2.kruh;

        /*  Začínáme na nejnižších řádech a potom opakovaně provádíme
            příkazovou část příkazu do až do vyčerpáni kratšího čísla      */

        do  {
              p = p->vpravo; q = q->vpravo;
              rozdil = p->hod - prenos - q->hod;
              if ( rozdil >= 0 ) prenos = 0;
              else {  /* Doplněk */
                 rozdil += MODUL; prenos = 1 ;
               }
              /* Vlož rozdíl do c */    insertc ( &c, rozdil ) ;
```

```
            /* Test, zda do c jsme vložili 0 */
            if ( rozdil ) uk_nula = NULL ;
            else if ( !uk_nula ) uk_nula = c.kruh;
        } while ( q != c2.kruh ) ;

        /* Průchod zbývající částí c1 */

        while ( p != c1.kruh )  {
            p = p->vpravo; rozdil = p->hod - prenos;
            if ( rozdil >= 0 ) prenos = 0;
            else {  rozdil += MODUL; prenos = 1; }
            insertc ( &c, rozdil ) ;
            if ( rozdil ) uk_nula = NULL ;
            else if  ( !uk_nula ) uk_nula = c.kruh;
        }

        /* Nyní ze seznamu odstraníme všechny vedoucí nuly */

         if ( uk_nula ) {     /* Odstraněni vedoucích nul */
            if ( uk_nula->vlevo == c.kruh ) uk_nula = uk_nula->vpravo;
            uk_nula->vlevo->vpravo = c.kruh->vpravo;
            p = c.kruh; c.kruh = uk_nula->vlevo;
            p->vpravo->vlevo = c.kruh;
            while ( p != c.kruh ) {
                q = p->vlevo; free ( p ) ;
                c.pocet--; p = q;
            }
        }

        /* Opatření čísla znaménkem */
        if ( ( c.pocet != 1 || c.kruh->hod ) && c1.pocet < 0 )  c.pocet *= -1 ;
        return ( c ) ;
}
```

Pro součet dvou celých čísel se shodnými znaménky reprezentovaných seznamy *c1* a *c2* použijeme funkci s prototypem :

```
CISLO sumas ( CISLO c1, CISLO c2 )  ;
```

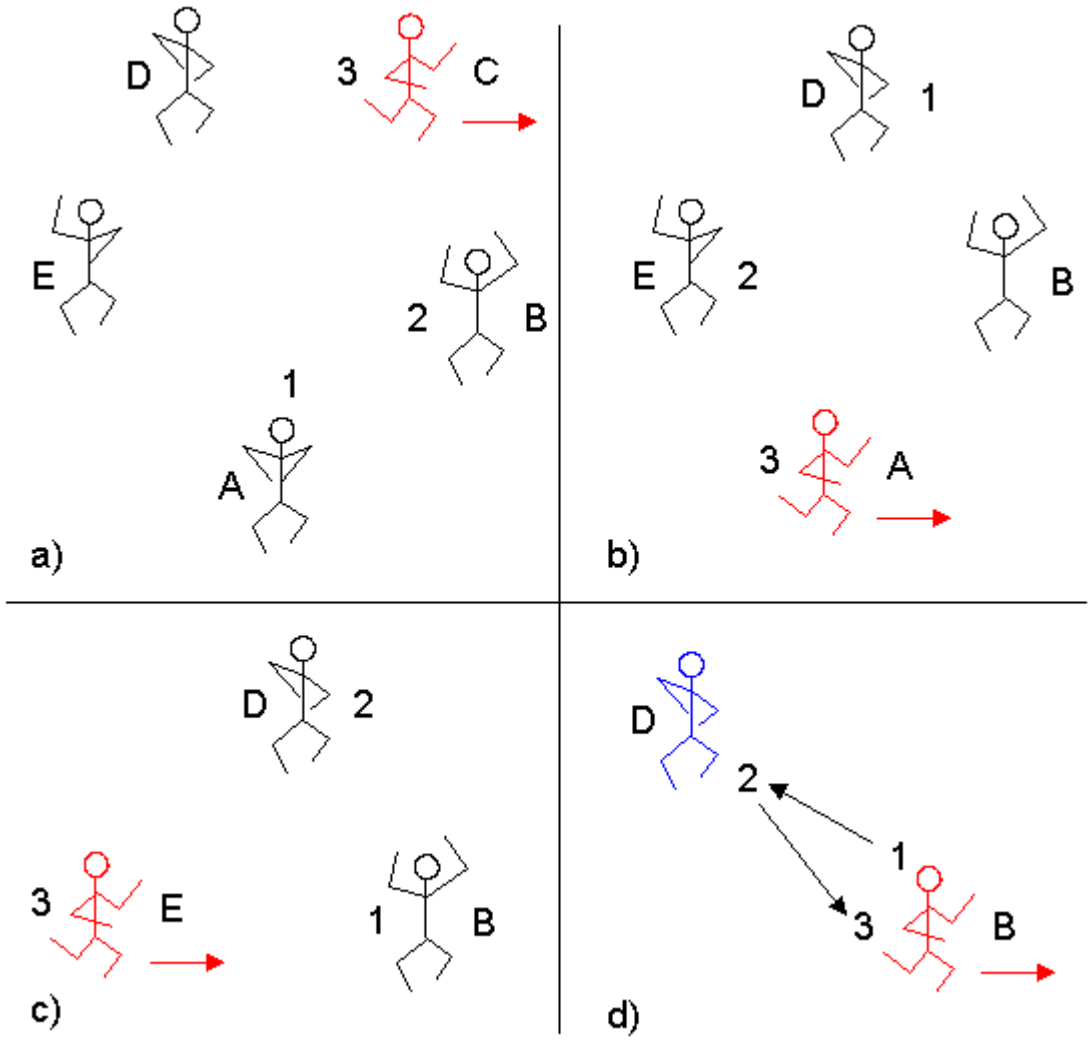
Rovněž vlastní zápis této funkce je ponechán na cvičení. Pak je již nasnadě, že pro součet dvou celých čísel, reprezentovaných seznamy *c1* a c2 použijeme následující funkci :

```
CISLO suma ( CISLO c1, CISLO c2 ) {
        CISLO c ;

        if ( c1.pocet*c2.pocet > 0 ) /* c1 a c2 jsou shodného znaménka */
                c = sumas ( c1, c2 ) ;
        else  if ( relaceabs ( c1, c2 ) == 1 )      c = sumar ( c1, c2 ) ;
                else                                c = sumar ( c2, c1 ) ;
        return ( c ) ;
}
```

4.4 Cvičení

1. Zapište svoje funkce realizující základní operace se seznamem, doplňující v textu již uvedené funkce *init*, *insertp* a *delete_s*. Dále pak napište funkci, která k danému seznamu vytvoří nový seznam, který bude obsahovat stejné prvky jako originální seznam, ale v opačném pořadí.
2. Zapište svoji implementaci seznamu pomocí pole (viz obr. 21 a návod v textu).
3. Napište funkce pro následující operace se seznamem (seznamy) :
   - sloučení svou seznamů *s1* a s2 do jednoho,
   - sloučení dvou uspořádaných seznamů *s1* a *s2* do jednoho uspořádaného seznamu s,
   - umístění nového prvku do uspořádaného seznamu,
   - odebrání každého druhého prvku ze seznamu,
   - určení počtu prvků seznamu,
   - pro záměnu *m*-tého a *n*-tého prvku seznamu.
4. Napište funkci, která v seznamu *s* vyhledá prvek s hodnotou *x* a vrátí ukazatel na tento prvek, nebo vrátí prázdný ukazatel v případě, že v seznamu *s* není prvek s hodnotou *x*. Dále napište funkci, která v případě, že seznam neobsahuje prvek s hodnotou *x*, takový prvek do seznamu vloží a vždy vrátí ukazatel na tento prvek.
5. Uvažujte frontu jako kruhový seznam tak, jak je to popsáno v předešlém textu a zapište svoje funkce pro realizaci základních operací s takto implementovanou frontou.
6. Simulujte pomocí kruhového seznamu následující úlohu, která bývá v anglosaské literatuře označována jako “Josephus problem”. Malá skupina vojáků je beznadějně obklíčena mnohonásobnou přesilou nepřátelských vojáků a nemůže doufat ve vítězství. Jedinou jejich záchranou může být únik na koni, potíž však je v tom, že mají k dispozici pouze jednoho koně. Vytvoří tedy kruh a dejme tomu z přilby vylosují jméno jednoho z nich a nějaké přirozené číslo *n*.

Počínaje vojákem s vylosovaným jménem začnou počítat v určeném směru až k číslu *n*, při jehož dosažení odstupuje z kruhu odpovídající voják a počítání pokračuje u následujícího vojáka v kruhu tak dlouho, až zůstane pouze jediný voják. Ten se pokusí uniknout na koni a přivolat pomoc obklíčeným vojákům. Jako příklad uvažujme, že kruh tvoří vojáci se jmény A, B, C, D a E, vylosován byl voják se jménem A a číslo 3 (tedy $n = 3$). Nastane situace podle obr. 29a, kdy po prvním počítání jde z kruhu ven voják C. Po druhém počítání jde z kruhu ven voják se jménem A (obr. 29b), po třetím voják E (obr. 29c) a po čtvrtém voják B (obr. 29d) Zůstává tedy voják D, který se pokusí o únik na koni.



obr. 29

7. Uvažujme následující modifikaci předchozího příkladu. V kruhu se nachází určitý počet osob a každá osoba si vybrala nějaké přirozené číslo. Dále je vybrána jedna z osob v kruhu a nějaké přirozené číslo *n*. Po prvním odpočítání je z kruhu eliminována *n*-tá osoba. Při každém

dalším odpočítávání se eliminace provádí nikoli podle hodnoty *n,* ale podle hodnoty přirozeného čísla, kterou si vybrala právě eliminovaná osoba. Tak např. pět osob A, B, C, D a E se nachází v kruhu a tyto osoby si vybraly čísla 3, 4, 6, 2 a 7 (tzn. A číslo 3, B číslo 4, atd.). V tomto výchozím stavu zvolíme osobu A a *n*=2. Pak posloupnost, ve které budou osoby eliminovány z kruhu bude B, A, E, C až nakonec zůstane osoba D.

8. Doplňte v textu chybějící funkce v příkladě pro součet celých čísel, reprezentovaných obousměrně propojeným kruhovým seznamem.

Literatura

[1] KERNIGHAN, B.W. – RITCHIE, B.M. : Programovací jazyk C. Bratislava, Alfa 1988.
[2] MÜLLER, K. : Programovací jazyky. Praha, ČVUT 1981.
[3] RYCHLÍK, J. : Programovací techniky. České Budějovice, KOPP 1995.
[4] SEDGEWICK, R. : Algorithms in C. Massachusetts, Addison-Wesley Publishing Company, Inc. 1990.
[5] TANENBAU, A. M. – AUGENSTEIN, M.J. : Data Structures Using Pascal. New Jersey, Prentice-Hall Inc. 1980.
[6] VĚCHET, V. : Počítače a programování. Díl III : Výběrový kurs Turbo Pascalu. Liberec, TU 1994.

PŘÍLOHA

ALGEBRAICKÁ SPECIFIKACE TYPU ZÁSOBNÍK

Syntaktická část (signatura) :

NEW	:	⟶ Zásobník	(generátor typu)
PUSH	:	Prvek x Zásobník	⟶ Zásobník
POP	:	Zásobník	⟶ Zásobník
TOP	:	Zásobník	⟶ Prvek
EMPTY?	:	Zásobník	⟶ boolean

Sémantická část (axiomy) :

TOP (PUSH (x, Z)) = x

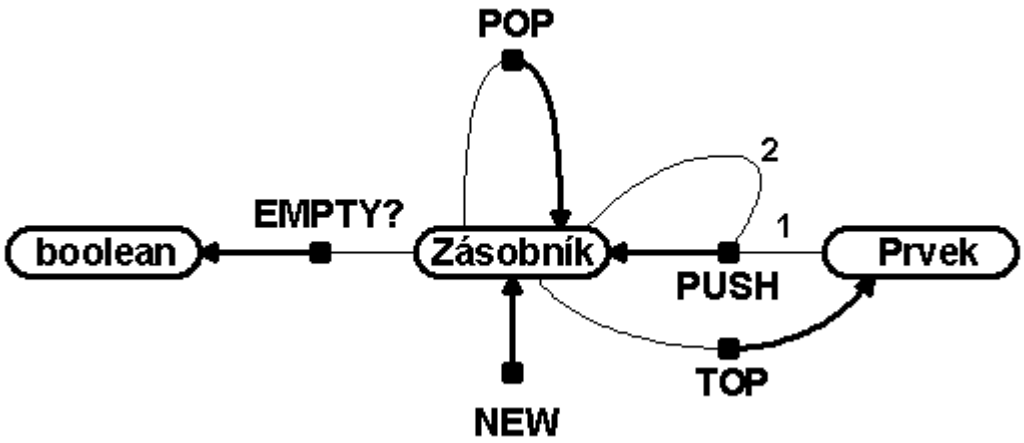
POP (PUSH (x, Z)) = Z

EMPTY? (NEW) = true

EMPTY? (PUSH (x, Z)) = false

POP (NEW), TOP (NEW) není definováno

Grafické znázornění signatury typu zásobník :



ALGEBRAICKÁ SPECIFIKACE TYPU FRONTA

Syntaktická část (signatura) :

NEW	:	⟶ Fronta (generátor typu)
ADD	:	Prvek x Fronta ⟶ Fronta
REMOVE	:	Fronta ⟶ Fronta
FRONT	:	Fronta ⟶ Prvek
EMPTY?	:	Fronta ⟶ boolean

Sémantická část (axiomy) :

EMPTY? (NEW) = true

EMPTY? (ADD (x, F)) = false

FRONT (ADD (x, NEW)) = x

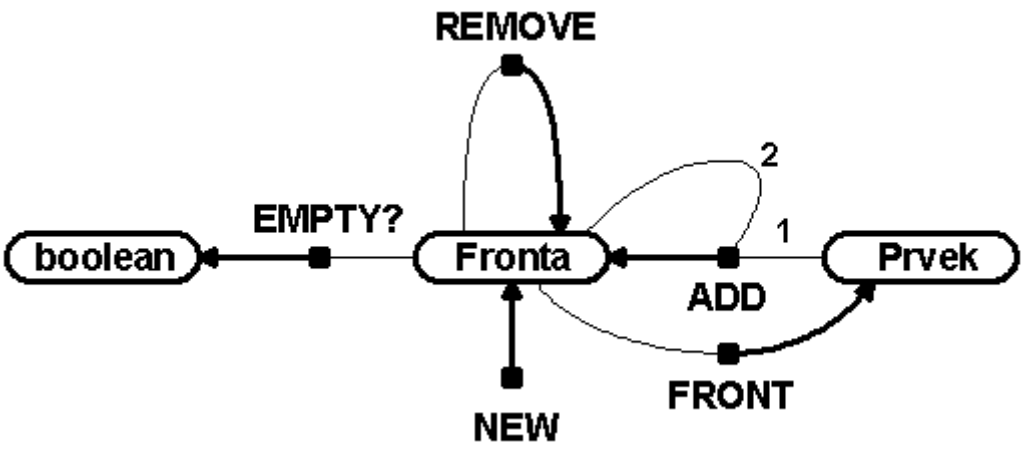
FRONT (ADD (y, ADD (x, F))) = FRONT (ADD (x, F))

REMOVE (ADD (x, NEW)) = NEW

REMOVE (ADD (y, ADD (x, F))) = ADD (y, REMOVE (ADD (x, F)))

FRONT (NEW), REMOVE (NEW) není definováno

Grafické znázornění signatury typu fronta :



ALGERBAICKÁ SPECIFIKACE TYPU SEZNAM

Syntaktická část (signatura) :

NEW	:	⟶ Seznam	(generátor typu)
CONS	:	Prvek x Seznam ⟶	Seznam
FIRST	:	Seznam ⟶	Seznam
P-INSERT	:	Prvek x Seznam ⟶	Seznam
N-INSERT	:	Prvek x Seznam ⟶	Seznam
DELETE	:	Seznam ⟶	Seznam
READ	:	Seznam ⟶	Prvek
NEXT	:	Seznam ⟶	Seznam
EMPTY?	:	Seznam ⟶	boolean
NULL?	:	Seznam ⟶	boolean

Sémantická část (axiomy) :

FIRST (NEW) = NEW

FIRST (FIRST (S)) = FIRST (S)

P-INSERT (x, NEW) = CONS (x, NEW)

P-INSERT (x, FIRST (S)) = CONS (x, FIRST (S))

P-INSERT (x, CONS (y, S)) = CONS (y, P-INSERT (x, S))

N-INSERT (x, NEW) není definováno

N-INSERT (x,FIRST(CONS (y, S))) =FIRST (CONS (y, CONS (x, S)))

N-INSERT (x, CONS (y, S)) = CONS (y, N-INSERT (x, S))

DELETE (NEW) = NEW

DELETE (FIRST (CONS (x, S))) = FIRST (S)

DELETE (CONS (x, S)) = CONS (x, DELETE (S))

READ (NEW) není definováno

READ (FIRST (CONS (x, S))) = x

READ (CONS (x, S)) = READ (S)

NEXT (NEW) není definováno

NEXT (CONS (x, S)) = CONS (x, NEXT (S))

NEXT (FIRST (CONS (x, S))) = CONS (x, FIRST (S))

EMPTY? (NEW) = true

EMPTY? (FIRST (S)) = EMPTY? (S)

EMPTY? (CONS (x, S)) = false

NULL? (NEW) = TRUE

NULL? (FIRST (CONS (x, S))) = false

Grafické znázornění signatury typu seznam :