TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta strojní **Katedra aplikované kybernetiky**

doc. Ing. Vladimír V ě c h e t , CSc.

ALGORITMY A DATOVÉ STRUKTURY

Liberec 1997

Recenzoval : doc. RNDr. Ing. Miloslav Košek, CSc.
ISBN 80 - 7083 - 246 - 0

OBSAH

Motto: *Největší zábavu skýtají právě ty počítačové úlohy, které nejsou v praxi naprosto k ničemu.*

Ze zákonů Murphyho.

1. ÚVOD

Učební text je určen studentům IV. ročníku strojní fakulty, kteří si zvolili studijní obor automatizované systémy řízení strojírenské výroby. Pokrývá jednu část obsahu předmětu Algoritmy a datové struktury, vyučovaného v tomto studijním oboru. Nenahrazuje přednášky z tohoto předmětu, ani studium literatury, doporučované na konci tohoto textu. Vytváří především zásobník (také abstraktní datová struktura) úloh pro samostatnou práci studentů. Vřele lze doporučit ke studiu literaturu [2], ze které je převážně čerpáno (pokud její získání je v možnostech čtenáře).

Pokud některé počítačové úlohy jsou zábavné, stávají se i zajímavé. Existuje velmi mnoho zajímavých problémů, které mají velmi silný praktický průnik. Autor doufá, že tomu tak je i u problémů, jejichž řešení je diskutováno v tomto učebním textu. Murphyho zákony fungují, ale nejsou objektivní.

Pro zápis algoritmů je volen záměrně jazyk C. Odpovídá to posloupnosti předmětů vyučovaných v oboru automatizované systémy řízení ve strojírenské výrobě!

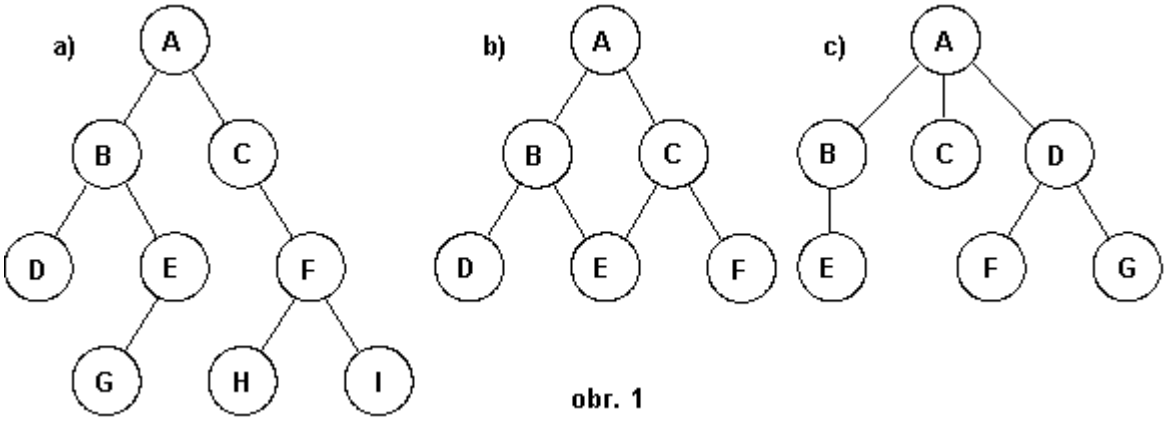
Za velmi pečlivé přečtení rukopisu a korektury textu děkuje autor Doc. RNDr. Ing. Miloslavu Koškovi, CSc.

2. STROMY

2.1 Binární stromy

Binární stromy (binary trees) jsou konečné množiny prvků, které jsou buď prázdné, nebo obsahují jeden prvek nazývaný **kořen stromu** (root of the tree) a všechny ostatní prvky jsou rozděleny do **dvou disjunktních** podmnožin, přičemž obě tyto podmnožiny jsou též binární stromy a nazývají se **levý a pravý podstrom** originálního stromu. Každý prvek stromu se označuje jako **uzel**.

Nejčastěji se používá grafická interpretace stromových struktur. Na obr. 1a je znázorněn binární strom s devíti uzly, kořen stromu je A, levý podstrom má kořen B a pravý C - to indikují dvě větve vycházející z A. Chybějící větev indikuje prázdný podstrom, např. na obr. 1a jsou levý podstrom k C a pravý podstrom k E prázdné. Uzly, z nichž nevycházejí žádné větve (na obr. 1a jsou to uzly D, G, H a I) se nazývají **listy**.



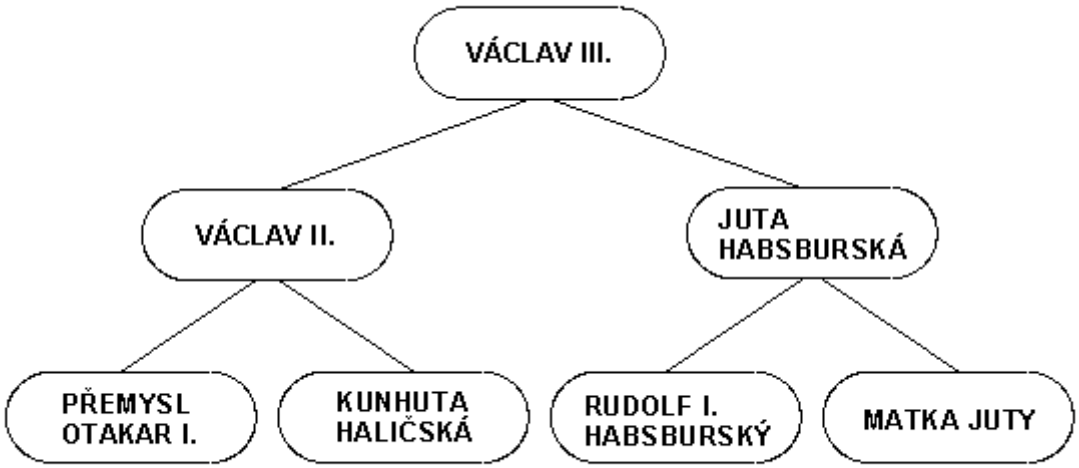
obr. 1

Z každého uzlu binárního stromu mohou vycházet nanejvýše dvě větve, ovšem struktura znázorněná na obr. 1b nemůže být binárním stromem, protože v definici se mluví o dvou disjunktních podmnožinách. Jak uvidíme později, tak struktura dle obr. 1b reprezentuje neorientovaný graf.

Ani struktura podle obr. 1c není binárním stromem. Jedná se o tzv. vícecestný strom, neboť z uzlu A vycházejí tři větve.

Úroveň (hladina) uzlu binárního stromu je definována následovně. Kořenu stromu je přiřazena úroveň 0, kořenům podstromů 1, atd. Tak např. na obr. 1a je uzel E na úrovni 2 a H na úrovni 3.

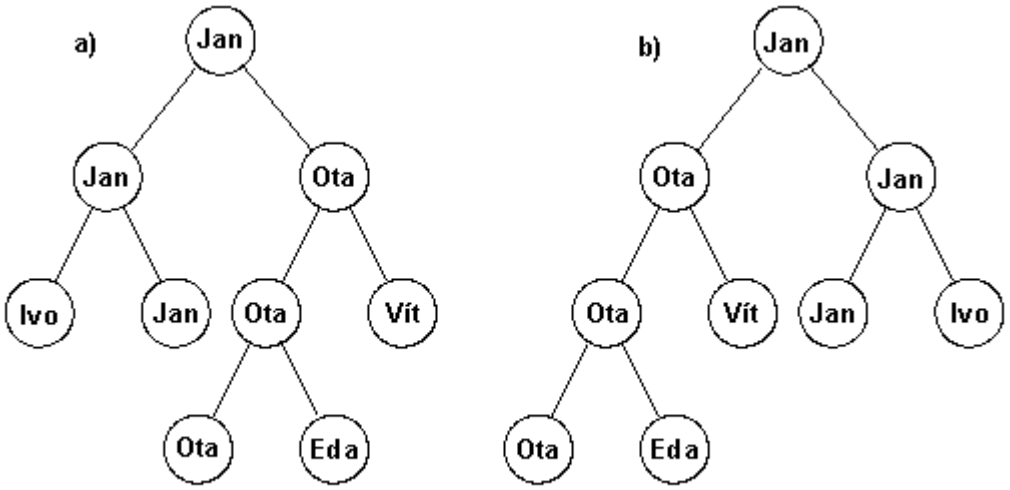
Často se též pro uzly binárního stromu používá označení "otec", "levý syn", "pravý syn", "bratři", "následník" - "potomek", "předchůdce" - "předek". Tak např. podle obr. 1a je B otec pro D i E, D je levý syn a E pravý syn B, D a E jsou bratři. E i G jsou praví následníci B, H je pravý následník A a levý následník F. Pokud to nebude s ohledem na stručnost vyjádření nutné, tak se těmto označením budeme vyhýbat. Zkuste si např. pomocí binárního stromu znázornit rodokmen !



obr. 2

Úplný binární strom úrovně *n* je takový binární strom, jehož všechny uzly na úrovni *n* jsou listy a každý uzel na úrovni menší než *n* má neprázdný levý i pravý podstrom. Podle této definice binární strom podle obr. 1a není úplný. Příkladem pro úplné binární stromy mohou být rodokmeny nějakých osob (viz obr. 2). Takové schéma je jistě čitelné a na první pohled je zřejmé, že např. dědečkem Václava III. byl Přemysl Otakar I. Kdybychom však na tomto příkladu aplikovali dříve vysvětlená označení "syn", "následník", atd., došli bychom k absurdnímu závěru, že např. česká královna Kunhuta je pravým synem Václava II. Pokud tedy takových označení uzlů binárních stromů budeme používat, tak pouze jako abstraktní vyjádření relativní polohy uzlů, kterému je třeba v daném kontextu přisoudit odpovídající význam.

Dále definujme tzv. **striktně binární stromy**, u kterých každý uzel, který není listem, má neprázdný levý i pravý podstrom. Příkladem může být znázornění výsledků tenisového turnaje ve dvouhře (viz obr. 3a). V předkole se utkal Ota a Eda, zatímco Ivo, Jan i Vít byli nasazeni přímo do prvního kola. Vítězem turnaje je Jan.

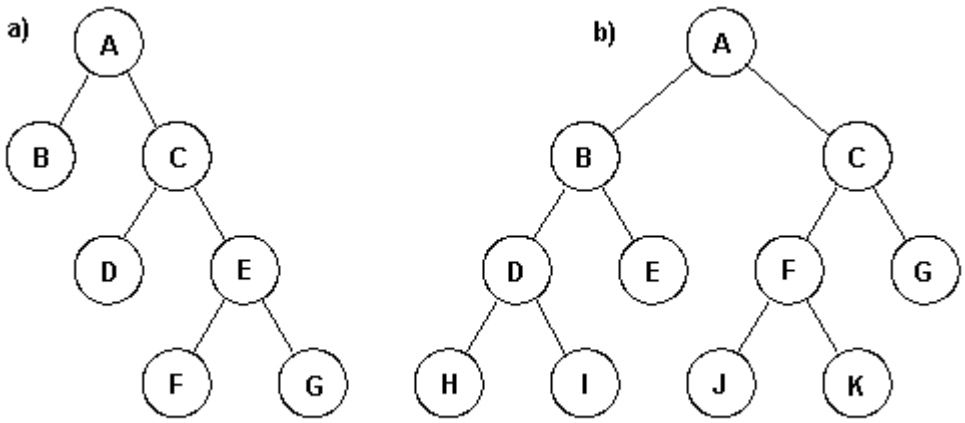


obr. 3

Téměř úplný binární strom je striktně binární strom, kde pro nějaké nezáporné celé *k* platí :

- všechny listy jsou na úrovni *k*, nebo *k*+1,
- pokud nějaký uzel binárního stromu má pravého následníka na úrovni *k*+1, tak všechny jeho leví následníci, kteří jsou listy, jsou také na úrovni *k*+1.

Není důvodu, proč bychom výsledky našeho tenisového turnaje neznázornili místo obr. 3a podle obr. 3b, který bude reprezentovat strukturu, označovanou jako téměř úplný binární strom. Struktury podle obr. 4a,b reprezentují striktně binární stromy, nikoliv však téměř úplné binární stromy. U binárního stromu podle obr. 4a jsou listy na úrovni 1, 2 i 3, tudíž je porušena první podmínka v uvedené definici.



obr. 4

Binární strom podle obr. 4b první podmínku sice splňuje, ovšem A má pravého následníka, který je listem na úrovni 3 (J), ale také levého následníka, který je listem na úrovni 2 (E) a tím je porušena druhá podmínka.

Poznámka : Někdy se z definice téměř úplných binárních stromů vypouští podmínka, že se musí jednat o striktně binární strom. Potom bychom mohli považovat za téměř úplný binární strom např. i takovou strukturu, která vznikne odebráním uzlu Eda z binárního stromu dle obr. 3b. Můžeme to např. chápat i tak, že Eda se urazil, protože by musel hrát v předkole a k zápasu proti Otovi odmítl nastoupit a ten pak postoupil do prvního kola bez boje.

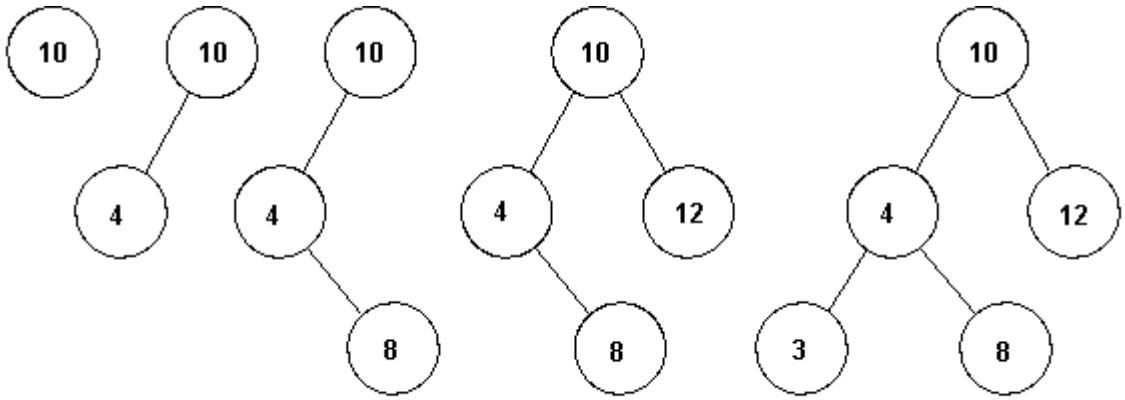
Spektrum použití binárních stromů je velmi široké. Jsou vhodné především pro reprezentaci takových procesů, v jejichž uzlových bodech se provádí rozhodování o dvou možnostech (vpravo-vlevo, ano-ne, menší nebo rovno - větší apod.).

Příklad

Nechť je dána nějaká posloupnost celých čísel a máme určit, která čísla se v posloupnosti vyskytují vícekrát. Tak např. v posloupnosti čísel

10 4 8 12 4 3 8 4

jsou to čísla 4 a 8. To lze zjistit i tak, že každé číslo x_i pro $i > 1$ porovnáme se všemi předcházejícími čísly (tedy x_1 , x_2 , ... , x_{i-1}). To ovšem předpokládá velký počet porovnávání, který lze snížit použitím binárního stromu.



obr. 5

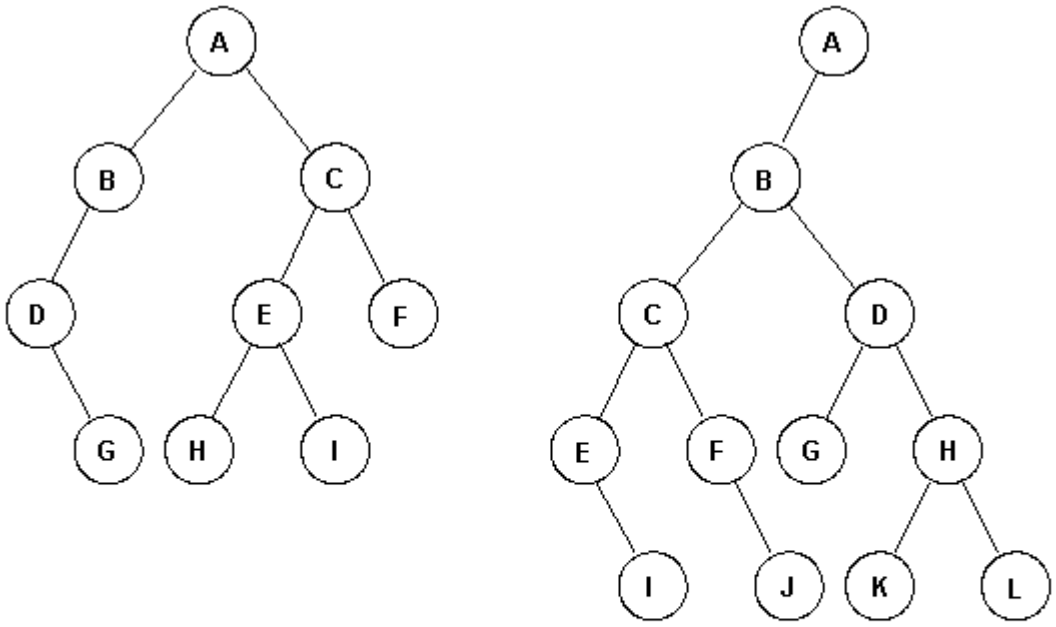
První číslo x_1 nechť je kořenem binárního stromu s prázdným levým i pravým podstromem. Každé následující číslo x_i , $i > 1$, se porovná s kořenem stromu - rovnost indikuje, že se toto číslo vyskytuje v posloupnosti vícekrát, je-li menší než kořen stromu, tak se totéž provede v levém podstromu, je-li větší než kořen stromu, tak v pravém podstromu. Tento postup se opakuje tak dlouho, až se buď zjistí, že x_i již v posloupnosti je, nebo se dosáhne prázdného podstromu a pak se x_i umístí do nového

uzlu místo prázdného podstromu. Pro uvedenou posloupnost je konstrukce odpovídajícího binárního stromu na obr. 5.

Konec příkladu

Jinou společnou operací nad binárním stromem je **průchod binárním stromem**. Dejme tomu, že nad každým uzlem binárního stromu chceme provést nějakou operaci, např. v nejjednodušším případě indikovat obsah každého uzlu. Za tímto účelem si budeme definovat tři metody průchodu. Tyto definice jsou rekursivní a mají tři kroky : operace nad kořenem binárního stromu, průchod levým podstromem, průchod pravým podstromem. Jednotlivé metody se liší pouze pořadím těchto kroků :

Průchod metodou preorder



obr. 6

- operace nad kořenem stromu ,
- průchod levým podstromem metodou preorder ,
- průchod pravým podstromem metodou preorder.

Průchod metodou inorder

- průchod levým podstromem metodou inorder ,
- operace nad kořenem stromu ,
- průchod pravým podstromem metodou inorder.

Průchod metodou postorder

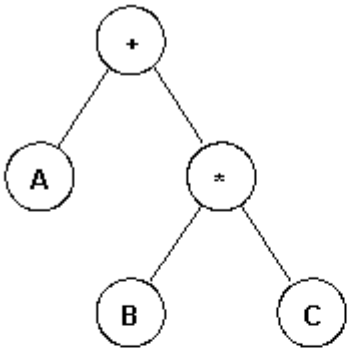
- průchod levým podstromem metodou postorder ,
- průchod pravým podstromem metodou postorder ,
- operace nad kořenem stromu .

Příklady jsou na obr. 6, kde je též vyznačeno pořadí uzlů při průchodu binárním stromem jednotlivými metodami.

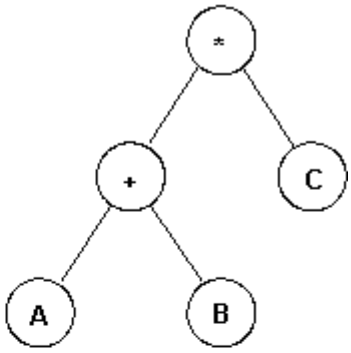
Jednou z oblastí aplikace binárních stromů je i reprezentace výrazů, které se skládají z operandů a binárních operátorů. Kořen takového binárního stromu obsahuje operátor, který je třeba aplikovat na výsledky výpočtu výrazů reprezentovaných levým a pravým podstromem. Příklady jsou na obr. 7. Každý uzel reprezentující operátor má dva neprázdné podstromy, zatímco každý uzel reprezentující operand má dva prázdné podstromy. V tomto smyslu se jedná o striktně binární stromy.

Průchod takovými stromy metodou preorder (postorder) generuje prefixový (postfixový) zápis výrazu. Pochopitelně uvažujeme vyšší prioritu multiplikativních operátorů než je priorita aditivních operátorů a při stejné prioritě vyhodnocování zleva doprava.

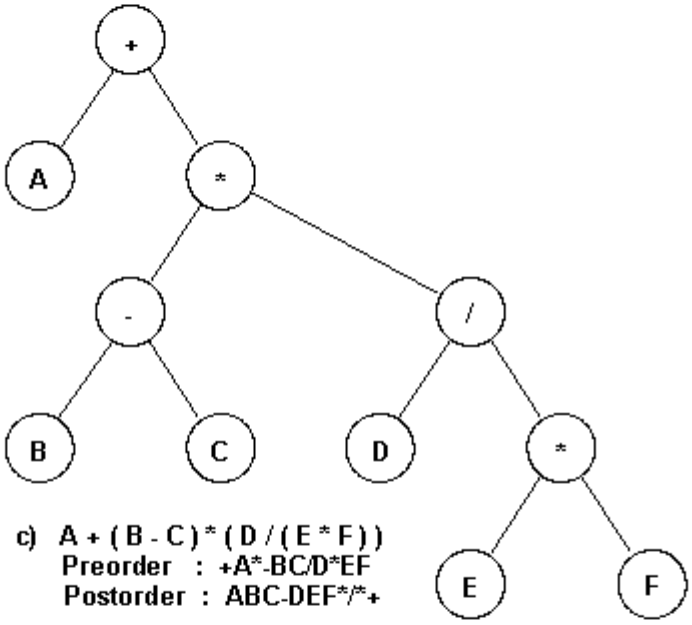
Všimněme si nyní průchodu takovými binárními stromy metodou inorder. Tak např. v příkladě podle obr. 7a i 7b generuje takový průchod stejný tvar : A + B * C, což však v případu podle obr. 7b není korektní zápis výrazu (chybí závorky) vzhledem k domluvené prioritě operátorů. Takže na takový případ lze aplikovat zmíněná pravidla jen s omezeními (závorky!), ale z průběhu průchodu takovým binárním stromem lze korektní infixovou verzi výrazu odvodit.



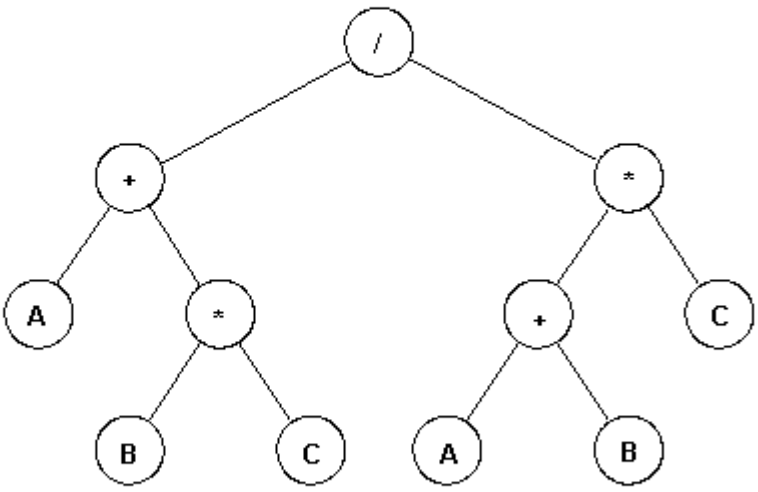
a) A + B * C
Preorder : +A*BC
Postorder : ABC*+



b) (A + B) * C
Preorder : *+ABC
Postorder : AB+C*



c) A + (B - C) * (D / (E * F))
Preorder : +A*-BC/D*EF
Postorder : ABC-DEF*/*+



d) (A + B * C) / ((A + B) * C)
Preorder : /+A*BC*+ABC
Postorder : ABC*+AB+C*/

obr. 7

Cvičení

1. Ukažte kolik předchůdců má uzel binárního stromu na úrovni *n* a jaký je maximální počet uzlů binárního stromu, které jsou na úrovni *n*.

2. Dokažte, že neprázdný striktně binární strom, který má *m* listů, má celkem 2*m*- 1 uzlů.

3. Říkáme, že dva binární stromy jsou podobné, když jsou buď oba prázdné, nebo když jsou neprázdné, tak jsou podobné jejich levé i pravé podstromy. Napište rekursivní algoritmus, který určí, zda dva dané binární stromy jsou podobné.

4. Říkáme, že dva binární stromy jsou zrcadlově podobné, když jsou buď prázdné, nebo když jsou neprázdné, tak levý podstrom obou binárních stromů je zrcadlově podobný pravému podstromu druhého binárního stromu, tj. porovnání levého podstromu s_1 s pravým podstromem s_2 a levého podstromu s_2 s pravým podstromem s_1 . Napište algoritmus, který určí, zda dva dané binární stromy jsou zrcadlově podobné.

5. Nechť operátor $ reprezentuje mocninu (např. 3$2 je 9) a má vyšší prioritu, než multiplikativní operátory. Reprezentujte následující výrazy pomocí binárních stromů a poté je zapište v prefixové a postfixové notaci :

A $ B * C - D + E / F / (G + H)

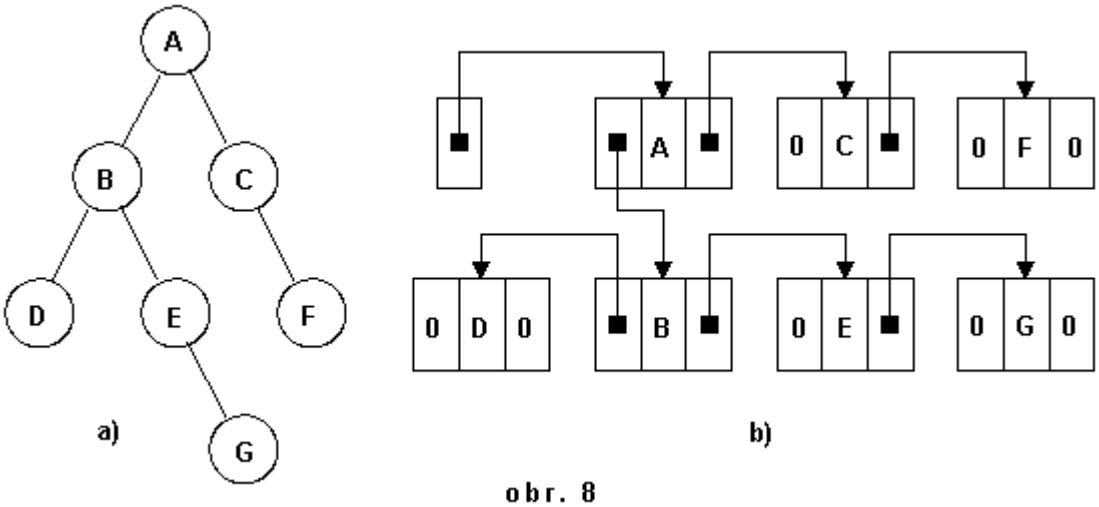
((A + B) * C - (D - E)) $ (F + G)

A - B / (C * D $ E)

2.2 Reprezentace binárních stromů

Reprezentace binárních stromů pomocí pole bude použita v následující kapitole, zabývající se jednou z mnoha aplikací binárních stromů. Zatím se tedy zabývejme spojovou reprezentací s dynamickou alokací. Fyzická interpretace binárního stromu podle obr. 8a je schematicky znázorněna na obr. 8b.

Prázdné stromy jsou indikovány hodnotou 0 (NULL, **nil**). Uvažujme nějaký dříve zavedený typ TYPINFO a použijme typ UZEL :



obr. 8

```
typedef struct uzel {
        TYPINFO  info ;
        struct uzel *levy;
        struct uzel *pravy;
 } UZEL ;
```

V mnoha případech lze pak výhodně pro operace s binárními stromy použít následující funkce :

```
#define ERROR     1
#define SUCCESS  0
/*      Umísti nový uzel jako kořen binárního stromu  s prázdným levým
        i pravým podstromem   */
UZEL *vytvor ( TYPINFO x )  {
        UZEL *u ;
        u = (UZEL *) malloc ( sizeof ( UZEL ) ) ;
        u->info = x ; u->levy = NULL ; u->pravy = NULL ;
        return ( u ) ;
 }


 /*     Přidej nový list, který ponese informaci předávanou parametru x
        a který bude levým synem uzlu, referencovaného pointerem p   */
```

```
int vlozvlevo ( UZEL *p, TYPINFO x ) {
        UZEL *q ;
        if ( !p )
         /* Chyba:není "otec" */ return ( ERROR ) ;
        else if ( p->levy )
                /* Chyba:levý "syn" již ex. */return ( ERROR) ;
            else {
                q = vytvor ( x ) ; p->levy = q ;
                return ( SUCCESS ) ;
                }
}

/*      Přidej nový list, který ponese informaci předávanou parametru x
        a který bude pravým synem uzlu, referencovaného pointerem p   */
int vlozvpravo ( UZEL *p, TYPINFO x ) {
        UZEL *q ;
        if ( !p )
          return ( ERROR ) ;
        else if ( p->pravy )
                return ( ERROR) ;
            else   {
                q = vytvor ( x ) ; p->pravy = q ;
                return ( SUCCESS ) ;
                }
}
```

Podle dříve uvedených rekursivních definic tří základních průchodů binárními stromy snadno zapíšeme odpovídající rekursivní funkce. Předpokládáme nějakou (např. externí) funkci operace, která realizuje ten krok algoritmu, který byl nazván "operace nad kořenem stromu" :

```
void preorder ( UZEL *strom ) {
        if ( strom ) {
                operace ( strom ) ;
                preorder ( strom->levy ) ;
                preorder ( strom->pravy ) ;
        }
}

void inorder ( UZEL *strom ) {
        if ( strom ) {
                inorder ( strom->levy ) ;
                operace ( strom ) ;
                inorder ( strom->pravy ) ;
        }
}

void postorder ( UZEL *strom ) {
        if ( strom ) {
                postorder ( strom->levy ) ;
                postorder ( strom->pravy ) ;
                operace ( strom ) ;
        }
}
```

Vraťme se nyní k reprezentaci výrazů s binárními operátory. Každý uzel binárního stromu nese informaci, jejíž hodnotou mohou být např. znaky A až Z pro operandy a +, -, *, - pro operátory. Jestliže požadujeme, aby operandy byly nějaké číselné hodnoty, tak každý list takového binárního stromu bude obsahovat číselnou hodnotu, zatímco ostatní uzly specifikaci operátoru. Obecně pak, pokud uzly binárního stromu obsahují informace různého typu, mluvíme o tzv. **heterogenních binárních stromech**. Jejich implementace je však ponechána na cvičení.

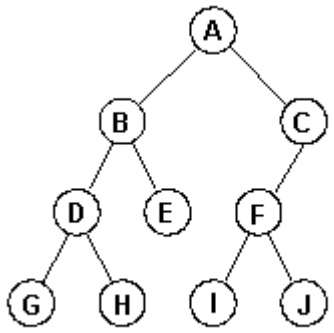
Cvičení

1. Napište nerekursivní verzi funkce pro průchod binárním stromem metodou inorder.

2. Je dán pointer `bs` na kořen binárního stromu a pointer `p` na nějaký uzel binárního stromu referencovaného hodnotou `bs`. Napište funkci (nejlépe rekursivní), která bude vracet pointer na ten uzel binárního stromu, který je "bratrem" uzlu referencovaného pointerem `p`. Vycházíme z typů, zavedených v textu.

3. Uvažte řešení předchozího příkladu pro případ, kdy každý uzel binárního stromu bude kromě pointeru na levý a pravý podstrom obsahovat též pointer na "otce".

4. Je dán pointer `bs` na kořen binárního stromu a pointery `p` a `q` na nějaké uzly binárního stromu. Napište funkci, která bude vracet pointer na "nejmladšího společného předchůdce" uzlů referencovaných `p` a `q`. Tak např. podle obr. 9 je to pro uzly G a H uzel D, pro G a E uzel B, pro J a D uzel A atd.



obr. 9

5. Uvažujme výrazy s binárními operátory a číslenými operandy. Zaveďte si odpovídající typ(y) a napište funkci (nejlépe rekursivní), která bude vracet hodnotu takového výrazu, reprezentovaného jako binární strom.

2.3 Huffmanův strom

Jako jiný příklad aplikace binárních stromů uveďme tzv. Huffmanův algoritmus. Nechť je dána nějaká abeceda sestávající z *n* symbolů a nějaký řetězec, obsahující symboly pouze z této abecedy. Uvažujme nyní nějaký binární abecední kód, který posloupnosti symbolů dané abecedy přiřadí posloupnost binárních 0 a 1.

Nechť např. abeceda obsahuje 4 symboly A, B, C a D a uvažujme následující 3-bitový kód :

Symbol :	Kód :
A	010
B	100
C	000
D	111

Pak např. posloupnost

ABACCDA

bude zakódována jako

01010001000000111010 .

Takové zakódování však nebude efektivní, neboť pro každý symbol jsou užity 3 bity a tudíž zakódování celého řetězce vyžaduje 21 bitů. Uvažujme proto 2-bitový kód :

Symbol :	Kód :
A	00
B	01
C	10
D	11

Pak stejný řetězec bude zakódován takto :

00010010101100 .

Zakódování pak vyžaduje celkem 14 bitů. Pokusme se nyní nalézt takový kód, aby délka zakódování daného řetězce symbolů byla minimální.

V našem příkladě se symboly B a D vyskytují v posloupnosti pouze jednou, zatímco symbol A třikrát. Použijme proto takový kód, který symbolu A přiřazuje kratší bitový řetězec, než symbolům B a D. Protože kratší kód (reprezentující symbol A) se bude vyskytovat v zakódovaném řetězci s největší frekvencí, tak délka zakódování může být menší. Tak např. pro

Symbol :	Kód :
A	0
B	110
C	10
D	111

bude : 0110010101110 ,

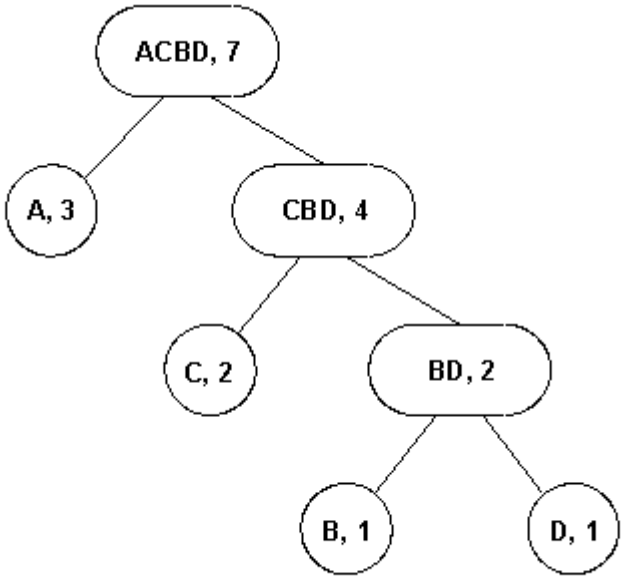
tedy délka celkem 13 bitů. Předpokládáme, že případné dekódování budeme provádět zleva doprava. Pak ovšem kód pro nějaký symbol nemůže tvořit počáteční část kódu pro jiný symbol. Tak např. v našem příkladě, je-li hodnota prvního bitu 0, musí se jednat o symbol A. V opačném případě se může jednat o jeden ze symbolů B, C, D a je třeba vyšetřit další bit. Je-li druhý bit 0, jedná se o symbol C, jinak se může jednat buď o B, nebo D a pak je třeba vyšetřit další, třetí bit. Je-li třetí bit 0, jedná se o B, jinak o D. Jakmile byl identifikován první symbol, celý postup se opakuje od následujícího bitu vpravo.

Vytváření takového kódu ovšem předpokládá znalost frekvence výskytu jednotlivých symbolů abecedy. Najděme pak nejprve dva symboly, které se vyskytují s nejmenší frekvencí. V našem příkladě jsou to symboly B a D. Poslední bity jejich kódu musí být rozdílné : 0 pro B a 1 pro D. Sloučíme tyto dva symboly do jednoho symbolu BD, jehož kód reprezentuje znalost, že symbol je buď B, nebo D. Frekvence výskytu tohoto nového symbolu je součtem frekvencí výskytu obou symbolů, ze kterých byl vytvořen. V našem příkladě bude frekvence výskytu BD rovna 2. Nyní máme 3 symboly : A (frekvence 3), C (frekvence 2) a BD (frekvence 2). Znovu zvolíme dva symboly s nejmenší frekvencí, nyní tedy C a BD. Poslední bit jejich kódu musí být odlišný, tj. 0 pro C a 1 pro BD. Získáme tak nový symbol CBD s frekvencí 4 a zbývají nám pouze dva symboly A a CBD a pak poslední bit kódu pro A bude 0 a pro CBD 1. Sloučením získáme symbol ACBD, který již obsahuje všechny symboly naší abecedy.

Symbolu ACBD přiřazuje kód prázdný řetězec bitů, neboť na začátku dekódování (předtím, než byla vyšetřována hodnota bitů) je jisté, že každý symbol je obsažen v ACBD. Dva symboly, jejichž sloučením vznikl symbol ACBD, mají kód 0 (pro A) a 1 (pro CBD). Podobně pak pro CBD bude kód 10 pro C a 11 pro BD. První bit pak indikuje, že se jedná o jeden ze symbolů, jejichž sloučením vznikl symbol CBD

- tedy buď C, nebo BD. Druhý bit pak indikuje, zda se jedná o C, nebo BD. Nakonec pak symbolům tvořícím BD je přiřazen kód 110 (pro B) a 111 (pro D).

Právě operace sloučení dvou symbolů do jednoho předpokládá užití binárního stromu. Všechny uzly takového binárního stromu, které nejsou listy, reprezentují symboly, listy reprezentují symboly originální abecedy. Obr. 10 ilustruje náš příklad. V každém uzlu je uveden symbol a dále i frekvence výskytu tohoto symbolu. Jiný příklad je na obr. 11 - odpovídá hodnotám uvedeným v tab. 1 (*f* je frekvence výskytu symbolů nějaké abecedy).



obr. 10

Binární stromy tohoto druhu se nazývají **Huffmanovy stromy**. Jsou to striktně binární stromy.

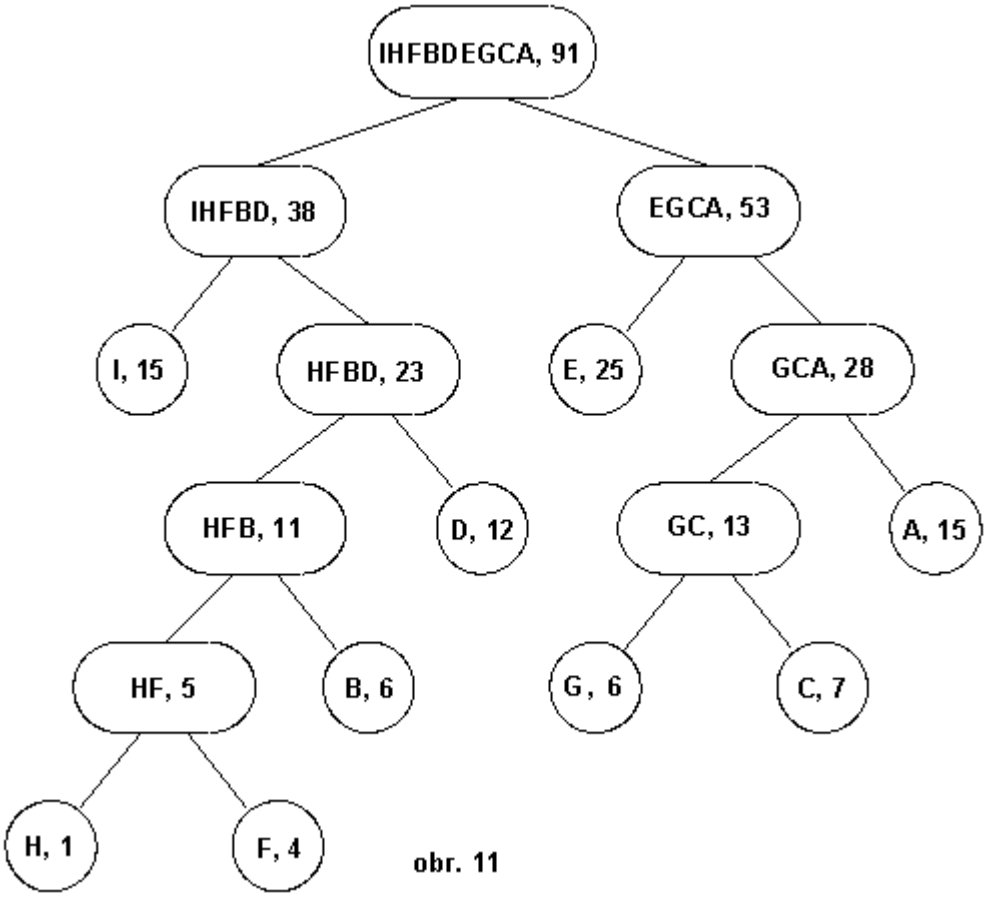
Pokud je zkonstruován takový binární strom, tak lze určit kód každého symbolu abecedy následujícím způsobem. Začíná se v listu reprezentujícím daný symbol a postupuje se směrem ke kořeni stromu. Kód je inicializován jako prázdný. Pohyb po levé větvi vyvolá přidání 0 zleva ke kódu, zatímco pohyb po pravé větvi znamená přidání 1 zleva ke kódu.

Ke konstrukci Huffmanova stromu a získání kódu je postačující, když každý uzel obsahuje ukazatel na "otce", informaci, zda daný uzel je levým, nebo pravým synem a frekvenci výskytu symbolu reprezentovaného daným uzlem. Protože Huffmanovy stromy jsou striktně binární stromy a každý list Huffmanova stromu reprezentuje jeden z *n* symbolů originální abecedy, tak celkový počet uzlů

Huffmanova stromu je 2 *n* - 1 a můžeme zde s výhodou použít implementaci takového stromu pomocí pole.

Tab. 1

Symbol	**f**	**Kód**	**Symbol**	**f**	**Kód**	**Symbol**	**f**	**Kód**
A	15	111	D	12	011	G	6	1100
B	6	0101	E	25	10	H	1	01000
C	7	1101	F	4	01001	I	15	00



obr. 11

Je-li dáno zakódování symbolů abecedy (jako posloupnost 0 a 1) a odpovídající Huffmanův strom, lze snadno provést i dekódování. Vycházejíc z kořene tohoto stromu pokaždé, když je v zakódování 0, tak se přechází ke kořenu levého podstromu a pro 1 ke kořenu pravého podstromu a to tak dlouho, až se dosáhne listu a ten reprezentuje hledaný symbol.

V příloze 1 je jedna z mnoha možností zápisu algoritmu pro kódování pomocí Huffmanova stromu. Tato verze byla zkoušena na hardwarové platformě DEC s operačním systémem ULTRIX. Modifikace funkcí (nebo ještě lépe zápis vlastních funkcí) pro řešení tohoto problému na konkrétní hardwarové platformě, včetně úprav vstupů, je ponecháno na cvičení. Poznamenejme pouze, že pokud je dána nějaká posloupnost symbolů abecedy, tak frekvencí výskytu nějakého symbolu jsme rozuměli počet opakování tohoto symbolu v dané posloupnosti. Pokud máme na mysli obecně nějaký text jako posloupnost znaků, tak frekvence výskytu mohou být jakési průměrné relativní četnosti výskytu (v %) daného symbolu ve všech různých variantách textu, resp. relativní četnosti získané analýzou daného textu.

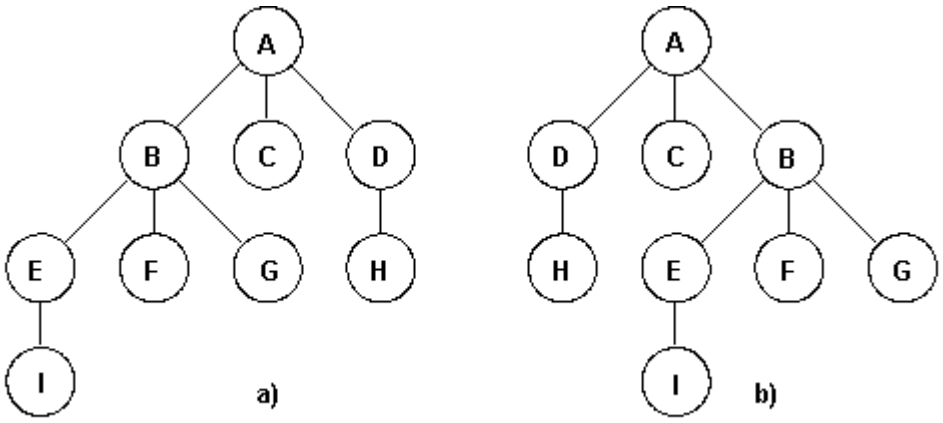
Cvičení

1. Napište svoje funkce, nebo modifikujte funkce podle přílohy 1, pro kódování a dekódování textových souborů Huffmanovým algoritmem.

2. Upravte datové struktury a algoritmus vytváření Huffmanova stromu tak, aby se zrychlila operace dekódování.

2.4 Vícecestné stromy a lesy

Vícecestné stromy budeme v dalším textu označovat krátce jen jako stromy. **Strom** je konečná neprázdná množina prvků, z nichž jeden se nazývá **kořen** a ostatní prvky jsou rozděleny do disjuktních podmnožin, z nichž každá je též stromem. Všechny prvky se nazývají **uzly** stromu. Každý uzel může být kořenem stromu s prázdným podstromem, či více podstromy. Uzly bez podstromů se nazývají **listy**. Pojmy "otec", "syn", "předchůdce", "následník" a "úroveň uzlu" budeme používat ve stejném smyslu, jako u binárních stromů. Dva uzly, které mají společného "otce", se nazývají "bratři". **Stupeň uzlu** je pak definován jako počet jeho "synů".

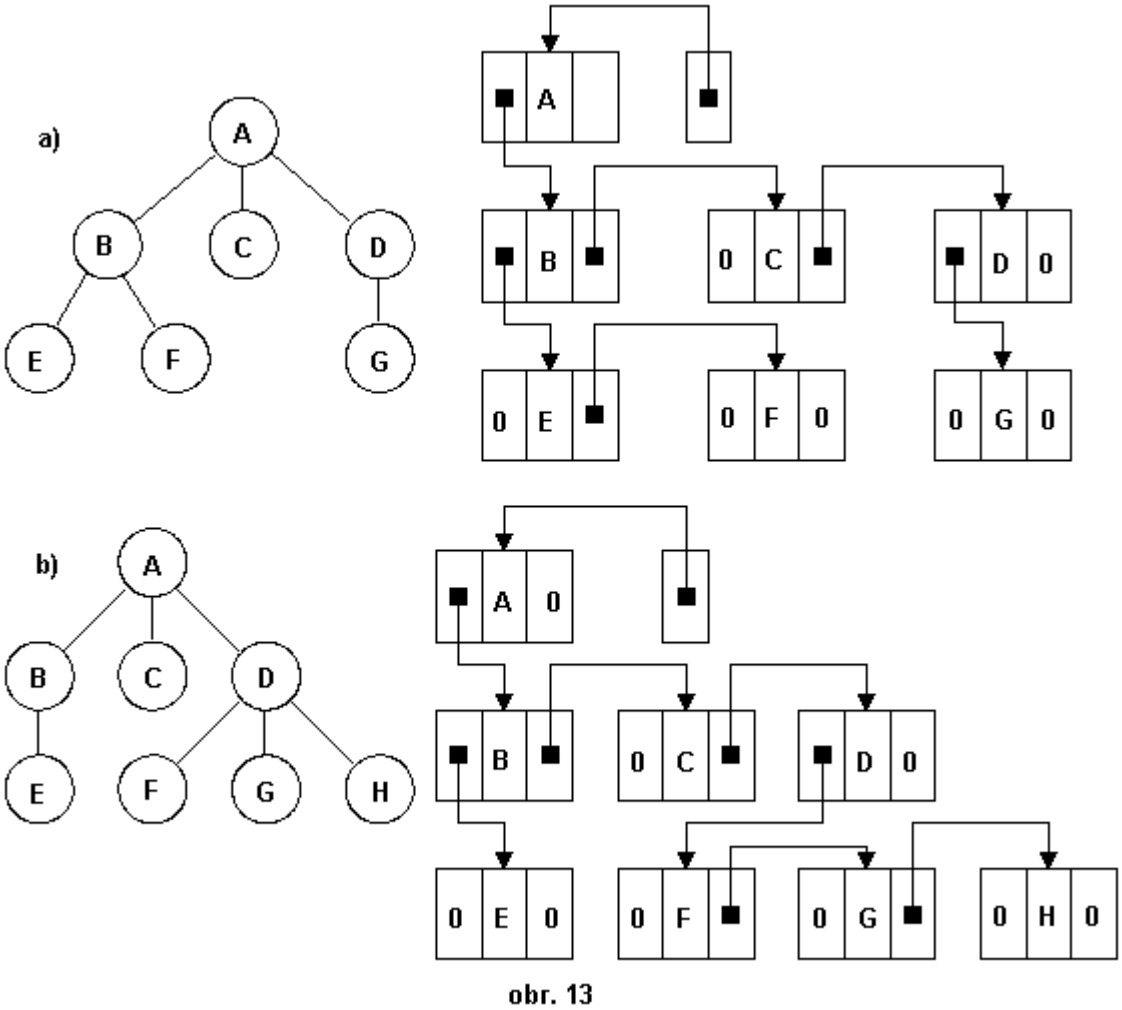


obr. 12

Tak např. dle obr. 12 uzel A má stupeň 3, D stupeň 1, C stupeň 0 atd. Porovnejme nyní stromy na obr. 12a a 12b. V obou případech je A kořenem stromu se 3 podstromy, z nichž C je listem, další s kořenem D a jedním podstromem a třetí s kořenem B a 3 podstromy atd. Z hlediska definice stromu se tedy jedná o stejné stromy. Liší se však uspořádáním podstromů (v případě binárních stromů je uspořádání vyjádřeno termíny levý a pravý podstrom). Definujme proto **uspořádaný strom** jako takový strom, u kterého podstromy všech uzlů tvoří uspořádanou množinu. U uspořádaných stromů pak můžeme mluvit o prvním, druhém, až posledním "synovi" nějakého uzlu, z nichž první bývá označován jako **nejstarší syn** a poslední jako **nejmladší syn**. Tak např. na obr 12a je D nejmladším synem A, zatímco na obr. 12b je D nejstarším synem A. Tudíž dle obr. 12a a 12b se jedná o dva různé uspořádané stromy, které jsou ekvivalentní pouze jako neuspořádané stromy. V dalším textu budeme užívat pojmu strom pro označení uspořádaných stromů.

Definujme ještě tzv. **les** (forest) jako uspořádanou množinu uspořádaných stromů. Pak má také smysl mluvit o prvním, či posledním stromu v lese apod.

Poznámka : Pochopitelně, že každý neprázdný binární strom je také stromem. Naopak to však neplatí. Rovněž strom, jehož všechny uzly mají nanejvýše dva syny, nemusí být nutně binární., protože když nějaký uzel má pouze jednoho syna, nemusí být tento nutně označen jako levý nebo pravý syn, zatímco pro binární stromy je takové označení nutné. Můžeme však říci, že každý neprázdný binární strom je zároveň stromem, jehož všechny uzly mají dva podstromy označované jako levý a pravý podstrom.



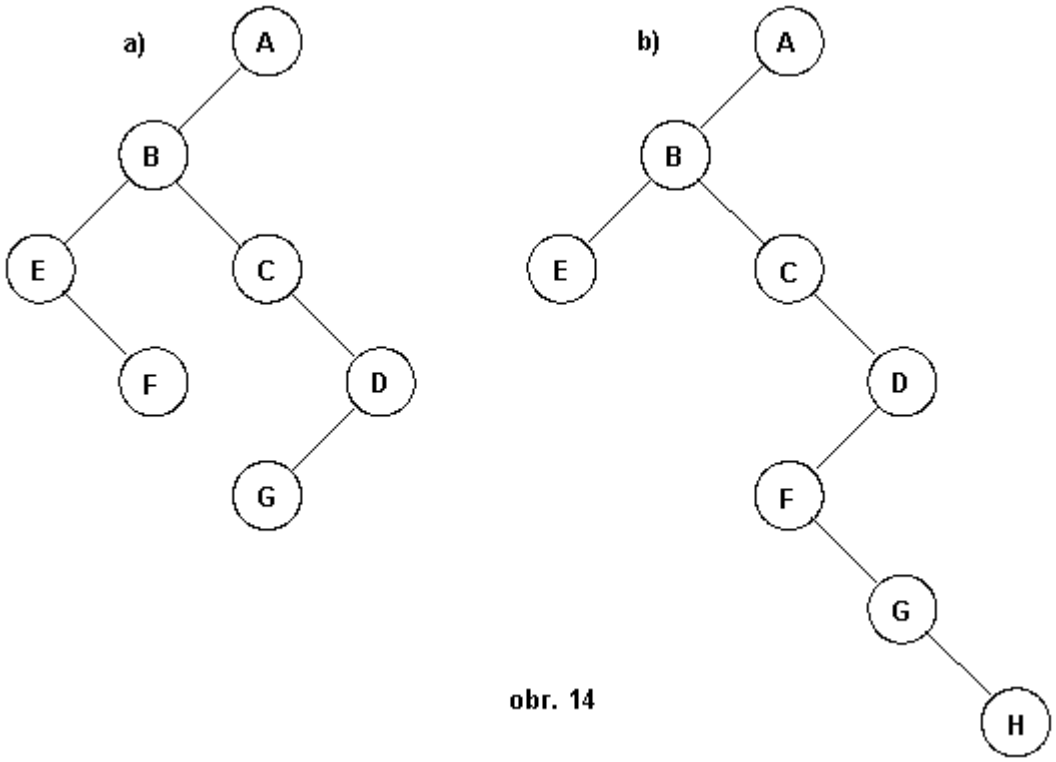
obr. 13

Strom lze samozřejmě implementovat pomocí pole. Ovšem na rozdíl od binárních stromů, kde každý uzel nesl jistou informaci a dále obsahoval dva ukazatele (na pravého a levého syna), tak zde počet synů není teoreticky omezen. Proto budeme využívat spojovou reprezentaci s dynamickou alokací (viz dva příklady na obr. 13). Jde tedy o vytváření lineárních seznamů synů se společným otcem. Vycházíme-li z

dříve zavedeného typu TYPINFO pro informaci obsaženou v uzlu stromu, tak můžeme použít např. typu UZEL :

```
typedef struct uzel {
        TYPINFO  info ;              /* Informace obsažená v uzlu */
        struct uzel *nsyn, *bratr ;  /* Pointer na "nejstaršího syna"  a na "bratra"  */
} UZEL ;
```

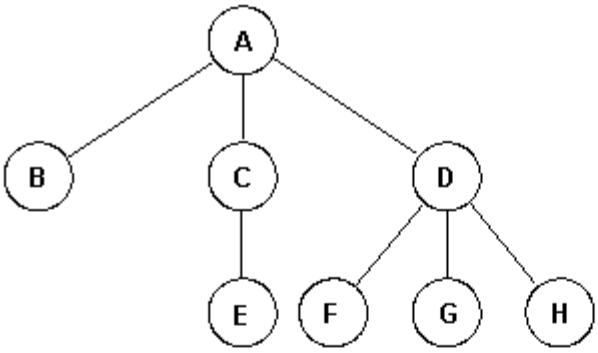
Přisuďme nyní pointeru nsyn význam ukazatele na levý podstrom binárního stromu a pointeru bratr význam ukazatele na pravý podstrom binárního stromu. Pak se ovšem jedná o reprezentaci stromů (vícecestných a uspořádaných) pomocí binárních stromů. Tak např. pro stromy dle obr. 13a a 13b lze odpovídající binární stromy ilustrovat obr. 14a a 14b.

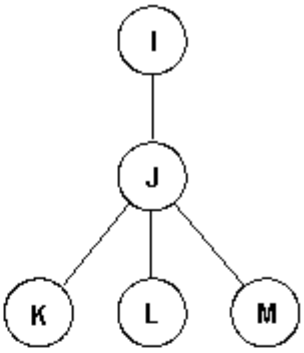


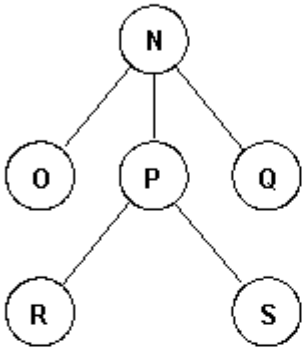
obr. 14

Jestliže pointeru bratr v kořeni stromu přisoudíme význam ukazatele na další strom lesu, tak můžeme pomocí binárních stromů reprezentovat celý les. Jeden takový příklad je na obr. 15, kde pro les sestávající ze tří stromů je nakreslen odpovídající binární strom.

Metody průchodu lesem mohou být definovány pomocí průchodů odpovídajícím binárním stromem metodou preorder, inorder a postorder. Lze je též definovat přímo takto:



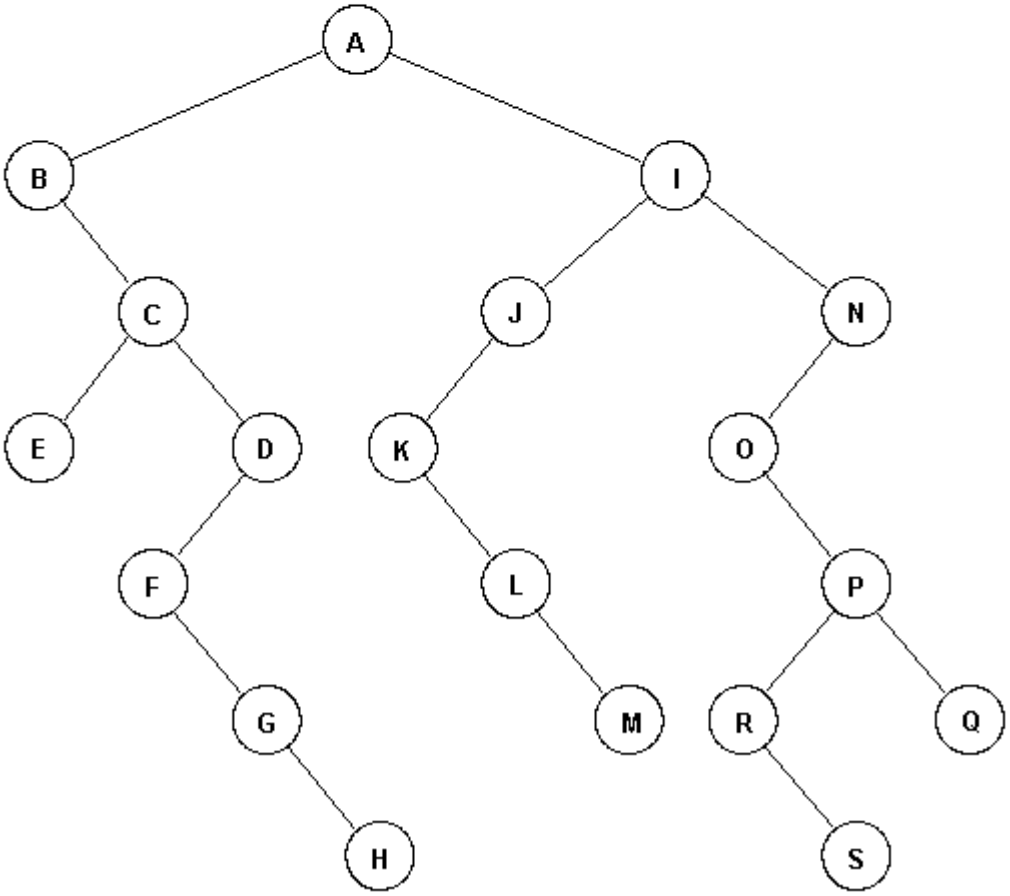




LES : **kořen 1. stromu je A ,**

kořen 2. stromu je I ,

kořen 3. stromu je N



obr. 15

Průchod metodou preorder

- operace nad kořenem prvního stromu v lese ,
- průchod lesem tvořeným podstromy prvního stromu (pokud takové existují) metodou preorder ,
- průchod zbývajícími stromy lesa (pokud nějaké existují) metodou preorder .

Průchod metodou inorder

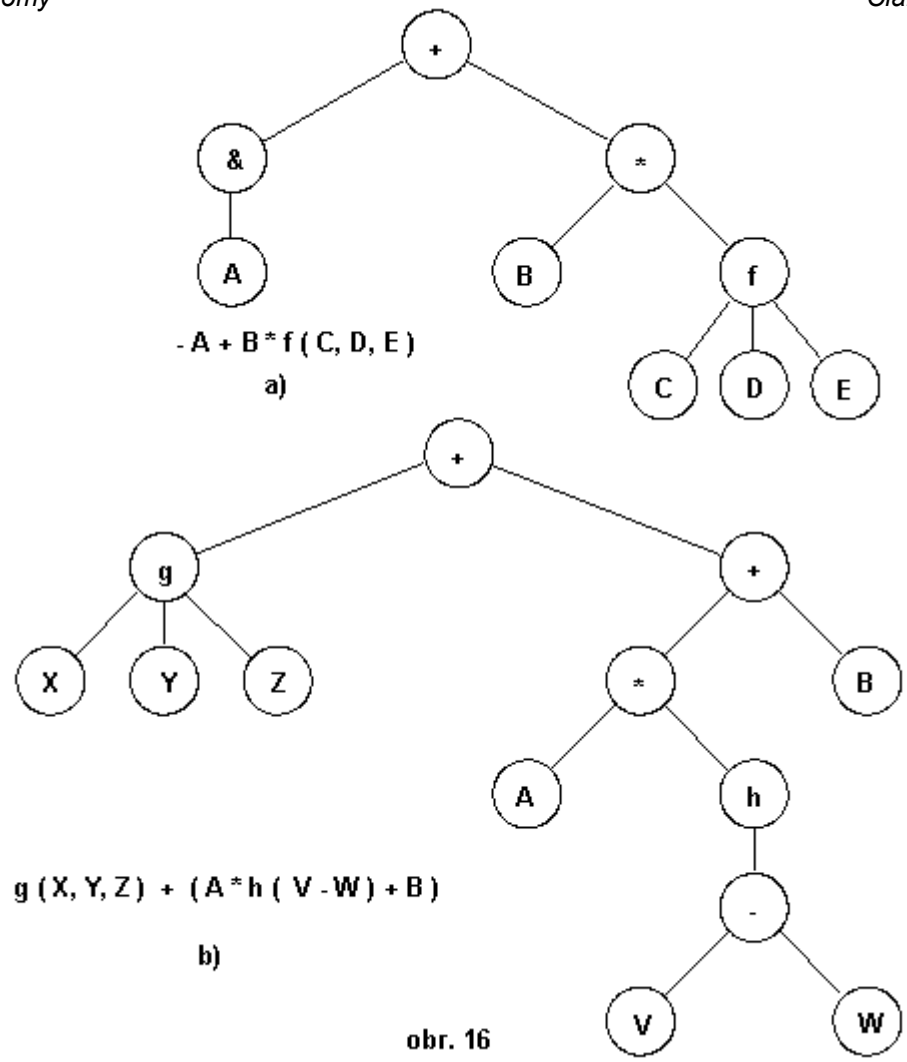
- průchod lesem tvořeným podstromy prvního stromu (pokud takové existují) metodou inorder ,
- operace nad kořenem prvního stromu v lese ,
- průchod zbývajícími stromy lesa (pokud nějaké existují) metodou inorder .

Průchod metodou postorder

- průchod lesem tvořeným podstromy prvního stromu (pokud takové existují) metodou postorder ,
- průchod zbývajícími stromy v lese (pokud nějaké existují) metodou postorder ,
- operace nad kořenem prvního stromu v lese .

Podobně jako byly dříve užity binární stromy pro reprezentaci binárních výrazů (tj.výrazů, které se skládají z operandů a binárních operátorů), tak lze užít stromů pro reprezentaci obecných výrazů. Dva takové příklady jsou znázorněny na obr. 16. Symbol "&" zde reprezentuje unární mínus, zatímco symbol "-" reprezentuje binární operátor odečítání. Zápis např. *f (C, D, E)* je aplikací operátoru *f* na operandy *C*, *D* a *E*. Všimněte si nyní, že **průchod takovými stromy metodou preorder generuje prefixový zápis výrazu**, zatímco **průchod metodou inorder generuje postfixový zápis výrazu** !! Tak např. v příkladě podle obr. 16a bude metoda preorder generovat *+&A*BfCDE*, tedy prefixový výraz, ovšem metoda inorder generuje *A&B*CDEf+*. Tuto zdánlivou nesrovnalost lze ozřejmit transformací, vyvolanou reprezentací stromů pomocí binárních stromů.

Pro bližší vysvětlení uvažujme příklad dle obr. 17. Jestliže stromy na obr. 17a, b chápeme jako binární stromy, tak průchody metodami preorder, inorder a postorder generují posloupnosti uvedené na tomto obrázku. Všimněme si shody u metody preorder. Jestliže však chápeme obr. 17 jako vícecestný, uspořádaný strom, tak na obr.

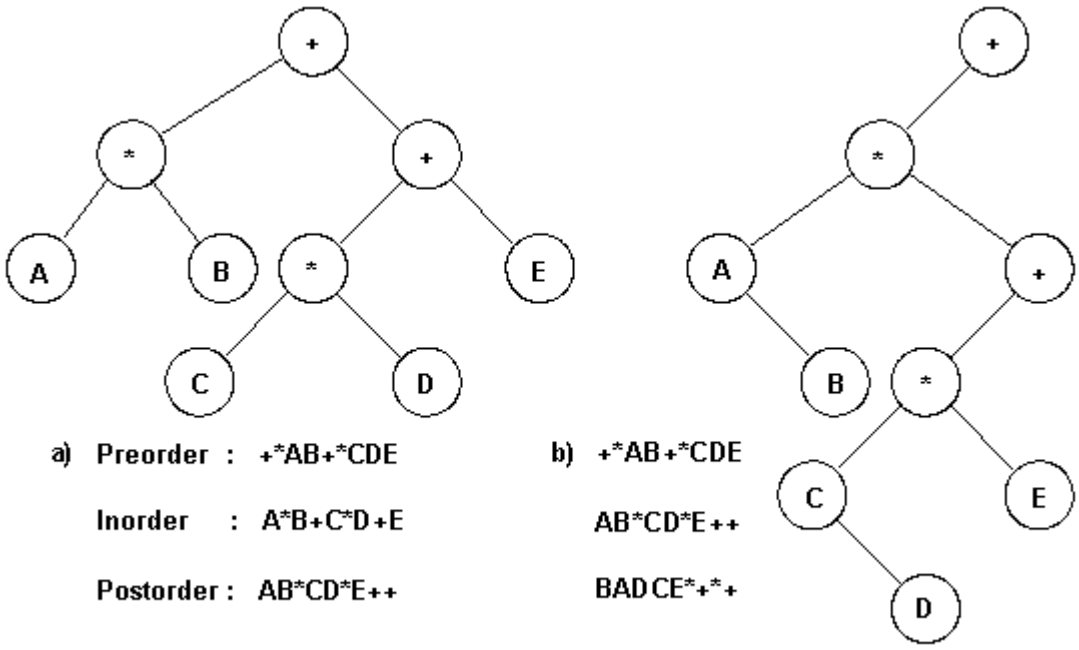


obr. 16

17b je odpovídající binární strom a metody průchodu preorder, inorder a postorder stromem dle obr. 17b vypadají tak, jak jsou na obr. 17b zapsány. Průchod stromem dle obr. 17b metodou inorder pak generuje postfixový zápis binárního výrazu reprezentovaného tímto stromem.

Uvažujme nyní, že je třeba vyhodnotit nějaký výraz, jehož všechny operandy jsou číselné hodnoty. Uzel pak může obsahovat informaci dvojího druhu. Buď je to číselná hodnota (jako operand), nebo znak (jako operátor). Použijeme typy :

```
typedef enum {
        OPERATOR, OPERAND
} TYPINFO ;
```



obr. 17

```
typedef union {
        char    operator ;
        double  operand ;
} TYP ;
typedef struct uzel  {
        TYPINFO info ;
        TYP     typ ;
        struct uzel  *nsyn, *bratr ;
} UZEL ;
```

Dále použijeme funkci

```
double  aplikuj ( UZEL *p ) ;
```

které se předá pointer na kořen stromu reprezentující výraz s jedním operátorem a operandy a funkce nám vrátí výsledek aplikace tohoto operátoru na operandy. Aby mohla být tato funkce použita, tak argument předávaný `p` musí referencovat kořen stromu, obsahující operátor a všechny jeho podstromy musí být nahrazeny uzly reprezentujícími číselný výsledek výpočtu v podstromech. Takže tak, jak je výraz vyhodnocován, musí se uvolňovat všechny uzly stromu reprezentující operandy a uzly reprezentující operátory jsou postupně konvertovány na uzly reprezentující operandy.

Tyto postupné náhrady provádí rekursivní funkce náhrada, které se předává pointer na kořen stromu, reprezentujícího celý výraz :

```
void nahrada ( UZEL *p ) {
        UZEL *q, *r ;
        double hodnota ;

        if ( p->info == OPERATOR ) {
                q = p->nsyn ;
                /* Nahraď všechny podstromy operandy */
                while ( q ) {
                        nahrada ( q ) ; q = q->bratr ;
                }
                /* Aplikace operátoru v kořeni na operandy v podstromu */
                hodnota = aplikuj ( p ) ;
                /* Náhrada operátoru výsledkem */
                p->info = OPERAND ; p->typ.operand = hodnota ;
                /* Uvolni všechny podstromy */
                q = p->nsyn ; p->nsyn = NULL ;
                while ( q ) {
                        r = q ; q = q->bratr ; free ( q ) ;
                }
        }
 }
```

Funkci pro vyhodnocení výrazu pak můžeme zapsat např. následovně :

```
double hodvyrazu ( UZEL *p ) {
        double h ;

        nahrada ( p ) ;
```

```
        h = p->typ.operand ;
        free ( p ) ;
        return ( h ) ;
}
```

Poznamenejme, že vyhodnocováním výrazu dochází k destrukci stromu !

Cvičení

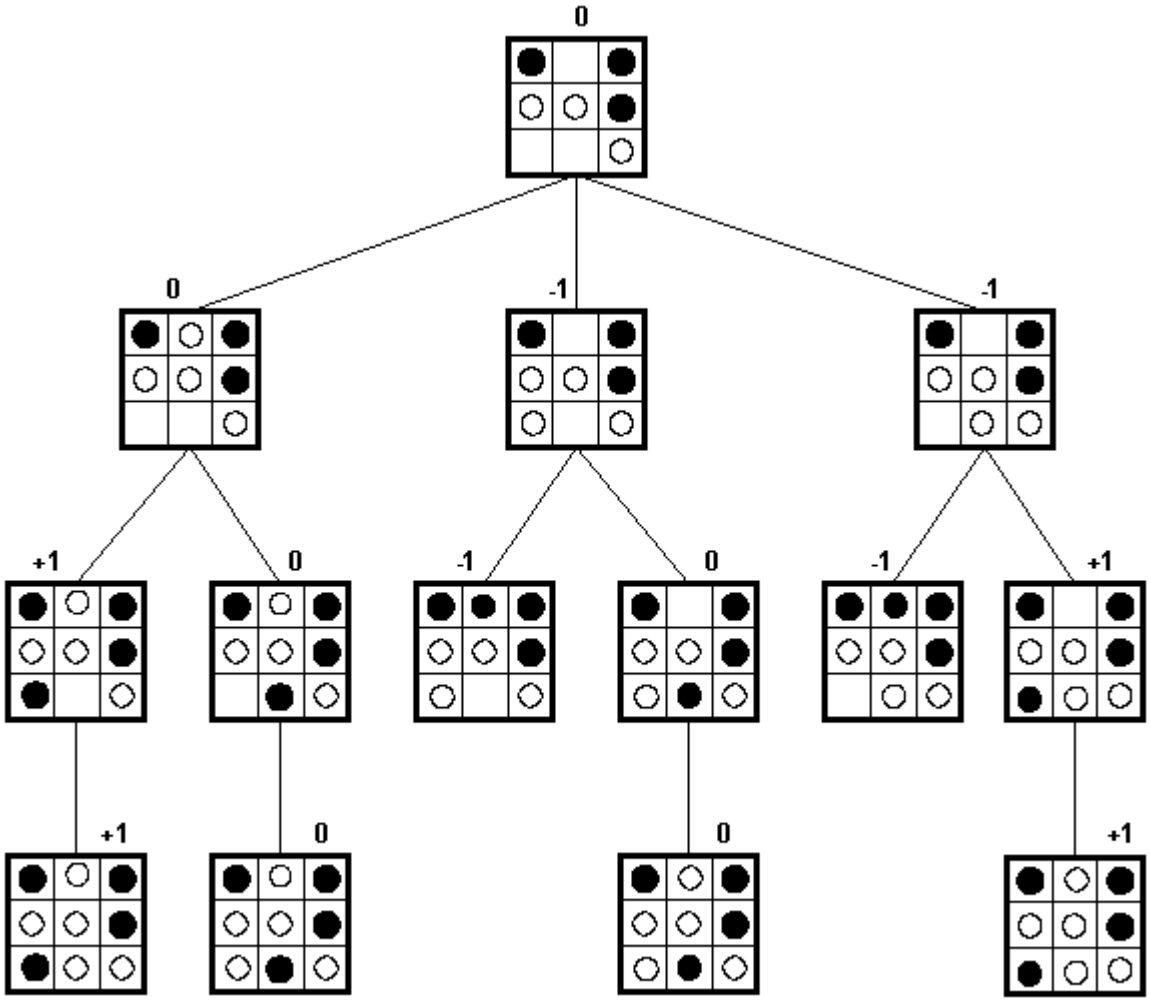
1. Napište funkci vlozsyny (vlož syny) , které se předává pointer p na nějaký uzel stromu a dále pointer na lineární seznam propojený členy bratr. Pokud uzel referencovaný p dosud nemá žádného syna, vloží funkce uzly seznamu jako syny uzlu stromu, referencovaného hodnotou p.

2. Napište funkci pripojsyna, které se předá pointer p na nějaký uzel stromu a informace typu TYPINFO. Funkce vloží nový uzel do stromu, přičemž tento uzel ponese předanou informaci a bude nejmladším synem uzlu, referencovaného hodnotou p.

3. Podle zvolených operátorů zapište funkci aplikuj a použijte ji pro vyhodnocování výrazů s číselnými operandy. Poté modifikujte v textu funkce pro vyhodnocování takových výrazů tak, aby nedocházelo k destrukci stromu.

2.5 Použití stromů pro strategické hry

Jinou významnou oblastí použití stromů jsou tzv. **strategické hry** (odtud někdy používané označení **hrací stromy**, **stromy stavů** apod.). Ilustrujme tuto problematiku na velmi jednoduché hře na čtvercové hrací desce, rozdělené do devíti polí a na které střídavě dva hráči pokládají své kameny s cílem obsadit svými kameny celý řádek, sloupec, nebo diagonálu hrací desky. Kterému hráči se to podaří dříve, ten vyhrává. V literatuře se tato hra nazývá tic-tac-toe, u nás je známa i jako "piškvorky".



obr. 18

Nechť je dána nějaká aktuální situace na hrací desce jako výsledek předcházejících 6-ti tahů, která je reprezentována kořenem stromu dle obr. 18. Uzly takového stromu reprezentují nějaké aktuální situace na hrací desce, které mohou nastat po některém z dalších tahů, přičemž přípustné tahy jsou vyjádřeny větvemi takového stromu. Daná aktuální situace odpovídá 0-té úrovni stromu, uzly na 1. úrovni odpovídají situacím vzniklým po tahu v našem příkladě "bílého" (a rovněž tak všechny další uzly na liché úrovni), zatímco uzly na 2. úrovni (a ev. na každé další sudé úrovni) situaci po tahu "černého". **Terminálové uzly** reprezentují situace, kdy hra končí vítězstvím, porážkou, nebo remisou (v našem příkladě jsou to všechny listy stromu dle obr. 18). Nyní je třeba vybrat pro aktuální situaci na hrací desce nejlepší tah pro hráče, který je v této aktuální situaci na tahu (v našem příkladě "bílý"), tzn. vybrat nějaký z uzlů na 1. úrovni. Za tím účelem použijeme tzv. **metodu minimax**, jejíž princip je zřejmý z obr. 18. Každému terminálovému uzlu přiřadíme např. hodnotu +1, když reprezentuje vítězství "bílého", 0 pro remisu a -1 pro porážku. Všem neterminálovým

uzlům na 2. úrovni přiřadíme hodnoty, které jsou maximem hodnot jejích synů, všem uzlům na 1. úrovni přiřadíme hodnoty, které jsou minimem hodnot jejich synů a na 1. úrovni vybereme uzel s maximální hodnotou. Nejlepší tah pro "bílého" reprezentuje nejstarší syn kořenu - pokud jej "bílý" učiní, nemůže prohrát. Pokud by "bílý" vybral jiný tah, mohl by dosáhnout remis, nebo dokonce i vyhrát, ovšem to za předpokladu, že odpověď "černého" by byla velmi stupidní.

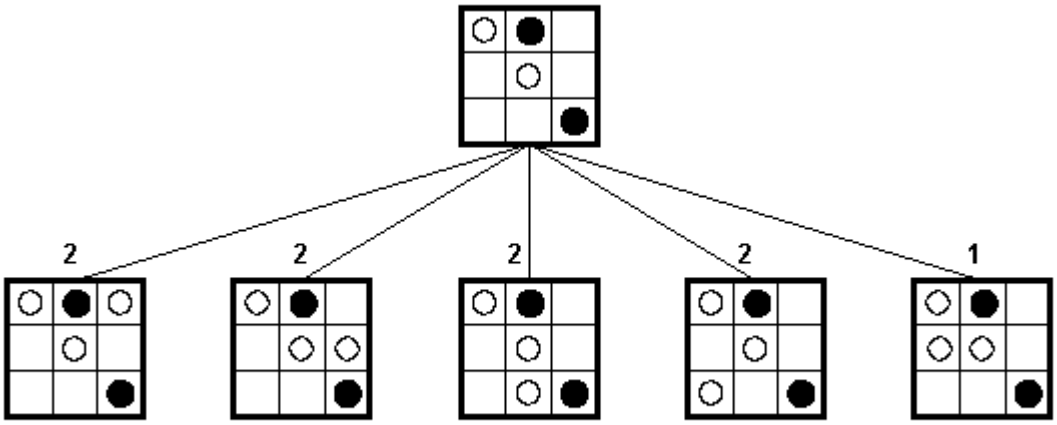
V praktických případech strategických her však hrací stromy mají daleko více rozvětvení, než v uvedeném příkladu. Tak např. u šachů může být počet možných tahů 30. To znamená, že když budeme chtít provést prognózu na 4 tahy dopředu, museli bychom provést analýzu $30^4 = 8.1 * 10^5$ uzlů.

Proto se dříve vysvětlená metoda minimax používá v oslabené formě :

- analýza nemusí začínat v terminálovém uzlu ,
- "málo perspektivní uzly" se neuvažují.

V terminálovém uzlu je výsledek hry známý, ovšem v neterminálovém uzlu lze získat pouze jeho přibližné hodnocení. Tak např. u šachů by takové hodnocení situace v neterminálovém uzlu mohlo být provedeno na základě počtu a hodnoty figur, schopnosti kontrolovat střed šachovnice, relativní manévrovací schopnosti figur apod. Pro hodnocení situace v neterminálovém uzlu použijeme funkci, která vrací nějakou číselnou hodnotu. Tato vrácená hodnota reprezentuje, jak výhodná se jeví situace v neterminálovém uzlu z hlediska jednoho z hráčů, při vítězné situaci vrací funkce největší možnou hodnotu, při situaci, která znamená porážku, vrátí funkce nejmenší možnou hodnotu. Výběr takové funkce má principiální význam. Rozeberme tuto skutečnost na jednoduchém příkladu hry tic-tac-toe.

U této hry by taková funkce mohla vracet počet řádků, sloupců a diagonál, které zůstávají otevřené pro hráče, mínus počet řádků, sloupců a diagonál, které zůstávají otevřené pro protihráče. Při situaci reprezentující vítězství hráče nechť tato funkce vrací 9, a při situaci reprezentující vítězství protihráče nechť tato funkce vrací hodnotu -9. V situaci podle obr. 19 je toto hodnocení napsáno u každého z uzlů na 1. úrovni. Vidíme, že první 4 situace jsou z hlediska takového hodnocení rovnocenné. Ovšem ve 4. situaci nenalezne "černý" již žádný tah, který by mohl zabránit vítězství "bílého" v jeho následujícím tahu, zatímco v prvních třech situacích může dosáhnout remis. Dokonce ani v 5. situaci, kde je hodnocení nejnižší, nenalezne "černý"



obr. 19

odpověď, která by mohla zabránit "bílému" ve vítězství. Řekneme však, že prognóza byla učiněna do nedostatečné **hloubky** (v našem příkladě do hloubky 1, tzn. na 1 tah dopředu). Kdybychom udělali takovou prognózu do hloubky 3, bude situace zřejmá (to je ponecháno na čtenáři).

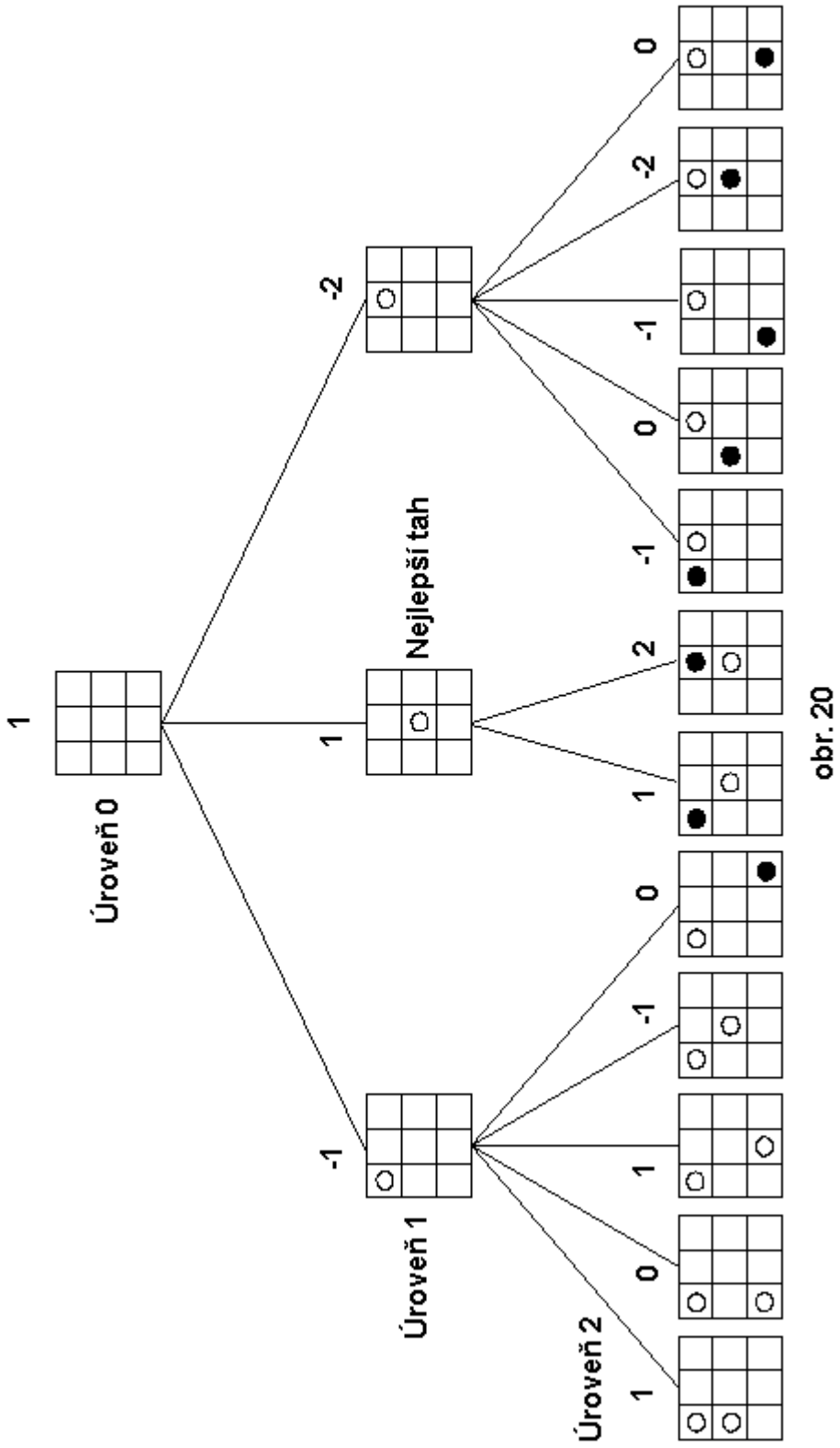
Na obr. 20 je schematicky znázorněna prognóza do hloubky 2 při zahájení hry. Všimněme si, že na 1. úrovni byly z devíti možných vybrány pouze 3 uzly. To proto, že z důvodů symetrie zbylých 6 uzlů nereprezentuje kvalitativně nové situace. Podobně je tomu též na 2. úrovni. Aplikací metody minimax na strom podle obr. 20 obdržíme nejlepší tah tak, jak je to vyznačeno na tomto obrázku.

Počet analyzovaných uzlů hracího stromu lze zmenšit použitím tzv. **alfa-beta algoritmu** (metoda alfa-beta minimax), jehož princip je následující :

1. Nechť A je nějaký uzel na takové úrovni hracího stromu, ve které se provádí minimalizace číselných hodnocení příslušejících uzlům na této úrovni a nechť B je otec A. Pokud číselné hodnocení v takovém uzlu A je menší nebo rovno parametru

alfa v uzlu B, nebudou se dále sledovat všechny podstromy, jejichž kořeny by byly mladšími bratry A. Přitom *alfa* je největší z číselných hodnocení, příslušejících starším bratrům uzlu B.

2. Nechť C je nějaký uzel na takové úrovni hracího stromu, ve které se provádí maximalizace číselných hodnocení příslušejících uzlům na této úrovni a nechť D je otec C. Pokud číselné hodnocení v takovém uzlu C je větší nebo rovno parametru *beta* v uzlu D, nebudou se dále sledovat všechny podstromy, jejichž kořeny by byly mladšími bratry C. Přitom *beta* je nejmenší z číselných hodnocení příslušejících starším bratrům D.



obr. 20

Hloubka prognózy při aplikaci alfa - beta algoritmu bude vždy větší než 1.

Pro implementaci uvedeného algoritmu v symbolickém jazyku si musíme nejprve zvolit vhodné typy podle toho, pro jakou strategickou hru je algoritmus použit. Tak např. pro jednoduchou hru tic-tac-toe můžeme zavést následující typy :

```
typedef enum  {
 /* Pole hrací desky může být obsazeno bílým, nebo černým  kamenem, nebo může
    být volné */
        BILY, CERNY, VOLNE
} POLE ;
typedef struct uzel {
        POLE deska[3][3] ;          /* Hrací deska pro tic-tac-toe */
        POLE hrac ;                 /* Hráč hrající bílými, nebo černými kameny */
        struct uzel *nsyn, *bratr ; /* Pointer na "nejstaršího syna" a na "bratra"  */
} UZEL ;
```

Tyto definice jsou obsahem souboru alfabeta.h, na který se uvedená implementace odvolává. Jedno z možných řešení (využívající přímé rekurse) je uvedeno v příloze 2, kde jednotlivé kroky jsou popsány přímo ve zdrojovém textu. Funkci uvedenou prototypem

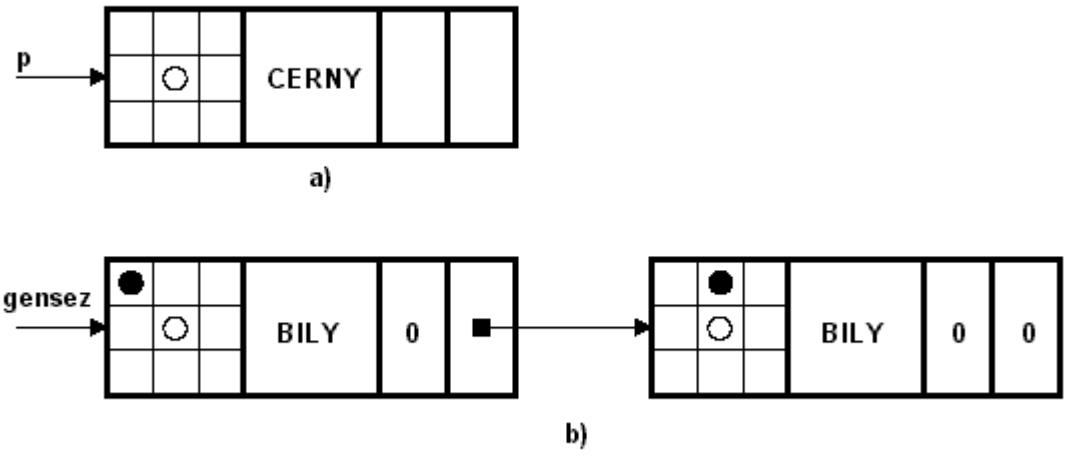
```
extern int hodnoceni ( UZEL *p, POLE  h ) ;
```

si doplní uživatel podle toho pro jakou strategickou hru algoritmus použije.

Funkce

```
extern UZEL *gensez ( UZEL *p ) ;
```

generuje lineární seznam prvků, které reprezentují situace odvozené z aktuální situace na hrací desce (parametr p) a vrací ukazatel na tento seznam. Tak např. v situaci podle obr. 21a má tato funkce efekt znázorněný obr. 21b (samozřejmě, pokud bereme v úvahu symetrická řešení - jinak by takových prvků v seznamu bylo 8). Všimněte si též funkce uvedené v příloze 2, kde je použito pro vyhodnocování následujícího obratu. Je-li $min\ (x, y)$ minimum z x a y a $max\ (x, y)$ maximum z x a y, tak $min\ (x, y) = - max\ (-x, -y)$.



obr. 21

Cvičení

1. Hra "zápalky" se hraje následovně. Na hrací desce leží n zápalek, které hráči střídavě odebírají. Přitom v každém tahu se musí odebrat k zápalek ,$1 \le k \le m$, přičemž m je celé a $m << n$. Prohraje ten hráč, který odebere poslední zápalku(y). Napište pro tuto hru svoje funkce gensez a hodnoceni a s použitím funkce v příloze 2 (kterou si můžete samozřejmě vytvořit sami, nebo podle svých potřeb modifikovat) řešte tuto hru. Pro menší hodnoty n lze hrací strom rozvětvit až k terminálovým uzlům a pak obě funkce jsou velmi jednoduché. Hrací deska je v tomto případě reprezentována členem struktury UZEL celočíselného typu, např. unsigned int deska .

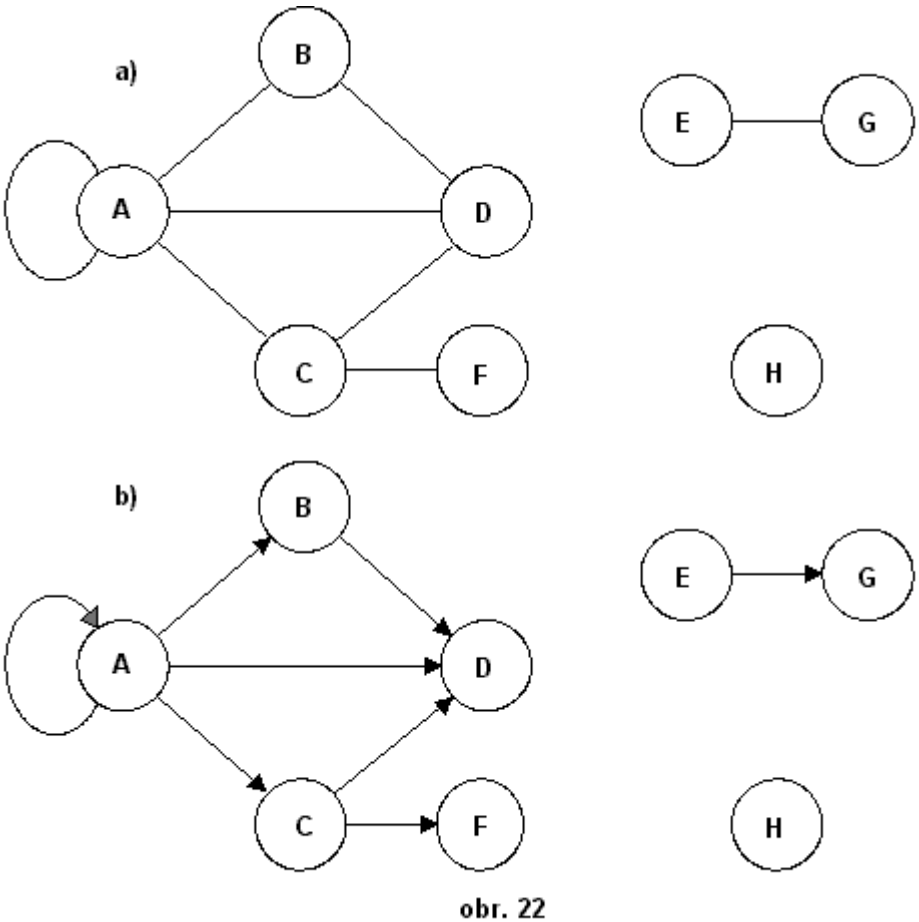
2. Zapište svoje funkce gensez a hodnoceni pro hru tic-tac-toe a použijte je pro řešení této hry.

3. Modifikujte funkci gensez pro nová pravidla této hry, podle kterých není hráčům dovoleno obsazovat zahajovacím tahem středové pole hrací desky.

3. GRAFY

3.1 Úvod ke grafům

Graf se skládá z množiny **uzlů** (vrcholů) a množiny **hran** (spojnic). Každá hrana grafu je specifikována dvojicí uzlů. V grafu podle obr. 22a je množina uzlů {A,B,C, D,E,F,G,H} a množina hran je {(A,B), (A,C), (A,D), (B,D), (C,D), (C,F), (E,G), (A,A)}. Jedná se o tzv. **neorientovaný graf**, na rozdíl od **orientovaných grafů**

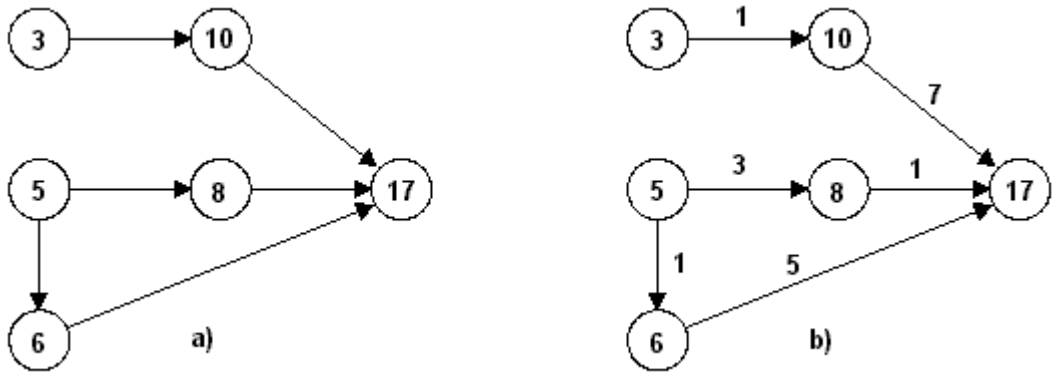


obr. 22

(angl. directed graph, digraph), kde je explicitně stanovený směr vztahů a graficky znázorněný šipkou. Na obr. 22b je orientovaný graf, kde množina hran je { <A,B>, <A,C>, <A,D>, <B,D>, <C,D>, <C,F>, <E,G>, <A,A> }. Dvojice uzlů je v lomených závorkách, které indikují, že tato dvojice je uspořádaná. Dále se omezíme na orientované grafy (digrafy).

Poznámka : Každý strom je též grafem, ovšem naopak to neplatí - ne každý graf je stromem. Ne všem uzlům musí příslušet nějaká hrana (viz uzel H na obr. 22).

Vstupní stupeň uzlu je počet hran vstupujících do daného uzlu - tento uzel je **konečným uzlem** hrany, která je vzhledem k tomuto uzlu **vstupní hranou**. Analogicky **výstupní stupeň uzlu** je počet výstupních hran tohoto, vzhledem k těmto hranám **počátečního uzlu**. Vstupní a výstupní stupně dohromady se označují jako **stupně uzlu**. Tak např. uzel A dle obr. 22b má 1 vstupní a 4 výstupní stupně, celkem tedy 5 stupňů. Uzel *n* je **přilehlý** (adjacent) k uzlu *m*, pokud je na hraně z *m* do *n*.



obr. 23

Relace ***R*** na množině ***A*** je množina uspořádaných dvojic prvků z ***A***. Když <*x*,*y*> je člen relace ***R***, tak říkáme, že *x* je vztaženo k *y* v ***R***. Např. pro ***A*** = { 3, 5, 6, 8, 10, 17 } je ***R*** = { <3,10>, <5,6>, <5,8>, <6,17>, <8,17>, <10,17> } relací na ***A***. Tato relace může být popsána tak, že *x* je menší než *y* a současně zbytek po celočíselném dělení *y*/*x* je lichý. Tato relace může být ilustrována digrafem podle obr. 23a. Přiřadíme-li každé hraně číslo, které je zbytkem po celočíselném dělení čísel v počátečním a konečném uzlu této hrany, tak dostaneme graf dle obr. 23b. Čísla přiřazená hranám se nazývají **váhy** a grafy se pak obvykle nazývají **sítěmi** (**ohodnocenými grafy**).

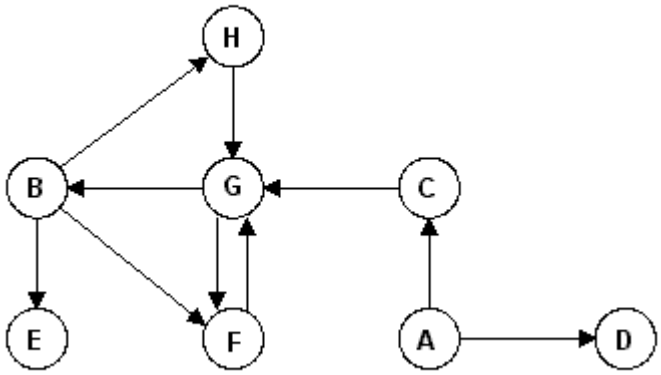
Definujme nyní primitvní operace nad grafy :

1. join (a , b) - přidání hrany, jejíž počáteční uzel je *a* a konečný *b*, pokud taková hrana dosud neexistuje,

join (a , b , x)- přidání hrany s váhou *x*, jejíž počáteční uzel je *a* a konečný *b*.

2. remv (a, b) , resp remvwt (a , b, x) - odebrání (remove) hrany s počátečním uzlem *a* a konečným *b*, pokud ovšem taková hrana existuje (resp. s váhou *x* u remvwt).

3. adjacent (a , b) - predikát, který vrací *true* (hodnotu různou od 0), když uzel *a* je počátečním a *b* konečným uzlem nějaké hrany (lépe *b* přiléhá k *a*). Jinak vrací hodnotu *false* (0).



obr. 24

Cesta délky *k* od uzlu *a* do uzlu *b* je definována jako posloupnost k+1 uzlů $n_1, n_2, \ldots\ldots, n_{k+1}$ takových, že $n_1 = a$, $n_{k+1} = b$ a adjacent (n_i , n_{i+1}) je *true* pro $i = 1, 2, \ldots, k$. Cesta od nějakého uzlu do toho samého uzlu se nazývá **cyklus**. Grafy obsahující cyklus se nazývají **cyklické**, v opačném případě **acyklické**. Uvažujme např. graf dle obr. 24. Zde existuje jedna cesta délky 1 od A do C, dvě cesty délky délky 2 od B do G a jedna cesta délky 3 od A do F. Ale od B k C neexistuje cesta. Z obr. 24 jsou patrné i cykly např. od F do F, od H do H atd.

Jako jednoduchý příklad uveďme rekursivní algoritmus pro nalezení cesty délky *k* od uzlu *a* do uzlu *b*. Je-li $k = 1$ tak výsledek vrací přímo funkce adjacent (*a*,*b*). Jinak budeme hledat cestu délky $k - 1$ od nějakého uzlu *c* do *b*, pokud adjacent (*a*,*c*) vrací *true*. V jazyku C můžeme zapsat odpovídající funkci např. takto (pro zjednodušení jsou uzly očíslovány, n uzlů označených 0, 1, ... , n-1) :

```
int  pocet_uzlu ;
int cesta ( int k, int a, int b )  {
        int c, p = 0 ;
        if ( k == 1 )  return ( adjacent ( a, b ) ) ;
        for ( c = 0 ; c < pocet_uzlu && !p ; c++ )
                if ( adjacent(a,c) ) p = cesta( k-1,c,b) ;
```

```
        return ( p ) ;
    }
```

Takový algoritmus má však řadu nedostatků. Především rekursivní proces vyžaduje mnoho času na prošetření cest. Kromě toho takový algoritmus podává pouze zprávu, zda hledaná cesta existuje, či nikoliv, neprodukuje tuto cestu jako takovou. V případě existence takové cesty je užitečné znát i odpovídající hrany grafu, po kterých se cesta realizuje.

Předpokládejme nyní, že digraf není ohodnocený a uzlům grafu není přiřazena žádná informace. Takový digraf lze úplně popsat **maticí přiléhavosti (sousednosti, adjacence)** ***P*** typu (*n,n*), jejíž prvky P_{ij} udávají, zda uzel *j* přiléhá k uzlu *i* či nikoliv (*n* je počet uzlů). Jestliže tedy existuje hrana od uzlu *i* do uzlu *j*, tak $P_{ij} = 1$ (*true*), v opačném případě 0 (*false*). Uvažujme nyní výraz

$$P_{ik} \wedge P_{kj} \quad . \qquad (1)$$

Hodnota tohoto výrazu (logický součin) je 1 (*true*) tehdy a jen tehdy, když P_{ik} a P_{kj} mají současně hodnotu 1 (*true*) - jinými slovy, když v grafu existuje cesta délky 2 od uzlu *i* do uzlu *j* přes uzel *k*. Nyní uvažujme výraz

$$(P_{i1} \wedge P_{1j}) \vee (P_{i2} \wedge P_{2j}) \vee \ldots\ldots\ldots\ldots \vee (P_{in} \wedge P_{nj}) \quad . \qquad (2)$$

Symbol V značí logický součet a hodnota tohoto výrazu je 1 (*true*) tehdy a jen tehdy, když v grafu existuje nějaká cesta délky 2 od uzlu *i* do uzlu *j*. Uvažujme nyní nějakou matici $_2\boldsymbol{P}$, jejíž prvky $_2P_{ij}$ jsou dány výrazem (2). Tato matice se nazývá matice cest délky 2 a její prvky indikují, zda v grafu existuje cesta délky 2 mezi dvěma uzly. Tato matice je tedy logickým součinem $\boldsymbol{P} \bullet \boldsymbol{P}$ (symbol ● znamená tedy logický součin matic ***P*** - v logickém součinu jsou aritmetické operace nahrazeny logickými, sčítání logickým součtem, násobení logickým součinem). Analogicky matice cest délky 3 bude

$$_3\boldsymbol{P} = \boldsymbol{P} \bullet {}_2\boldsymbol{P} \qquad (3)$$

a její prvky indikují, zda v grafu existuje cesta délky 3 od *i* do *j*, či nikoliv. Obecně matice cest délky *k* bude

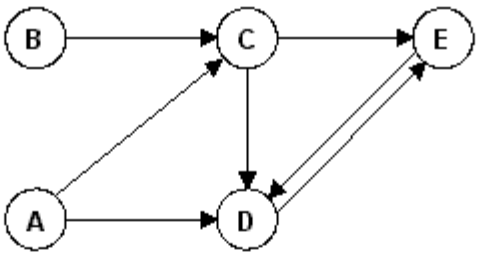
$$_k\boldsymbol{P} = \boldsymbol{P} \bullet {}_{k-1}\boldsymbol{P} \quad . \qquad (4)$$

Příklad je na obr. 25.

Uvažujme nyní výraz

$$P_{ij} \vee {}_2P_{ij} \vee {}_3P_{ij} \quad . \qquad (5)$$

Jeho hodnota bude 1 (*true*), pokud v grafu existuje cesta délky 3 nebo kratší od uzlu *i* do uzlu *j*. Vytvořme nyní matici ***S***, jejíž prvky S_{ij} budou mít hodnotu 1 (*true*)



P	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

$_2P$	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

$_3P$	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

$_4P$	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

Transitivní uzávěr *P* ⟶

S	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

obr. 25

tehdy a jen tehdy, když v grafu existuje nějaká cesta od uzlu *i* do uzlu *j* (cesta nějaké délky). Pokud graf má *n* uzlů, tak

$$S_{ij} = P_{ij} \vee {}_2P_{ij} \vee \ldots\ldots\ldots \vee {}_nP_{ij} \quad . \qquad (6)$$

Existuje-li totiž v grafu cesta délky $m > n$ od *i* do *j*, musí existovat cesta od *i* do *j* délky kratší nebo rovné *n*. Jistě, pokud graf obsahuje pouze *n* uzlů a cesta je délky $m > n$, tak musí procházet vícekrát nějakým uzlem *k* - stačí tedy odebrat z grafu cykly. Matice ***S*** se často označuje jako **transitivní uzávěr matice *P*** - viz příklad na obr. 25.

Pro výpočet transitivního uzávěru se používá tzv. **Warshallův algoritmus**. Definujme matici ${}_k\boldsymbol{S}$ takovou, že ${}_kS_{ij} = 1$ (*true*) tehdy a jen tehdy, když existuje cesta od *i* do *j,* neprocházející žádným uzlem s číslem větším než *k* (pochopitelně s vyjímkou *i* a *j*). Pokusme se nyní odvodit ${}_{k+1}S_{ij}$ z matice ${}_k\boldsymbol{S}$. Pokud ${}_kS_{ij} = 1$, tak zřejmě ${}_{k+1}S_{ij} = 1$. Pokud ovšem ${}_kS_{ij} = 0$, tak ${}_{k+1}S_{ij}$ může být 1, jen pokud existuje cesta od *i* do $k+1$ procházející dále pouze uzly 1 až *k* a podobně cesta od $k+1$ do *j*. Tedy ${}_{k+1}S_{ij} = 1$ tehdy a jen tehdy, když je splněna jedna z podmínek :

$${}_kS_{ij} = 1 \quad , \qquad (7)$$

$${}_kS_{i,\,k+1} = 1 \;\Lambda\; {}_kS_{k+1,\,j} = 1 \quad . \qquad (8)$$

Označíme ${}_0\boldsymbol{S} = \boldsymbol{P}$ (přímé cesty od *i* do *j*) a dle výše uvedeného budeme ${}_k\boldsymbol{S}$ počítat z ${}_{k-1}\boldsymbol{S}$, ovšem místo (pseudo)příkazu

```
for ( i=0; i<n; i++ ) for ( j=0; j<n; j++ )
        kS[i][j] = k-1S[i][j]  ||  k-1S[i][k]  &&  k-1S[k][j]
```

použijeme efektivnější ve tvaru

```
kS  =  k-1S  ;
for ( i=0; i<n; i++ )
  if ( k-1S[i][k] ) for ( j=0; j<n; j++ ) kS[i][j] = k-1S[i][j]  ||  k-1S[k][j]
```

Pak funkci pro výpočet transitivního uzávěru lze zapsat i takto :

```
int *prileha,                    /* Matice přiléhavosti P  */
    *tran_uzaver ;               /*  Transitivní uzávěr S  */
int  n ;                         /*  Počet uzlů  */
```

```
void transuzaver ( void ) {
    int i, j, k ;

    tran_uzaver=(int *)malloc(n*n*sizeof(int));
    for ( i=0; i<n; i++ ) for ( j=0; j<n; j++ )
       tran_uzaver[i*n+j] = prileha[i*n+j] ;
    for ( k=0; k<n; k++ ) for ( i=0; i<n; i++ )
       if ( tran_uzaver[i*n+k] )  for ( j=0; j<n; j++ )
             tran_uzaver[i*n+j]  = tran_uzaver[i*n+j] || tran_uzaver[k*n+j] ;
}
```

Cvičení

1. Na množině $\boldsymbol{A}$ = { 2, 3, 4, 5, 6, 7, 8 } je definována relace $\boldsymbol{R}$ tak, že x je vztaženo k y v $\boldsymbol{R}$, když zbytek po celočíselném dělení x/y je roven 2. Zobrazte $\boldsymbol{R}$ digrafem.

2. Napište funkci v jazyku C, která pro graf daný maticí přiléhavosti a pro dva dané uzly u_1 a u_2 určí, zda v grafu existuje nějaká cesta od uzlu u_1 do uzlu u_2.

3. Napište funkce, které pro graf daný maticí přiléhavosti a pro dva dané uzly u_1 a u_2 určí :

   a) **počet** cest dané délky k od uzlu u_1 do uzlu u_2 ,

   b) celkový počet cest od uzlu u_1 do uzlu u_2 .

4. Pro acyklické grafy lze uzly přečíslovat tak, že odpovídající matice přiléhavosti bude spodní trojúhelníková matice. Napište funkci, která pro danou matici přiléhavosti vrátí novou matici přiléhavosti jako spodní trojúhelníkovou matici (pokud je to možné) s přečíslováním uzlů.

5. **Pravděpodobnostně orientovaný digraf** je takový graf, kde každé hraně je přiřazena pravděpodobnost tak, že součet všech pravděpodobností přisouzených všem hranám vycházejícím z jednoho a téhož uzlu je roven 1.

Uvažujme tedy pravděpodobnostní digraf, který reprezentuje "tunelový systém". Dále uvažujme, že tento digraf je **acyklický** a že osoba nacházející se v nějakém uzlu tohoto systému tunelů si s danou pravděpodobností volí hranu pro pohyb k jinému

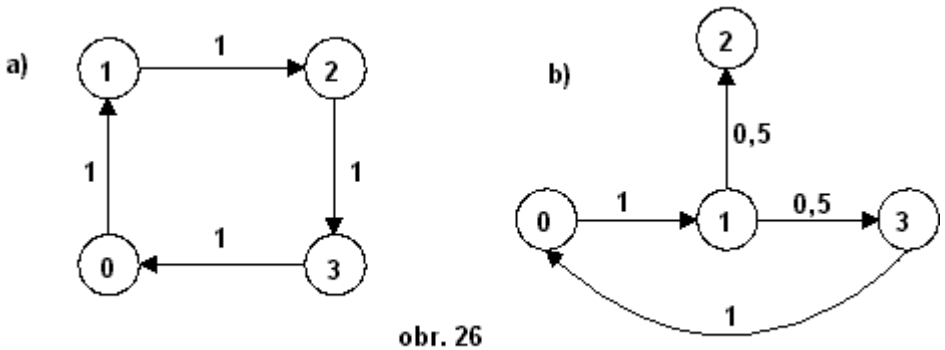
uzlu. Napište funkci pro určení pravděpodobnosti, že osoba projde každým uzlem grafu. Zvažte, co se stane, když graf je cyklický.

Návod

Předpokládáme, že acyklický graf obsahuje n uzlů a osoba se nachází v uzlu i. Pokud při pohybu má projít všemi uzly acyklického grafu, musí vykonat cestu (pokud taková existuje)

$$i\ i_2\ i_3\ \ldots\ldots\ldots\ldots\ i_n\ ,$$

tedy cestu délky n - 1, na které se žádný z uzlů nemůže vyskytovat více než jednou. Taková cesta v acyklickém grafu může být jen jedna.



obr. 26

Nechť $\boldsymbol{W}$ je matice pravděpodobností, jejíž prvky W_{ij} udávají pravděpodobnost, že osoba nacházející se v uzlu i se bude pohybovat do uzlu j. Platí :

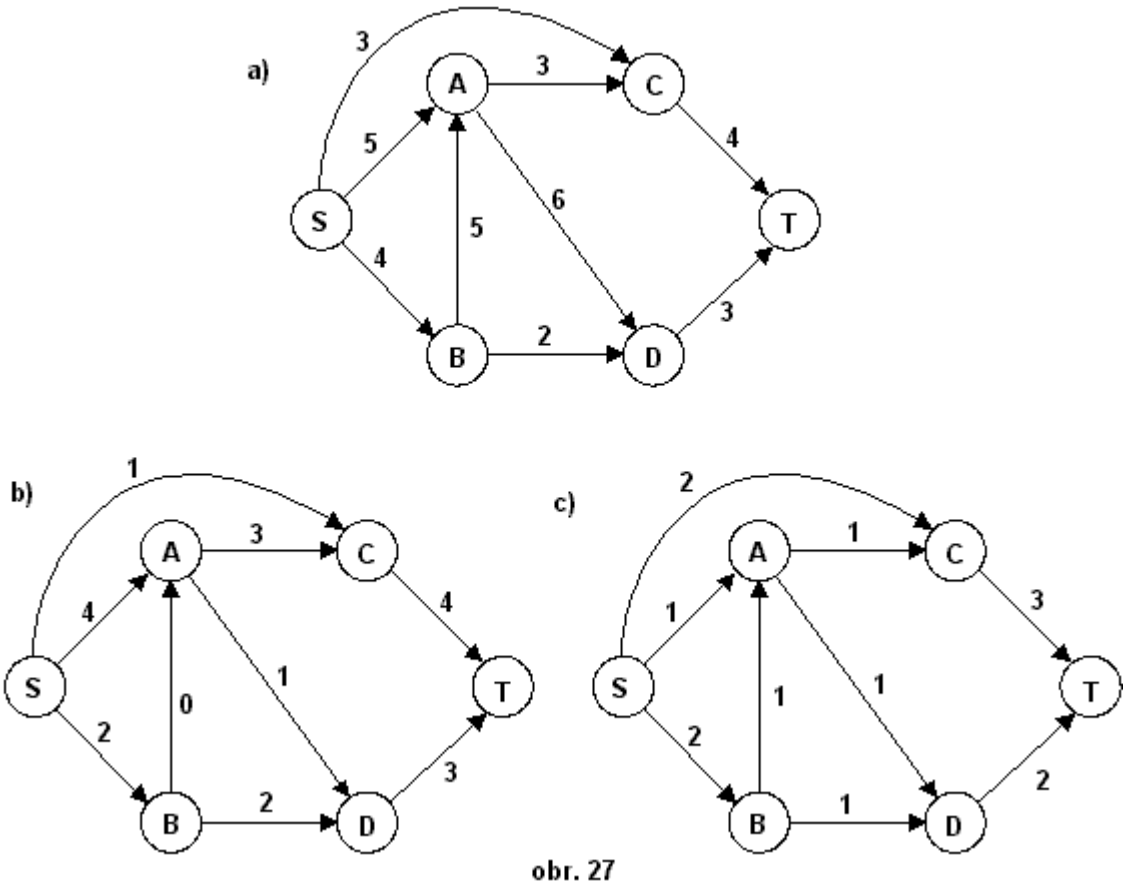
Přitom W_{i*} vlastně udává pravděpodobnost, že osoba nacházející se v uzlu i vykoná při pohybu cestu délky 1. Uvažujme dále $_2\boldsymbol{W} = \boldsymbol{W}\,\boldsymbol{W}$, ……..., $_{n-1}\boldsymbol{W} = {}_{n-2}\boldsymbol{W}\ \boldsymbol{W}$ a hledaná pravděpodobnost bude

$$\sum_{k=1}^{n} {}_{n-1}W_{ik}\ , resp.\ 0\ .$$

To může platit i v případě, kdy graf je cyklický (viz obr. 26a), ale také nemusí (viz obr. 26b) !!

3.2 Aplikace grafů - tok sítí

Uvažujme vodovodní síť schematicky reprezentovanou grafem dle obr. 27. V tomto ohodnoceném grafu každá hrana znázorňuje vodovodní potrubí a její váha udává kapacitu potrubí v objemových jednotkách za časovou jednotku (tedy maximální možný průtok potrubím). Uzly grafu reprezentují body, ve kterých je potrubí spojeno a



obr. 27

voda je v nich z přívodních potrubí rozdělována do odvodních potrubí. Dva uzly S a T jsou zde označeny jako **zdroj vody** a **spotřebič vody**. Vodovodním potrubím je voda od zdroje S dodávána spotřebiteli T. Voda může každým potrubím protékat pouze jedním směrem (což může být zajištěno zpětnými ventily) a žádné potrubí sítě nepřivádí vodu do S, nebo ji neodvádí z T.

Budeme chtít maximalizovat množství vody proudící od S do T. Budeme přitom uvažovat neomezené možnosti zdroje i spotřebiče, takže jediným limitujícím faktorem budou kapacity potrubí vodovodní sítě. Analogické problémy lze formulovat u elektrické, železniční a jiné distribuční sítě.

Definujme nejprve **kapacitní funkci** $c(a,b)$, kde a, b jsou uzly. Když funkce adjacent(a,b) má hodnotu 1 (*true*), tak $c(a,b)$ je dáno kapacitou potrubí od a do b, jinak bude $c(a,b) = 0$. Dále definujme **průtokovou funkci** $f(a,b)$, kde a, b jsou též uzly. Pokud adjacent(a,b) má hodnotu 1 (*true*), tak hodnotou funkce $f(a,b)$ je množství vody proudící od a do b, jinak $f(a,b) = 0$. Pochopitelně pro všechny uzly a i b bude $0 \le f(a,b) \le c(a,b)$. Nechť dále v je množství vody proudící od S do T. Pochopitelně, že množství vody odtékající od S, proudící vodovodní sítí a přitékající do T musí být stejné, tj. přítok do všech uzlů od S musí být roven přítoku od všech uzlů do T :

$$\sum_{x \in \Omega} f(S,x) \quad \sum_{x \in \Omega} f(x,T) = v \,, \tag{9}$$

kde Ω je množina všech uzlů grafu.

Žádný uzel kromě S nemůže vodu produkovat (jen rozvádět) a žádný uzel kromě T nemůže vodu spotřebovávat, tj. množství vody přitékající do uzlu (mimo uzly S, T) a odtékající z tohoto uzlu musí být stejné :

$$\sum_{y \in \Omega} f(x,y) = \sum_{y \in \Omega} f(y,x), \; \textit{pro všechny uzly } x \neq S,T. \tag{10}$$

Definujeme-li přítokovou funkci $p(x)$ jako množství vody přitékající do uzlu x a odtokovou funkci $o(x)$ jako množství vody odtékající z uzlu x, tak můžeme výše uvedené podmínky přepsat do následujícího tvaru :

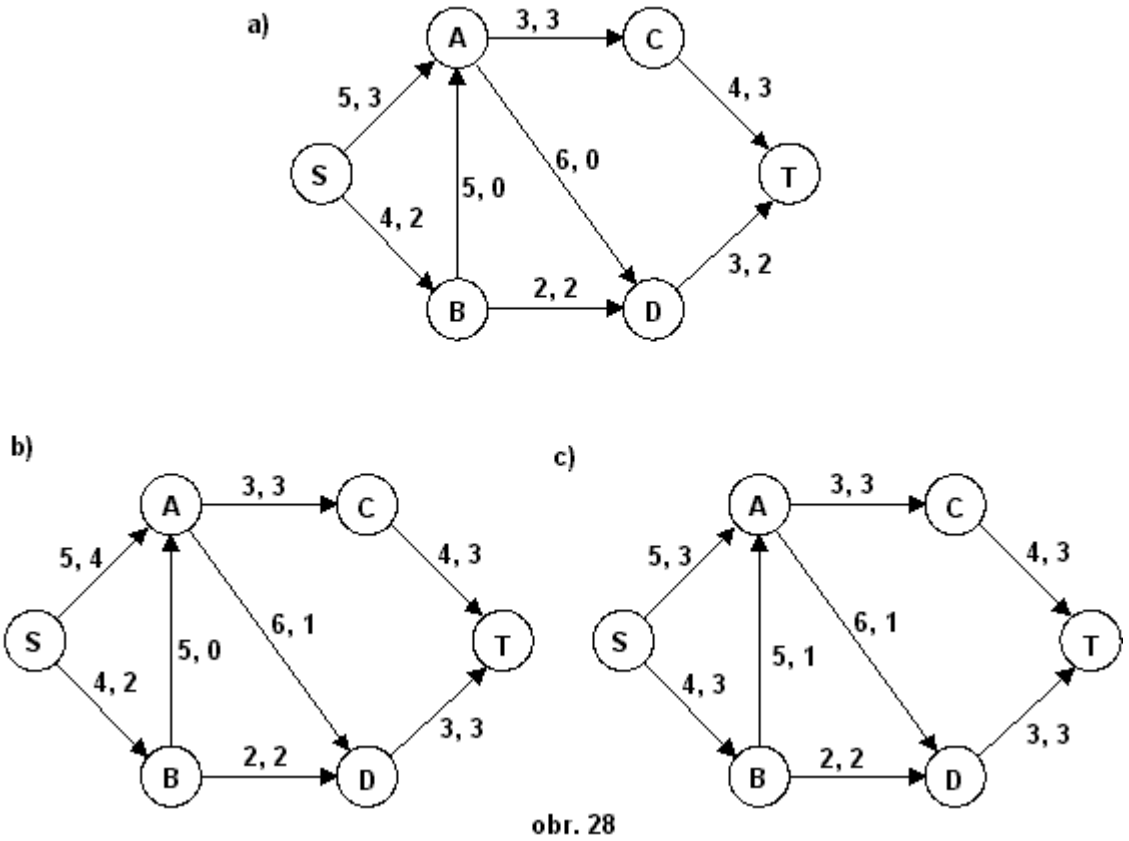
$$o(S) = p(T) = v \,, \tag{11}$$

$$p(x) = o(x), \; \textit{pro všechny uzly } x \neq S,T \,. \tag{12}$$

Pro graf a kapacitní funkci dle obr. 27a lze najít různé průtokové funkce - dvě takové možnosti ilustrují obr. 27b a obr. 27c. Obě průtokové funkce splňují dříve uvedené podmínky.

Úkolem je však najít takovou průtokovou funkci, kde hodnota v je maximální. Pochopitelně, že průtoková funkce dle obr. 27b je lepší ($v = 7$), než průtoková funkce dle obr. 27c ($v = 5$). Přípustná, resp. možná průtoková funkce je i taková, kde pro všechna a, b je $f(a,b) = 0$. Taková průtoková funkce však nemůže splňovat naše požadavky, neboť od S do T neproudí žádná voda. Je-li však dána nějaká průtoková funkce, tak každý přírůstek průtoku od S ku T představuje vylepšení průtokové funkce (samozřejmě jen za předpokladu, že tato vylepšená verze odpovídá dříve uvedeným podmínkám). Přesněji, každému přírůstku nebo úbytku přítoku do nějakého uzlu

(kromě S, nebo T), musí odpovídat přírůstek nebo úbytek odtoku z tohoto uzlu. Strategie určení maximální průtokové funkce spočívá v počáteční volbě nulového průtoku a v postupném vylepšování průtokové funkce, až dosáhneme žádané řešení.



obr. 28

Dále budeme sledovat dvě cesty pro vylepšení průtokové funkce. První způsob spočívá v nalezení cesty S = x_1 , x_2 , ... , x_n = T od uzlu S k uzlu T takové, že průtok každou hranou této cesty je menší než její kapacita, tj.

$$f(x_{k-1}, x_k) < c(x_{k-1}, x_k),\ k = 2,3,.....,n\,. \qquad (13)$$

Průtok každou hranou této cesty lze pak zvýšit o hodnotu

$$\min_{k=2,3,...,n} \left[\, c(x_{k-1}, x_k) - f(x_{k-1}, x_k)\right]. \qquad (14)$$

Zvýšením průtoku o tuto hodnotu dostaneme na této cestě alespoň jednu hranu kde

$$f(x_{k-1}, x_k) = c(x_{k-1}, x_k) \qquad (15)$$

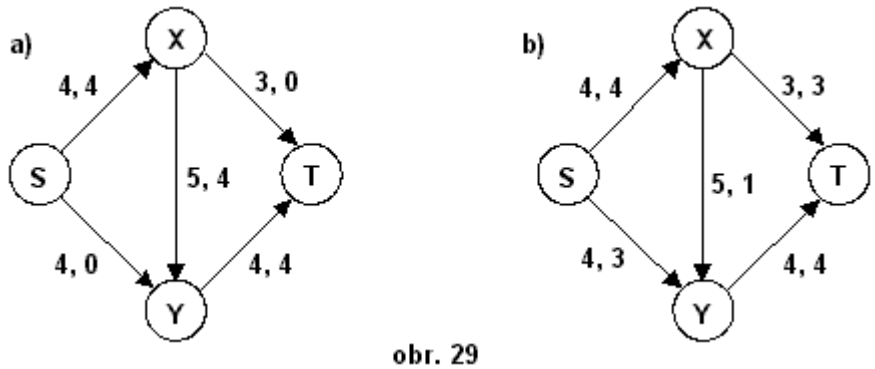
a průtok na této cestě tudíž nelze dále zvyšovat. To je na příkladě demonstrováno obr.28, kde čísla uvedená u jednotlivých hran znamenají kapacitu a aktuální průtok. Na tomto obrázku jsou patrné dvě cesty od S k T, kde existuje pozitivní průtok - (S, A, C,

T) a (S, B, D, T). Ovšem obě cesty obsahují hranu (<A,C> a <B,D>), kde průtok je roven kapacitě a tudíž nelze vylepšit. Existuje však i jiná cesta (S, A, D, T), kde kapacita každé hrany je větší než aktuální průtok. Průtok podél této cesty lze zvýšit o 1, neboť na hraně <D,T> je kapacita 3. Výsledkem bude průtoková funkce podle obr. 28b, průtok od S k T se zvýšil z 5 na 6. V grafu podle obr. 28b již neexistuje žádná cesta od S k T, kde by bylo možné vylepšit průtok. Ale v situaci podle obr. 28a lze místo cesty (S, A, D, T) zvolit cestu (S, B, A, D, T), podél které lze též zvýšit průtok - dostaneme pak průtokovou funkci podle obr. 28c, která není lepší, ale ani horší než průtoková funkce podle obr. 28b.

Pokud v grafu již neexistuje žádná cesta, podél které lze průtok zvýšit, tak to ještě neznamená, že řešení bylo dosaženo. Pro ev. zvýšení celkového průtoku od S k T lze použít jiné metody, jejíž princip lze vysvětlit na příkladě podle obr. 29. V grafu podle obr. 29 neexistuje žádná cesta, podél které by bylo možné zvýšit průtok. Když ale snížíme průtok od X do Y, tak můžeme zvýšit průtok od X do T. Snížení přítoku do Y kompenzuje zvýšení průtoku od S do Y, čímž zvýšíme celkový průtok od S k T - z hodnoty 4 na hodnotu 7 (viz obr. 29b).

Zevšeobecnění této druhé metody ilustrované příkladem dle obr. 29 lze provést následovně. Předpokládejme, že v grafu existují cesty od S do nějakého uzlu *y,* od nějakého uzlu *x* do T a od *x* do *y* s pozitivním průtokem. Pak průtok od *x* do *y* lze snížit a o stejné množství lze zvýšit průtok od *x* do T a od S do *y*. Toto množství je minimum z hodnot :

- průtok od *x* do *y* ,
- rozdíl mezi kapacitou a průtokem od S do *y* ,
- rozdíl mezi kapacitou a průtokem od *x* do T.



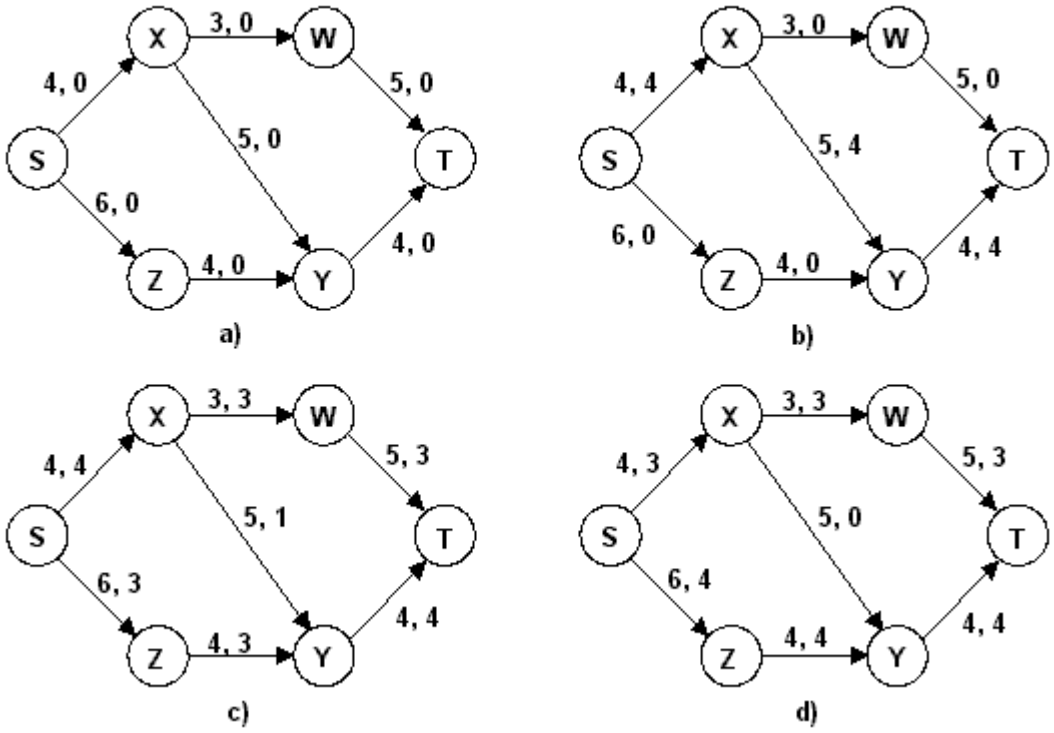
obr. 29

Obě metody mohou být kombinovány následovně. Množství vody proudící od S k T lze zvýšit jen v případě, když to dovoluje kapacita potrubí. Předpokládejme, že kapacita potrubí od S do *x* dovoluje zvýšit množství vody přitékající do *x* o hodnotu *h*. Lze-li s ohledem na kapacitu potrubí od *x* do T toto zvýšené množství vody přenést od *x* do T, tak toto zlepšení průtoku lze realizovat. Takže když nějaký uzel *y* přiléhá k uzlu *x* (tj. existuje hrana <*x*,*y*>), tak množství vody proudící od *y* do T lze zvýšit o minimum z hodnot *h* a nevyužité kapacity na hraně <*x*,*y*>. To je aplikace první metody. Podobně když *x* přiléhá nějakému uzlu *y* (tj. existuje hrana <*y*,*x*>), tak množství vody proudící od *y* do T lze zvýšit o minimum z hodnot *h* a aktuálního průtoku od *y* do *x*. Toho lze dosáhnout snížením průtoku od *y* do *x* dle druhé metody. Procházíme-li tímto způsobem grafem od S do T, tak lze určit, o jaké množství lze zvýšit průtok do T.

Definujme nejprve tzv. **semicestu** od S do T jako takovou sekvenci uzlů S = $x_1, x_2, \ldots, x_n$ = T, že pro všechna $1 < i \leq n$ je buď $<x_{i-1},x_i>$, nebo $<x_i,x_{i-1}>$ hranou ohodnoceného grafu. Užitím výše popsané techniky lze uvést algoritmus určení takové semicesty od S do T, že tok do každého uzlu této semicesty lze zvýšit. Přitom se užije již určené částečné semicesty od S. Když poslední uzel ve zjištěné částečné semicestě je *a*, tak dle algoritmu se uvažuje rozšíření do nějakého takového uzlu *b*, že buď <*a*,*b*>, nebo <*b*,*a*> je hrana grafu. Částečnou semicestu pak prodloužíme do uzlu *b*, když toto prodloužení vede ke zvýšení přítoku do *b*. Jakmile byla semicesta prodloužena do uzlu *b*, tak se tento uzel nebere dále v úvahu pro rozšíření nějaké jiné částečné semicesty (protože od tohoto místa se budeme snažit určit jednotlivou semicestu od S k T). Algoritmus zachycuje o jaké množství lze zvýšit přítok do *b* a zda tento přítok lze uvažovat na hraně <*a*,*b*>, nebo <*b*,*a*>.

Tento proces se opakuje, až částečná semicesta je prodloužena až do T. Dle algoritmu se pak postupuje podél této semicesty zpět až k S a vyrovnává se celkový průtok. Celý proces se pak opakuje za účelem nalezení nějaké jiné semicesty od S do T. Pokud již nelze žádnou semicestu úspěšně prodloužit, tak průtok nelze zvýšit a aktuální průtok je maximální.

Ilustrujme tento postup na příkladě podle obr. 30. Zpočátku uvažujeme nulový průtok, takže dostaneme situaci podle obr. 30a. Čísla přidružená každé hraně jsou kapacita a aktuální průtok. Semicestu od S prodloužíme na (S,X) nebo (S,Z). Průtok od S k X lze zvýšit o 4, průtok od S do Z lze zvýšit o 6. Semicestu (S,X) lze



obr. 30

prodloužit na (S,X,W) a (S,X,Y) s odpovídajícími přírůstky průtoku do W o 3 a do Y o 4. Semicestu (S,X,Y) lze prodloužit na (S,X,Y,T) - přírůstek toku do T bude 4. Dosáhli jsme uzlu T semicestou (S,X,Y,T) s čistým přírůstkem 4 a o toto množství zvýšíme průtok každou "dopřednou" hranou semicesty. Výsledek je na obr. 31b.

Poznámka : Bylo by možné volit i prodloužení (S,X,W) na (S,X,W,T). Samozřejmě by bylo možné provést rozšíření (S,Z) na (S,Z,Y) dříve než (S,X) na (S,X,W) a (S,X,Y). Takové volby jsou libovolné.

Opakujme výše popsaný proces s grafem dle obr. 30b. (S) lze prodloužit pouze na (S,Z), neboť průtok podél (S,X) je již roven kapacitě. Čistý přírůstek přítoku do Z touto semicestou je 6. (S,Z) lze prodloužit na (S,Z,Y) s přírůstkem 4. (S,Z,Y) nelze prodloužit na (S,Z,Y,T), neboť průtok podél hrany <Y,T> je již roven kapacitě. Ovšem je možné prodloužení na (S,Z,Y,X) s přírůstkem přítoku do X o 4. Poznamenejme, že tato semicesta obsahuje "protisměrnou" hranu <Y,X>, což znamená redukci toku od X do Y o 4. Semicestu (S,Z,Y,X) lze prodloužit na (S,Z,Y,X,W) s přírůstkem 3 (nevyužitá kapacita na <X,W>) do W. Tuto semicestu lze prodloužit na <S,Z,Y,X,W,T> s čistým přírůstkem přítoku do T o 3. I když jsme dosáhli přírůstku 3 na této semicestě, postupujeme podél této semicesty v následujícím kroku v opačném směru. Jelikož <W,T> a <X,W> jsou "dopředné" hrany, tak průtok podél těchto hran lze zvýšit o 3. Protože <Y,X> je "protisměrná" hrana, tak průtok podél <X,Y> se sníží

o 3. <Z,Y> a <S,Z> jsou "dopředné" hrany, tak průtok podél nich zvýšíme o 3. Výsledek ukazuje obr. 30c.

Nyní vše opakujeme. (S) lze prodloužit na (S,Z) s přírůstkem 3, (S,Z) na (S,Z,Y) s přírůstkem 1 do Y a (S,Z,Y) na (S,Z,Y,X) s přírůstkem 1 do X. Protože však na hranách <S,X>, <Y,T> a <X,W> je dosaženo kapacity, tak nelze dále prodloužit žádnou semicestu a tudíž jsme dosáhli hledané řešení. Poznamenejme však, že řešení úlohy nemusí být jediné. Jiné optimum je uvedeno na obr. 30d, které získáme z grafu na obr. 30a tím, že uvažujeme semicesty (S,X,W,T) a (S,Z,Y,T).

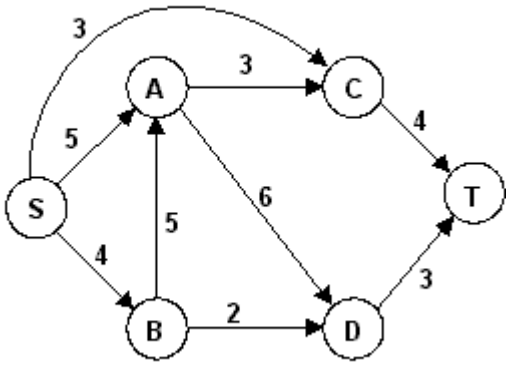
Zapíšeme nyní algoritmus řešení dané úlohy za použití takové pseudonotace, která je pochopitelná již z výuky předmětu programování v I. ročníku. Nechť je tedy dán ohodnocený graf (maticí kapacity potrubí), zdroj S a spotřebič T. Algoritmus pak lze vyjádřit např. následovně :

```
1. "Inicializace průtokové funkce (nulou) pro všechny hrany" ;
2. Nelzezlepsit := false ;
3. repeat
4.  "Pokus o nalezení semicesty od S do T s přírustkem přítoku x > 0 do T"  ;
5.  if  "Nelze najít semicestu"
6.    then    Nelzezlepšit := true
7.    else "Vyrovnání průtoku do každého uzlu semicesty   hodnotou x"
8. until  Nelzezlepsit
```

Těžiště celého algoritmu je v řádku 4. V literatuře je tento algoritmus označován jako **Ford-Fulkersonův algoritmus**. Počáteční verze zápisu tohoto algoritmu v jazyku C je v příloze 3. Pokud bylo jednou nějakého uzlu použito pro částečnou semicestu, nemůže být dále užit k rozšíření jiné semicesty. Proto je použito pole se jménem naceste, jehož složky indikují, zda se uzel nachází na nějaké semicestě, či nikoliv. Je třeba též indikovat, který uzel je koncovým uzlem částečné semicesty, kterou lze prodloužit připojením přiléhajícího uzlu. To indikují složky pole koneccest. Pro každý uzel semicesty je třeba si pamatovat předchozí uzel této semicesty a směr na hraně. Tudíž složka pole predchazi[i] je ukazatel na uzel, který na této semicestě předchází uzel *i*, a složka vpred[i] indikuje, zda existuje hrana od predchazi[i] do uzlu *i*. Složka pole zvyseni[i] udává, o jaké množství lze na této semicestě zvýšit průtok do uzlu *i*.

Cvičení

1. Využitím Ford-Fulkersonovy metody najděte (ručně) maximální průtok pro graf na obr. 31.



obr. 31

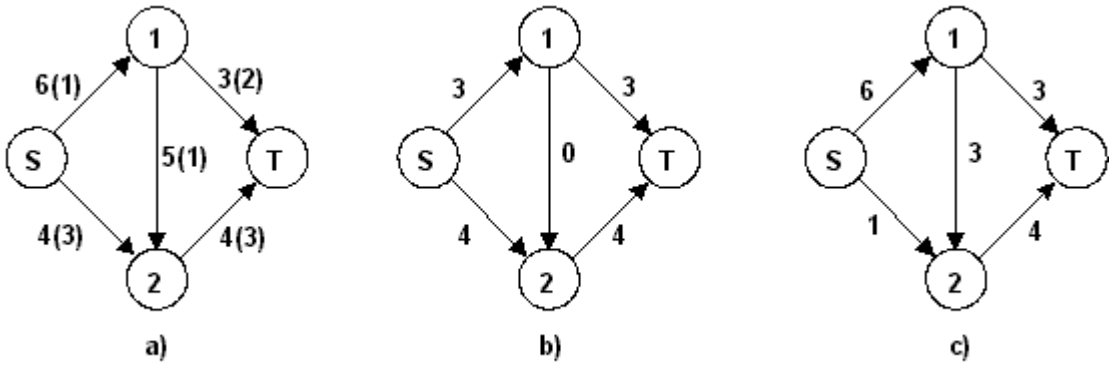
2. Pro velké grafy s mnoha uzly může být s ohledem na čas neefektivní prohledávání uzlů za účelem nalezení uzlu, tvořícího konec nějaké částečné semicesty. Je lepší, aby hodnoty, které jsou obsahem pole zvyseni byly udány jen pro takové uzly grafu, které tvoří konec semicest. Takové uzly grafu, které tvoří konec nějaké semicesty je pak výhodné ukládat do seznamu. Modifikujte odpovídajícím způsobem funkci pro určení maximálního toku sítí.

3. Zvažte, jak lze řešit případy, kdy kapacita zdroje a|nebo spotřebiče je omezená.

Návod : Zaveďte fiktivní zdroj (spotřebič) odkud vychází (kam vchází) jediná hrana, reprezentující toto omezení.

4. Uvažujme, že kromě kapacitní funkce $c(a,b)$ je dána též nákladovostní funkce $g(a,b)$, což jsou náklady na jednotku toku od uzlu a do uzlu b. Modifikujte funkci pro určení maximálního toku tak, aby maximalizovala celkový průtok od zdroje ke spotřebiči při nejnižších nákladech, tj. najděte takové řešení, kdy maximálnímu toku od S do T budou odpovídat nejmenší celkové náklady. Nechť např. údaje v závorkách u kapacit jednotlivých potrubí na obr. 32a znamenají jednotkové náklady. Pak řešení na obr. 32b znázorňuje jedno možné řešení maximálního toku sítí, kterému však jistě odpovídají vyšší náklady, než řešení podle obr. 32c.

Návod : Zaveďte si nejprve nějaký vektor, jehož složky udávají jednotkové náklady na vodu dodávanou do sítě v uzlu i. Při rozšiřování částečných semicest



obr. 32

lze realizovat prodloužení do nějakého uzlu i v případě, když tento uzel již na nějaké jiné částečné semicestě leží, pokud to ovšem vede ke snížení nákladů.

5. Modifikujte nyní řešení toku sítí pro případ, kdy je třeba určit takový tok sítí, aby celkový tok od zdroje ke spotřebiči dělený celkovými náklady byl maximální. Takový tok bychom mohli označit jako optimální.

Návod : Označíme-li G_i celkové náklady na přítok $f(x_i,T)$ ke spotřebiči, tak pro naše kriterium bude zřejmě platit :

$$\frac{\sum_i f(x_i,T)}{\sum_i G_i\, f(x_i,T)} \le \frac{f(x_k,T)}{G_k(x_k,T)} = \frac{1}{G_k} \ ,$$

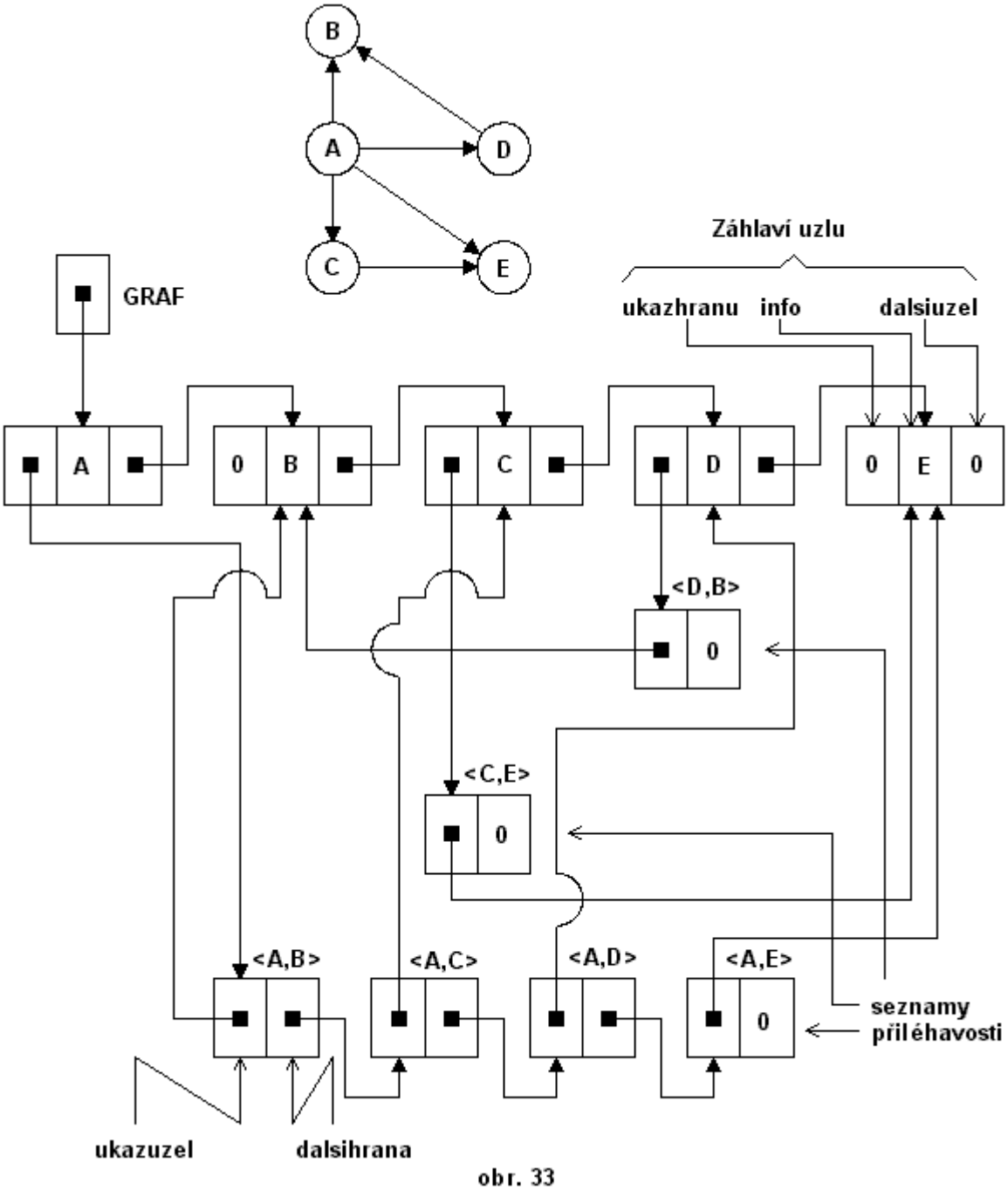
kde $G_k = \min_i G_i$.

Tedy je to jakýkoliv pozitivní tok podél nejlacinějších cest od S do T (pokud takových cest existuje více). Pak je logické určit maximální tok sítí, splňující zadané kriterium.

3.3 Spojová reprezentace grafů

Pokud graf musí být konstruován až během řešení problému, nebo musí být dynamicky měněn během provádění programu, tak při každém přidání nebo odebrání uzlu je třeba vytvořit novou matici přiléhavosti. To je neefektivní zvláště v takových v praxi se vyskytujících případech, kdy graf má mít 100 a více uzlů. I když graf obsahuje velmi málo uzlů (takže matice přiléhavosti a ev. matice vah pro ohodnocený graf

obsahuje ponejvíce nuly), tak je nutné rezervovat místo pro každou možnou hranu mezi dvěma uzly, bez ohledu na to, zda taková hrana existuje či nikoliv (nebo provést realokaci). Když graf obsahuje n uzlů, tak celkový počet míst je n^2. To jsou důvody pro použití spojové reprezentace grafů (viz příklad na obr. 33).



obr. 33

Pro každý uzel grafu lze vytvořit spojový seznam, který obsahuje všechny uzly přiléhající k danému uzlu. Daný uzel se pak pochopitelně může nalézat v mnoha jiných spojových seznamech přiléhavosti (jednou pro každý uzel, ke kterému přiléhá). To však vyžaduje, aby každý umístěný uzel obsahoval proměnný počet ukazatelů,

závisející na počtu přilehlých uzlů. Lepší je proto konstruovat tzv. **multispojovou strukturu** následujícím způsobem (viz obr. 33).

Uzly grafu jsou reprezentovány spojovým seznamem **záhlaví uzlů**. Každé toto záhlaví uzlu obsahuje 3 členy : info, dalsiuzel, ukazhranu. Člen info obsahuje nějakou informaci, spojenou s tímto uzlem, zatímco člen dalsiuzel je ukazatel na záhlaví uzlu, které reprezentuje další uzel grafu. Od záhlaví uzlu je referencován spojový seznam uzlů sekundárního typu, nazývaný **seznam přiléhavosti**, který reprezentuje hrany grafu vycházející z uzlu popsaného tímto záhlavím. Každý prvek seznamu přiléhavosti obsahuje dva členy : ukuzel a dalsihrana. Člen ukuzel je ukazatel na záhlaví uzlu grafu, obsahující uzel, do kterého vede odpovídající hrana grafu. Člen dalsihrana referencuje další prvek seznamu přiléhavosti.

Prvky seznamu záhlaví uzlu a seznamů přiléhavosti mohou mít odlišný formát. Ovšem v případě ohodnocených digrafů je třeba ke každé hraně udat její váhu a pak v případě, kdy informace v záhlaví uzlů a váhy jsou celočíselné, tak není nutné zvlášť definovat typ pro každý seznam a vystačíme se všeobecnou definicí typu, např. ve tvaru :

```
typedef struct prvek  {
        int info ;                  /* Záhlaví uzlů:celočíselné označení uzlů
                                       Seznam přiléhavosti : váha hran                */
        struct prvek *dalsi ;       /* Záhlaví uzlů  :  ukazatel na další uzel grafu
                                       Seznam přiléhavosti:ukazatel na přilehlý uzel */
        struct prvek *hrana ;       /* Záhlaví uzlů   : ukazatel na 1. hranu
                                       Seznam přiléhavosti : ukazatel na další hranu */
} PRVEK ;
```

Je pak užitečné, zapsat funkce realizující v daném kontextu primitivní operace s takovými grafy. Tak např. následující funkce přidá do seznamu záhlaví uzlů nový uzel, který ponese informaci x a vrátí ukazatel na tento uzel :

```
PRVEK *vloz ( PRVEK *graf, int x ) {
        PRVEK *p ;
        p = (PRVEK *) malloc ( sizeof ( PRVEK ) ) ;
        p->info = x ; p->hrana = NULL ; p->dalsi = graf ;
```

```
        return ( p ) ;
}
```

Jiná užitečná funkce vrací ukazatel na uzel nesoucí informaci x :

```
PRVEK *nalezni ( PRVEK *graf, int x ) {
        PRVEK *p = graf ;
        while ( p != NULL ) {
                if ( p->info == x ) break ; p = p->dalsi ;
        }
        return ( p ) ;
}
```

Zápis ostatních funkcí je ponechán na cvičení. Čtenář by si však měl uvědomit jiné důležité odlišnosti mezi maticí přiléhavosti a spojovou reprezentací grafů. Průchod každým řádkem matice přiléhavosti identifikuje všechny hrany vycházející z daného uzlu. U spojové reprezentace je tomu ekvivalentní průchod seznamem přiléhavosti počínající u záhlaví daného uzlu. Průchod sloupcem matice přiléhavosti identifikuje všechny hrany končící v daném uzlu - u spojové reprezentace taková možnost chybí. Ovšem spojovou reprezentaci lze modifikovat tak, že každé záhlaví by referencovalo 2 seznamy : jeden pro hrany vycházející z daného uzlu grafu a druhý pro hrany končící v tomto uzlu grafu. To by vyžadovalo zdvojené umístění hran a komplikovalo by to operace přidání a odebrání hran. Jinak by mohl každý prvek seznamu přiléhavosti obsahovat 4 ukazatele : na další hranu vycházející ze stejného uzlu, na další hranu končící ve stejném uzlu, na záhlaví uzlu ve kterém hrana končí a na záhlaví uzlu, ze kterého hrana vychází. Záhlaví uzlu by obsahovalo 3 ukazatele : na další záhlaví uzlu, na seznam hran vycházejících z uzlu a na seznam hran končících v uzlu. Programátor si pak musí zvolit alternativu podle specifiky řešeného problému a vzít v úvahu čas i paměť počítače. Je však samozřejmé, že pokud je uzel odebrán z grafu, tak se musí odebrat i všechny hrany, vycházející nebo končící v tomto uzlu.

Cvičení

1. Zvolte si svoji konkrétní reprezentaci grafu a zapište nové funkce pro přidání nového uzlu do grafu a pro nalezení uzlu s udanou informací.

2. Zapište funkci pro přidání hrany do grafu. Jako druhý a třetí argument předejte této funkci informaci, kterou nese uzel, z něhož hrana vychází a informaci, kterou nese uzel, v němž hrana končí. Jako poslední argument (pro ohodnocené grafy) předejte váhu hrany. V případě, že u ohodnoceného grafu taková hrana již existuje, tak se její váha přepíše, jinak se vytvoří nový prvek v odpovídajícím seznamu přiléhavosti.

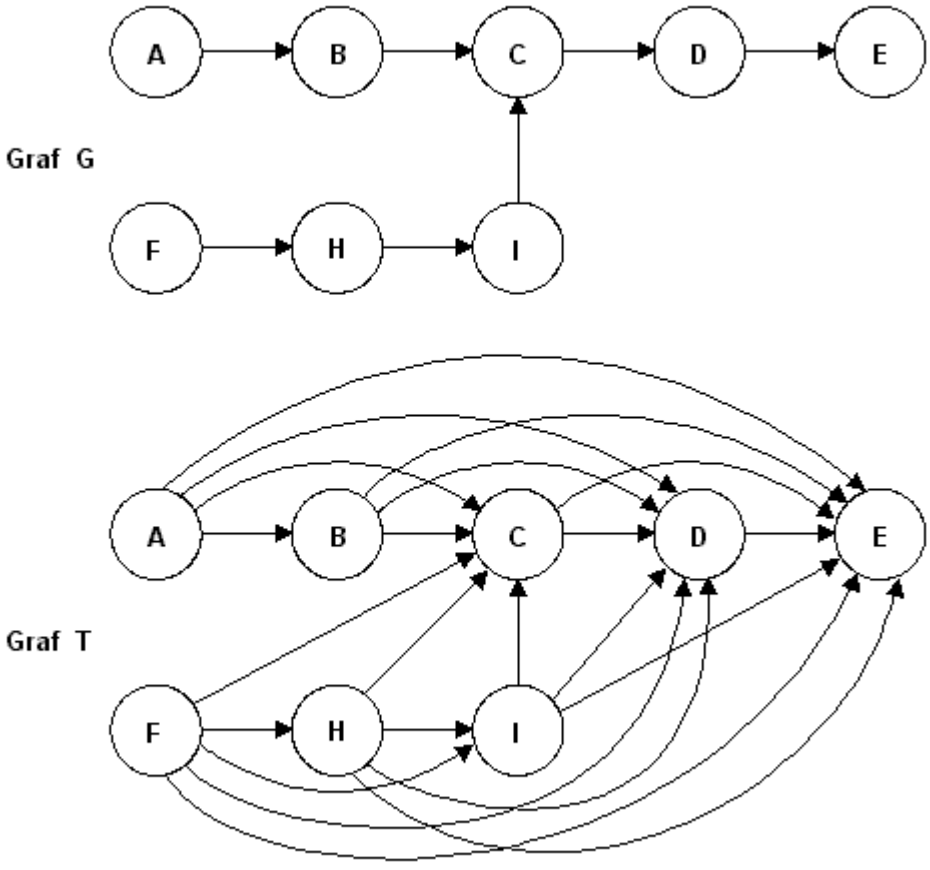
3. Zapište funkci pro odebrání hrany z grafu (pokud ovšem taková hrana existuje).

4. Zapište funkci, která zjistí, zda od prvního specifikovaného uzlu grafu do druhého specifikovaného uzlu grafu vede hrana, či nikoliv.

5. Zapište funkci pro odebrání uzlu z grafu.

3.4 Aplikace grafů na problémy plánování

Předpokládejme. že nějaká komplexní úloha je rozčleněna na podúlohy. Takovou komplexní úlohu znázorníme grafem, kde každý uzel reprezentuje jednu podúlohu, zatímco každá hrana $< x, y >$ reprezentuje požadavek, že podúlohu y nelze provést dříve, než byla skončena podúloha x (viz příklad na obr. 34). Některé podúlohy musí předcházet jiné, např. podúloha A musí být vykonána dříve, než podúloha B. Jiné podúlohy mohou být vykonány současně, např. A a F. Snahou bude seřadit jednotlivé podúlohy tak, aby komplexní úloha byla vykonána v nejkratším možném čase.



obr. 34

Tranzitivní uzávěr originálního grafu G je graf T, kde $< x, y >$ je hranou tehdy a jen tehdy, když v G existuje cesta z x do y (viz opět obr. 34). Tedy v grafu T existuje hrana z uzlu x do uzlu y, když podúloha x musí být vykonána dříve, než podúloha y. Poznamenejme hned, že jak G, tak i T nemohou obsahovat cykly, protože kdyby

existoval cyklus od *x* do *x*, tak podúloha *x* by nemohla být provedena dříve, než byla skončena ta samá podúloha - to je však zřejmé z kontextu.

Protože G nemůže obsahovat cyklus, tak v G musí být alespoň jeden uzel, který nemá předchůdce. To bude zřejmé z toho, když budeme předpokládat, že každý uzel má předchůdce. Zvolme uzel *z*, jehož předchůdce je *y*. Pak *y* nemůže být totožné se *z* - to by byl v grafu cyklus od *z* do *z*. Protože však každý uzel má předchůdce, tak *y* má předchůdce *x*, který není totožný s *y* ani *z*, atd. Tím dostaneme sekvenci disjunktních uzlů *z*, *y*, *x*, *w*, *v*, *u,* atd. Pokud by některé dva uzly této sekvence byly totožné, vznikl by cyklus, a protože graf obsahuje jen konečný počet uzlů, tak dva uzly musí být totožné - to je rozpor. Proto musí existovat alespoň jeden uzel, který nemá předchůdce.

V našem příkladě nemají předchůdce uzly A a F. Takové podúlohy lze provést okamžitě a současně s jinými takovými podúlohami, bez čekání na ukončení jiných podúloh. Jakmile tyto podúlohy byly provedeny, tak odpovídající hrany mohou být odebrány z grafu. Výsledný graf nemůže též obsahovat žádné cykly, protože uzly i hrany byly odebrány z grafu, který neobsahoval žádné cykly a tudíž i tento graf musí obsahovat alespoň jeden uzel, který nemá předchůdce. V našem příkladě jsou to uzly B a H. Podúlohy B a H mohou být provedeny současně ve druhé časové periodě.

Pokud předpokládáme, že každá podúloha trvá jednu časovou periodu, tak komplexní úlohu lze provést v šesti časových periodách :

časová perioda	podúloha	podúloha
1	A	F
2	B	H
3	I	
4	C	
5	D	
6	E	

Stejným způsobem lze formulovat mnoho reálných plánovacích problémů : plánování úloh zpracovávaných počítačem za účelem minimalizace času obrátky, kompilátor plánuje operace strojového kódu za účelem minimalizace času potřebného

pro provedení programu, plánování výrobního procesu za účelem minimalizace výrobního času atd. Všechny tyto problémy jsou stejné třídy a mohou být řešeny pomocí grafů.

V úplně nejhrubších rysech lze celý algoritmus popsat následovně :

1. čtení priorit a konstrukce grafu,
2. užití grafu pro určení podúloh, které mohou být provedeny současně.

V prvním kroku je nutné rozhodnout dvě podstatné záležitosti - formát vstupu a reprezentaci grafu. Je zřejmé, že ze vstupu musí být jasné, kterým podúlohám daná podúloha musí předcházet. Zcela konvenční cestou k naplnění tohoto požadavku je zadávání uspořádané dvojice podúloh. Každý vstupní řádek nechť obsahuje specifikaci dvou podúloh, přičemž první podúloha na řádku musí předcházet podúlohu, uvedenou na tomto řádku na druhém místě (nebo obecněji vstupní soubor obsahuje posloupnost uspořádaných dvojic). Data musí být taková, aby v grafu nevznikly cykly. Podúlohy mohou být reprezentovány číselným označením, posloupností znaků apod. Dále bude volena reprezentace řetězcem znaků, neboť to více odpovídá reálným situacím. Když je znám počet podúloh na začátku výpočtu, lze pro reprezentaci grafu užít matice přiléhavosti, jejíž všechny prvky se inicializují hodnotou *false*. Po přečtení priority se na odpovídající místo matice dosadí *true*. Ovšem je to nevhodné již vzhledem k označení uzlů, dále i vzhledem k tomu, že taková matice je v reálných situacích velmi řídká a i tomu, že je užitečné uvažovat z teoretického hlediska libovolný počet uzlů. Proto je volena spojová reprezentace grafů s dynamickou alokací..

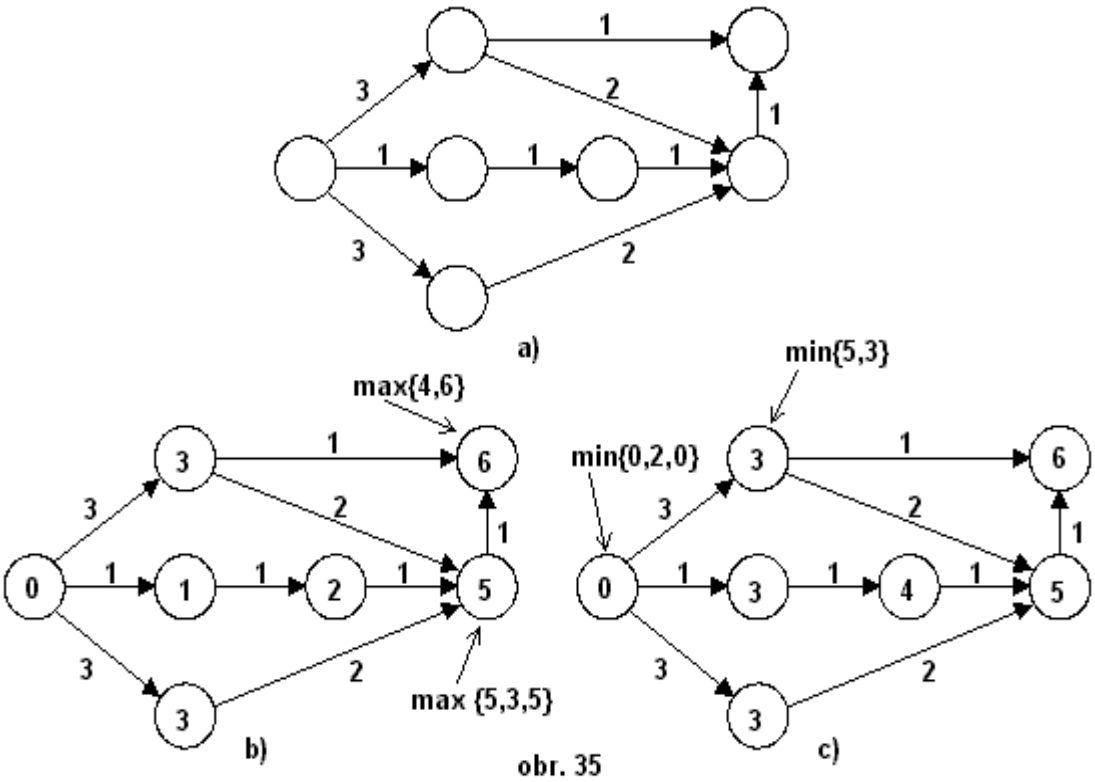
Druhý krok lze podrobněji popsat tímto algoritmem :

```
while  "Graf není prázdný" do
        begin   "Určení uzlů, které nemají předchůdce" ;
                "Výstup těchto uzlů s indikací, že v následující časové
                  periodě mohou být tyto činnosti provedeny současně" ;
                "Odebrání těchto uzlů a odpovídajících hran z grafu"
        end
```

Otázkou je, jak určovat uzly, které nemají žádného předchůdce. Nejjednodušší a funkční metodou je, že při konstrukci grafu budeme zapisovat do záhlaví každého uzlu další člen pocet, přičemž hodnota tohoto členu udává počet uzlů,

předcházejících daný uzel. Programové řešení této úlohy, resp. jedna z možných variant je uvedena v příloze 4 - všimněte si hlavně typů, zavedených zde pro uzly a hrany ! Poznamenejme, že nás nezajímá, které uzly předchází daný uzel, nýbrž jen jejich počet. Pokud člen pocet u nějakého uzlu je 0, tak tento uzel nepředchází jinému uzlu a může být umístěn do nějakého výstupního seznamu. Pokud však uzel umístíme do nějakého výstupního seznamu, tak je třeba projít jeho seznamem přiléhavosti a v každém uzlu přiléhajícím k tomuto uzlu snížit hodnotu pocet o 1.

Reálné situace však zpravidla bývají takové, že jsou dány i stovky podúloh, ovšem zpravidla jen 3, nebo 4 mohou být provedeny současně. Proto také provedení komplexní úlohy může vyžadovat řádově sto časových period a pokaždé by bylo nutné procházet seznamem za účelem lokalizace je mála takových uzlů, kde pocet = 0. Když ovšem identifikujeme uzly, kde pocet = 0, tak u lineárního, jednosměrně propojeného seznamu máme k dispozici pouze ukazatel na uzel samotný, ale při odebrání uzlu do výstupního seznamu potřebujeme i ukazatel na předchozí prvek. V příloze 4 bylo proto použito řešení, jehož princip spočívá v použití lineárního, obousměrně propojeného seznamu. Proto bylo třeba odpovídajícím způsobem modifikovat funkce nalezni a vloz - viz příloha 4.



obr. 35

Byť jsou uzly grafu ukládány do obousměrně propojeného seznamu, tak výstupní seznam může být jednosměrně propojený (chová se vlastně jako zásobník). Po provedení prvního kroku se jednou projde obousměrný seznam uzlů grafu za účelem inicializace výstupního seznamu obsahujícího takové uzly grafu, které při této inicializaci nemají žádného předchůdce. V každé časové periodě se pak prochází výstupním seznamem vytvořeným v předcházející časové periodě, výstupem jsou podúlohy reprezentované těmito uzly a hodnota pocet se v uzlech přiléhajících každému uzlu výstupního seznamu zmenší o 1. Pokud je takto snížená hodnota rovna 0, tak takový uzel lze umístit do výstupního seznamu pro následující časovou periodu. Pak jsou ovšem zapotřebí dva výstupní seznamy : jeden pro aktuální časovou periodu, který byl vytvořen v předchozí časové periodě a právě se jím prochází a druhý, který se vytváří v aktuální časové periodě a bude se jím procházet v následující časové periodě. Je na čtenáři, aby uvedenou variantu zdrojového textu programu v příloze 4 modifikoval pro svoje potřeby a svou hardwarovou platformu, nebo ještě lépe, aby zapsal svůj vlastní zdrojový text programu pro řešení této úlohy. To však spolu s dalšími úlohami je ponecháno na cvičení.

Uvažujme nyní jinou úlohu. Nechť je dán ohodnocený digraf, ve kterém každá hrana reprezentuje nějakou činnost a její váha čas, potřebný pro provedení této činnosti. Pokud v grafu existuje hrana $< a,b >$ a $< b,c >$, tak činnost $< a,b >$ musí být provedena dříve než $< b,c >$. Každý uzel x jako informaci nese i čas, ve kterém mohou být vykonány všechny činnosti, reprezentované hranami končícími v uzlu x. Takové grafy jsou označovány jako **sítě PERTH** - viz příklad na obr. 35a. Přiřadíme nyní každému uzlu x čas $\tau(x)$ jako minimální čas, ve kterém mohou být vykonány všechny činnosti, reprezentované hranami končícími v uzlu x. Začínáme v uzlech, ve kterých nekončí žádná hrana a kde $\tau(x) = 0$. Výsledek pro náš příklad je na obr. 35b. Dále přiřadíme každému uzlu x čas $T(x)$ jako maximální čas, ve kterém mohou být provedeny všechny aktivity, končící v uzlu x, aniž by se zvýšil celkový čas. Nyní naopak začínáme přiřazením času $\tau(x)$ v uzlech, které nemají následníka a končíme v uzlech bez předchůdce (viz obr. 35c). Lze pak ukázat, že v grafu existuje alespoň jedna cesta od uzlu, který nemá předchůdce do uzlu, který nemá následníka a to taková, že $\tau(x) = T(x)$ pro každý uzel x. Každá taková cesta se nazývá **kritickou cestou**. Pokud taková cesta je pouze jedna, tak když snížíme čas činností podél kritické cesty, sníží se

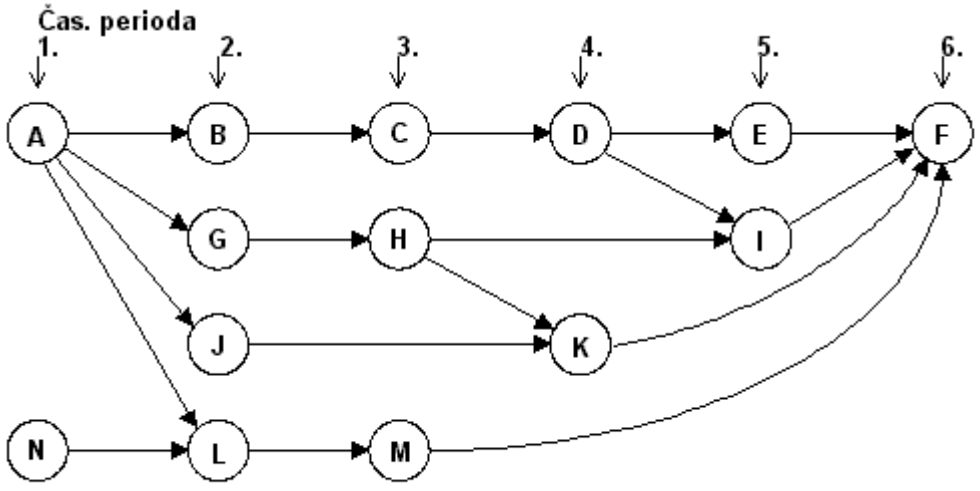
i minimální čas, ve kterém lze provést celou úlohu. Čtenář nechť zváží, jak postupovat v případě, kdy takových kritických cest je více.

Cvičení

1. Modifikujte pro svoje potřeby (nebo ještě lépe zapište svůj vlastní) zdrojový text programu, uvedený v příloze 4.

2. Je též možné, že provádění jednotlivých podúloh v minimálním počtu časových period lze organizovat různě. Tak např. v grafu dle obr. 34 lze celou úlohu provést v šesti časových periodách třemi různými způsoby :

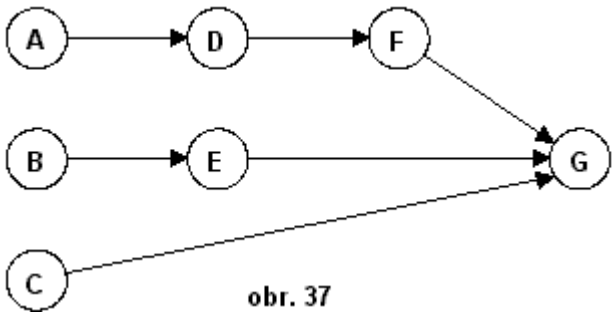
Perioda :	Metoda 1 :	Metoda 2 :	Metoda 3 :
1	A,F	F	A,F
2	B,H	A,H	H
3	I	B,I	B,I
4	C	C	C
5	D	D	D
6	E	E	E



obr. 36

Náš program generuje výsledné řešení tak, že každá podúloha je provedena v časové periodě s minimálním možným číslem (viz jiný příklad na obr. 36). Některé podúlohy však mohou být provedeny v následujících časových periodách. Tak např.

podúloha N dle obr. 36 může být provedena v 3. časové periodě, ovšem za předpokladu, že podúloha L bude provedena ve 4. a podúloha M v 5. časové periodě. Poúloha G může být též provedena ve 3. časové periodě, když podúloha H bude ve 4. a podúloha K v 5. časové periodě, atd. Pro každou podúlohu, která může být provedena v pozdější časové periodě generujte všechna taková možná řešení (včetně vyvolaných přesunů následujících podúloh).



obr. 37

Poznámka : Lze formulovat i obecnější úlohy, např. generujte všechny možné metody organizace provádění jednotlivých podúloh. Nebo uvažujme graf podle obr. 37. Náš program určí následující organizaci provádění komplexní úlohy :

Perioda :	Podúloha :
1	A,B,C
2	D,E
3	F
4	G

Tato organizace však předpokládá současné provedení až tří podúloh (v první časové periodě). Snadno však najdeme jinou organizaci provádění podúloh, aby se v žádné periodě neprováděly více než dvě podúlohy a komplexní úloha byly dokončena ve čtyřech časových periodách. Obecněji řečeno, je třeba najít takovou organizaci provádění podúloh, aby pro provedení komplexní úlohy v minimálním počtu časových period byl minimální počet současně prováděných podúloh. Takové úlohy jsou relativně složitější, jistě se však najdou takoví nadšenci, kteří je úspěšně vyřeší.

3. Předpokládejme nyní, že v každé časové periodě lze provést pouze jednu podúlohu, takže celou úlohu lze provést za k časových period, kde k je počet podúloh. Napište

zdrojový text programu, který vytvoří seznam přípustné posloupnosti podúloh. Takové konverze vstupní posloupnosti se nazývají **topologické třídění**.

Topologické třídění má mnoho praktických aplikací. Tak např. při sestavování studijního plánu je důležité myslet na to, že látka z určitých předmětů se musí probrat dřív, než se začne s dalším (novým) předmětem, protože tento předpokládá znalosti získané v předcházejících předmětech.

4. Zapište funkci pro určení všech kritických cest v síti PERTH.

Literatura

[1] RYCHLÍK, J. : Programovací techniky. České Budějovice, KOPP 1995.

[2] TENENBAUM, A. M. - AUGENSTEIN, M.J. : Data Structures Using Pascal. New Jersey, Prentice-Hall Inc. 1980.

[3] VĚCHET, V.: Programování. Rekursivní algoritmy v Pascalu. Liberec, VŠST 1990.

[4] WIRTH, N. : Algoritmy a štruktúry údajov. Bratislava, Alfa 1989.

Příloha 1

```
/* ================================
 *       DEC C  for  ULTRIX   v. 4 or higher
 * ================================
 */
#include <stdio.h>

typedef enum {
     LEVYSYN, PRAVYSYN
} TYPSYNA ;
typedef struct {
        unsigned kod ;
        char     pos ;
} TYPKOD ;
typedef struct {
        unsigned frekvence ;
        unsigned otec ;
        TYPSYNA  syn ;
} TYPUZLU ;

/* Pole abeceda :  */
static unsigned char *abeceda ;

/* Uzly Huffmanova stromu : */
static TYPUZLU *uzel ;

/*  Kódy znaků abecedy */
static TYPKOD *kodznaku ;

/* Počet symbolů abecedy */
static int pocet ;

/* Pole unsigned int obsahující Huffmanův kód */
static unsigned *huffkod ;

/*  poc_slov = počet slov délky 32b pole huffkod
    posl_bit = ukazatel na poslední bit kódu v posledním byte pole huffkod  */
```

```
static unsigned poc_slov ;
static char     posl_bit ;

/* Pomocné konstanty vytvoř podle délky slova
    0x80000000    lépe   0x1 << ( 8 * sizeof ( unsigned ) - 1 )
    0xFFFFFFFE    lépe   ~0 << 1
*/

/* Inicializace Huffmanova stromu. Předpokládá se vstup z textového
  souboru : počet symbolů abecedy, { symbol, cetnost, ... }              */
void init ( FILE *f )   {
        int i, j ;
        unsigned char s[2] ;

        fscanf  ( f, "%d", &pocet ) ;
        /* Alokace : */
        abeceda = (unsigned char *) malloc (pocet*sizeof(unsigned char)) ;
        uzel=(TYPUZLU *) malloc ((2*pocet-1)*sizeof(TYPUZLU)) ;
        kodznaku = (TYPKOD *) malloc ( pocet*sizeof(TYPKOD) ) ;
        /* Inicializce : */
        for ( i=0; i<2*pocet-1; i++ ) {
                uzel[i].frekvence = 0; uzel[i].otec = 0 ;
        }
        for ( i=0; i<pocet; i++ ) {
                fscanf ( f, "%s%d", s, &uzel[i].frekvence ) ;
                abeceda[i] = *s ;
        }

}

/* V poli uzel vytvoř Huffmanův strom */
void vytvor ( void ) {
        unsigned m, n1, n2, min1, min2, j ;

        for ( m=pocet; m<2*pocet-1; m++ ) {
                /* m =index pole "uzel" pro umístění uzlu,
                   vyber uzly n1 a n2 s nejmenší frekvencí */
                n1 = 0; n2 = 0 ;
                min1 = ~0; min2 = ~0 ;
                for ( j=0; j<m; j++ )
```

```
                    if ( !uzel[j].otec )
                     if ( uzel[j].frekvence < min1 ) {
                            min2 = min1 ; min1 = uzel[j].frekvence ;
                            n2 = n1 ; n1 = j ;
                     }
                     else   if ( uzel[j].frekvence < min2 ) {
                                min2 = uzel[j].frekvence; n2 = j ;
                            }
                /* Vytvoř n1 jako levý a n2 jako pravý podstrom k m */
                uzel[n1].otec = m ; uzel[n1].syn = LEVYSYN ;
                uzel[n2].otec = m ; uzel[n2].syn = PRAVYSYN ;
                uzel[m].frekvence = uzel[n1].frekvence + uzel[n2].frekvence;
        }
        /* Kořen Huffmanova stromu má index 2*pocet-2 */
}

/* Pro jednotlivé symboly abecedy zapiš binární kód do pole kodznaku */
void kod ( void )  {
        int i,j ;

        if ( pocet == 1 ) {
                kodznaku[0].pos = 0 ; kodznaku[0].kod = 0;
                return ;
        }
        for ( i=0; i<pocet; i++ )  {
                kodznaku[i].pos = -1 ;
                /* Průchod ke kořeni Huffmannova stromu */
                j = i ;
                while ( uzel[j].otec ) {
                        kodznaku[i].pos++ ;
                        /* Doplň zleva 0 : */   kodznaku[i].kod >>= 1  ;
                        if ( uzel[j].syn == PRAVYSYN )
                        /* Přepiš bit na 1 : */  kodznaku[i].kod | = 0x80000000 ;
                        j = uzel[j].otec ;
                }
        }
}

/* Vypsat binární kódy všech symbolů dané abecedy na standardní výstup */
void tisk ( void )  {
```

```
        int     i, j ;
        unsigned k;

        printf ( "Symbol:    Kod:\n" ) ;
        for ( i=0; i<pocet; i++ ) {
                printf ( "%8c       ", abeceda[i] ) ;
                k = kodznaku[i].kod ;
                for ( j=0; j<= kodznaku[i].pos; j++ ) {
                  if ( k & 0x80000000 ) printf ( "1" ) ;
                  else                  printf ( "0" ) ;
                  k <<= 1 ;
                }
                printf ( "\n" ) ;
        }
}


/* Zapsat bit do pole huffkod  */
static void zapis_bit ( int bit )  {
        static c_bitu = 0 ;

        if ( c_bitu > 8 * sizeof(unsigned)-1  )  {
         poc_slov++ ; c_bitu = 0 ;
         huffkod = (unsigned *) realloc( huffkod, (poc_slov+1)*sizeof ( unsigned ) ) ;

          huffkod[poc_slov] = 0 ;
        }
        huffkod[poc_slov] <<= 1 ;
        if ( bit ) huffkod[poc_slov] |= 0x1 ;
        else    huffkod[poc_slov] &= 0xFFFFFFFE ;
        c_bitu ++ ; posl_bit = c_bitu-1 ;
}

/* Zapsat binární kód do pole huffkod */
static void zapis_kod ( int i ) {
        unsigned k, j, posun ;
        char bit ;

        k = kodznaku[i].kod ;
        for ( j=0; j<=kodznaku[i].pos; j++ ) {
```

```
                if ( k & 0x80000000 ) zapis_bit ( 1 ) ;
                else                  zapis_bit ( 0 ) ;
                k <<= 1 ;
        }
}

/* Zapsat binární kód řetězce s do pole huffkod */
void string_kod ( char *s )  {
        int i=0, j, posun ;

        huffkod = (unsigned *) malloc ( sizeof(unsigned) ) ;
        poc_slov = 0 ; posl_bit = -1 ;
        while ( s[i] != '\0' ) {
                for ( j=0; j<pocet; j++ )  if ( s[i] == abeceda[j] ) break ;
                if ( j == pocet )  printf("\n*** Neznamy symbol ***\n" ) ;
                else zapis_kod ( j ) ;
                i++ ;
        }
        posun = 8 * sizeof ( unsigned ) - posl_bit - 1 ;
        huffkod[poc_slov] <<= posun ;
}

/*  Čti bit pole huffkod */
static char cti_bit ( int *posledni ) {
        static unsigned akt_slovo = 0 ;
        static char c_bitu = 0 ;
        static unsigned k ;
        char bit ;

        if ( !c_bitu && !akt_slovo ) k = huffkod[0] ;
        if ( c_bitu > 8*sizeof(unsigned) - 1 ) {
                c_bitu = 0 ; akt_slovo++ ; k = huffkod[akt_slovo] ;
        }
        *posledni = ( akt_slovo == poc_slov &&  c_bitu == posl_bit ) ? 1 : 0 ;
        bit = ( k & 0x80000000 ) ? 1 : 0 ; k <<= 1 ; c_bitu++ ;
        return ( bit ) ;
}

/* Dekóduj pole */
char *dekoduj ( void ) {
```

```
    char *s = (char *) malloc ( sizeof(char) ) ;
    int pos = 0, konec, koren, j ;
    TYPSYNA kam ;

    do  {
        konec = 0 ; koren = 2 * pocet - 2 ;
        do   {
            j = koren ;
            if ( cti_bit ( &konec ) ) kam = PRAVYSYN ;
            else                      kam = LEVYSYN ;
            while ( uzel[j].otec != koren || uzel[j].syn  != kam ) j-- ;
            koren = j ;
        } while ( koren >= pocet ) ;
        s[pos++] = abeceda[koren] ;
        s = (char *) realloc ( s, (pos+1)*sizeof(char));
    } while ( !konec ) ;
    s[pos] = '\0' ;
    return ( s ) ;
}
```

Příloha 2

```
/*****************************************
 *   DEC C for ULTRIX 4.1 or higher
 *****************************************/
#include <alfabeta.h>
#include <stdio.h>
#include <limits.h>

/*  Tato externí funkce vrací kvantitativní hodnocení situace  p->deska z hlediska
    "hráče" h. Vždy se doplní kontextuálně podle toho, pro jakou hru se algoritmus
    používá */
extern int hodnoceni ( UZEL *p, POLE  h ) ;

/*  Tato externí funkce vrátí ukazatel na spojový seznam  ( propojení prvků je jejich
    členy "bratr")  a všechny členy "deska" těchto  prvků jsou možné "reakce" hráče
    p->hrac na situaci p->deska. Pochopitelně člen "hrac" každého prvku seznamu
    musí být různy od p->hrac */
extern UZEL *gensez ( UZEL *p ) ;

static POLE kdo ;  /*   Specifikuje hráče, který je v situaci "as" na tahu; jako
                       externí proměnnou  ji používá funkce rozšíření   */

/*  Tato rekursivní funkce provádí rozšíření hracího stromu na požadovanou
    hloubku  "hloubka" počínaje uzlem referencovaným pointerem p, který je
    na úrovni "urovenp" hracího stromu, popřípadě ukončí rozšiřování při
    dosažení terminálového uzlu. V listech aktuálního hracího stromu se
    provádi kvantitativní hodnoceni situace na hrací desce a dále nepotřebné
    uzly se vrací do volné paměti. Vrací  kvantitivní hodnocení pro potřeby
    alfa-beta minimax */

int rozsireni ( UZEL *p, int urovenp, int hloubka, int albe ) ;
```

```
UZEL *dalsitah ( UZEL *as, int hloubka ) {
    /* Implementace alfa-beta algoritmu :
        as          : pointer na stav, např. na hrací desce ,
        hloubka     : požadovaná hloubka prognozy
     */
    int hodn, h ;
    UZEL *q, *u, *ns ;
    /* ns je pointer na stav, např. na hrací desce, který vznikne realizaci
       vybraného tahu */
    q = gensez ( as ) ;
    if ( q == NULL )
        /* Pak je ovšem as terminálový uzel. Odpověď hráče as->hrac na
           situaci as->deska je zbytečná a protože nemůže dojít ke změně
           situace, tak je logické vrátit "as" */
        return ( as ) ;
    else {
        /* Pak je q pointer na začátek seznamu na úrovni 1 hracího stromu,
           odkud vybereme nejlepší odpověď "hráče" as->hrac na situaci as-
           >deska */
        as->nsyn = q ;
        /* Inicializace : */ hodn = -INT_MAX ; kdo = as->hrac ; ns = NULL ;
        do {
            h = rozsireni ( q, 1, hloubka, hodn ) ;
            if ( h > hodn ) { /* q je aktuální "ns" */
                if ( ns != NULL ) free ( ns ) ;
                hodn = h ; ns = q ; q = q->bratr ;
            }
            else { /* q je pro as->hrac nezajímavý */
                u = q ; q = q->bratr ; free (u ) ;
            }
        } while ( q != NULL ) ;
```

```
                /* Pokud volající rutina nebude dále "as" používat, tak lze : */
                free ( as ) ;
                ns->nsyn = NULL ; ns->bratr = NULL ;
                return ( ns ) ;
        }
}


int rozsireni ( UZEL *p, int urovenp, int hloubka, int albe )  {
        UZEL *q, *u ;
        int  hodn, h, obr ;

        if ( hloubka == 1 ) /* Pak p je list hracího stromu a tudíž  vrátíme hodnocení
                               situace p->deska */
                return ( hodnoceni ( p, kdo ) ) ;
        else  { /* Generujeme další úroveň hracího stromu */
                q = gensez ( p ) ;
                if ( q == NULL )  /* Narazili jsme na terminálový uzel */
                        return ( hodnoceni ( p, kdo ) ) ;
                else {
                        /*  Seznam q je buď na úrovni "hloubka" a pak provedeme
                            přímo hodnocení, nebo toto hodnocení  provedeme po
                            návratu z rekursivního volání */
                        p->nsyn = q ;
                        hodn = ( q->hrac == kdo ) ? INT_MAX : -INT_MAX ;
                        obr  = ( q->hrac == kdo ) ? -1 : 1 ;
                        do  {
                                h = ( urovenp + 1 == hloubka )
                                  ? hodnoceni ( q, kdo )
                                  : rozsireni ( q, urovenp+1, hloubka, hodn ) ;
```

```
                        if ( obr * h >= obr * albe ) {
                                /* Uzel q a jeho mladší bratři jsou
                                   neperspektivni */
                            hodn = albe ;
                            while ( q != NULL ) {
                                u = q ;  q = q->bratr ; free ( u ) ;
                            }
                        }
                        else {
                           /* Určení, zda jsme již dosáhli perspektivni větve
                              hracího stromu : */
                            if ( obr * h > obr * hodn ) hodn = h ;
                            u = q ; q = q->bratr ; free ( u ) ;
                        }
                    } while ( q != NULL ) ;
                    return ( hodn ) ;
                }
            }
        }
```

Příloha 3

```
/* ====================================
 *   D E C   C   f o r   U L T R I X  (v. 4.1 or higher)
 * ====================================
 */

#define ANO     1
#define NE      0

void Max_tok ( unsigned int pocet,              /* Počet uzlů sítě  */
               unsigned int *kapacita,          /* Matice kapacit potrubí  */
               unsigned int zdroj,              /* Uzel sítě - zdroj  */
               unsigned int spotrebic,          /* Uzel sítě - spotřebič */
             /* ------------------------------  Výstup  -------------------------- */
               unsigned int *tok,               /* Matice maximálního toku podél
                                                   hran sítě  */
              unsigned int *sumtok              /* Celkový tok sítí    */
             )     {


 unsigned int *predchazi = (unsigned int *)malloc( pocet * sizeof (unsigned int) ) ;
 unsigned int *zvyseni    = ( unsigned int *)malloc(pocet * sizeof (unsigned int) ) ;
 char *koneccest = ( char *) malloc ( pocet * sizeof (char) ) ;
 char *vpred       = ( char *) malloc ( pocet * sizeof (char) ) ;
 char *naceste     = ( char *)  malloc ( pocet * sizeof (char) ) ;

 unsigned int pred, uzel, i, x ;

 /* ---------------------------------------------------------------------------------
  *  Uzly jsou na vstupu číslovány jako přirozená čísla 1, 2, ....., pocet
  * ---------------------------------------------------------------------------------
  */
 zdroj--; spotrebic--;
 /* ---------------------------------------
  *  Počáteční inicializace toku sítí
  * ---------------------------------------
  */
```

```
for ( uzel=0; uzel<pocet; uzel++ )  for ( i=0; i<pocet; i++ ) tok[uzel*pocet+i] = 0 ;
*sumtok = 0 ;


/* ------------------------------------------------------------------
 *  Hledání semicest od uzlu "zdroj" do uzlu "spotřebič"
 * ------------------------------------------------------------------
 */
for (;;)   {
        /* ----------------------------
         *  Počáteční inicializace
         * ----------------------------
         */
        for ( uzel=0; uzel<pocet; uzel++ ) koneccest[uzel] = naceste[uzel] = NE ;
        koneccest[zdroj] = naceste[zdroj] = ANO ; uzel = zdroj ; zvyseni[zdroj] = ~0 ;

        /* ------------------------------------------------------------------------------
         *  Hledání prodloužení částečné semicesty od koncového uzlu této
         *  semicesty
         *  ------------------------------------------------------------------------------
         */
        while ( uzel < pocet )    {
          koneccest[uzel] = NE ;
          for ( i=0; i<pocet; i++ )   {
                if ( tok[uzel*pocet+i] < kapacita[uzel*pocet+i]  && !naceste[i] )   {
                     koneccest[i] = naceste[i] = vpred[i] = ANO ; predchazi[i] = uzel ;
                     x = kapacita[uzel*pocet+i] - tok[uzel*pocet+i] ;
                     zvyseni[i] = ( zvyseni[uzel] < x ) ? zvyseni[uzel] : x;
                }
                if ( tok[i*pocet+uzel]  && !naceste[i] )   {
                   naceste[i] = koneccest[i] = ANO ;  vpred[i] = NE ;
                   predchazi[i] = uzel ;
                   zvyseni[i] = ( zvyseni[uzel] < tok[i*pocet+uzel] )
                                                  ?zvyseni[uzel] :tok[i*pocet+uzel] ;
                }
          }  /* for ..... */
```

```
            /* ------------------------------------------------------------------------
             *  Vybrat konec částečné semicesty pro prodloužení
             * ------------------------------------------------------------------------
             */
            for ( i=0 ; i<pocet && !koneccest[i] || i == spotrebic; i++ ) ; uzel = i ;

        } /* while ( uzel < pocet )  */
        /* ------------------------------------------------------------------------------------------
         *  Realizovat zvýšení toku na nalezené semicestě  od "zdroje" ke
         *  "spotřebiči"
         * ------------------------------------------------------------------------------------------
         */
        if ( ! naceste[spotrebic] )  break ;
        x = zvyseni[spotrebic] ; *sumtok += x ; uzel = spotrebic ;
        do     {
                pred = predchazi[uzel] ;
                if ( vpred[uzel] ) tok[pred*pocet+uzel] += x ;
                else               tok[uzel*pocet+pred] -= x ;
        } while ( (uzel = pred ) != zdroj ) ;
    }  /* for (;;) */ ;
    free ( predchazi ) ; free ( zvyseni ) ; free ( koneccest ) ;
    free ( vpred ) ; free ( naceste ) ;
}   /*  Max_tok */
```

Příloha 4

```
/* ------------------------------------------------------------
 *    D E C   C   f o r   U L T R I X  (v.4.1 or higher)
 * ------------------------------------------------------------
 */

#include <stdio.h>
#include <string.h>

typedef struct hrana  {
        void *uzel ;
        struct hrana *dalsi ;
} HRANA ;

typedef struct uzel  {
        char  info[36] ;
        int   pocet ;
        HRANA *hrana ;
        struct uzel *dalsi, *predchozi ;
} UZEL ;

/* Tato funkce vyhledá uzel s informací x a vrátí ukazatel na tento uzel.
   Pokud takový uzel neexistuje, tak jej přidá do grafu */

UZEL *nalezni ( UZEL **graf, char *x ) {
        UZEL *p = *graf ;

        while ( p != NULL )  {
                if ( strcmp ( p->info, x ) == 0 ) return ( p ) ;
                p = p->dalsi ;
        }
        p = (UZEL *) malloc ( sizeof ( UZEL ) ) ;
        strcpy ( p->info, x ) ; p->pocet = 0 ; p->hrana = NULL ;
        p->dalsi = *graf ; p->predchozi = NULL ;
        if ( *graf != NULL ) (*graf)->predchozi = p ;
        *graf = p ; return ( p ) ;
}
```

```
/* Tato funkce vloží do seznamu hranu od p do q ; pokud taková hrana již existuje,
   výpis poznámky na standardní výstup  */

void vloz ( UZEL *p, UZEL *q )  {
        HRANA *r = p->hrana ;

        while ( r != NULL )  {
                if (  (UZEL *)r->uzel == q ) break ;
                r = r->dalsi ;
        }
        if ( r != NULL )
           printf ( "Naslednost cinnosti %s a %s jiz byla zadana\n",  p->info, q->info ) ;
        else  {  /* Umístění nové hrany */
                r = (HRANA *) malloc ( sizeof ( HRANA ) ) ;
                q->pocet++ ; (UZEL *)r->uzel = q ;
                r->dalsi = ( p->hrana == NULL ) ? NULL : p->hrana ;
                p->hrana = r ;
        }
}


void main ( void )  {
        UZEL *graf = NULL, *vystup = NULL, *dalsivystup, *p, *q ;
        HRANA *t ;
        FILE *f ;
        char s1[36], s2[36] ;
        int perioda = 0 ;

        if ( ( f = fopen ( "vstup", "rt" ) ) == NULL )  {
                printf ( "Soubor nelze otevrit\n" ) ; exit ( -1 ) ;
        }
        while ( fscanf ( f, "%s %s" , s1, s2 ) )  {
                if ( feof ( f ) ) break ;
                vloz ( nalezni ( &graf,s1) , nalezni ( &graf, s2 ) ) ;
        }
        fclose ( f ) ;
        /*  Graf byl vytvořen. Prohledání seznamu uzlů grafu a umístění
            všech uzlů, kde člen pocet je 0, do výstupního seznamu,
            referencovaného proměnnou vystup */
```

```
p = graf ;
while ( p != NULL )  {
 q = p->dalsi ;
 if ( p->pocet == 0 ) {                /* Odeber uzel */
  if ( q != NULL ) q->predchozi = p->predchozi ;
  if ( p->predchozi != NULL ) p->predchozi->dalsi = q ;
  else                              graf = q ;
  /* Umístění do výstupního seznamu */
  p->dalsi = vystup ; vystup = p ;
 }
 p = q ;
}


/* Simulace časových period */
while ( vystup != NULL ) {
 printf ( "Perioda c.%d:\n", ++perioda ) ;
 /* Inicializace výstupního seznamu pro následující periodu */
 dalsivystup = NULL ; p = vystup ;
 while ( p != NULL ) {
  printf ( "%-40s\n", p->info ) ;
  /* Průchod hranami vycházejícími z p */
  t = p->hrana ;
  while ( t != NULL ) {
   /* Snížení hodnoty proměnné pocet */
   q = (UZEL *)t->uzel ; q->pocet-- ;
   if ( q->pocet == 0 ) {
        /*  Uzel t lze odebrat a umístit do výstupního seznamu
            pro další periodu */
        if ( q->dalsi != NULL ) q->dalsi->predchozi = q->predchozi ;
        if ( q->predchozi != NULL ) q->predchozi->dalsi = q->dalsi ;
        else                              graf = q->dalsi ;
        q->dalsi = dalsivystup ; dalsivystup = q ;
   }
   t = t->dalsi ;
  }
  p = p->dalsi ;
 }
 /* Další výstup se stává aktuálním */
 vystup = dalsivystup ;
```

```
            printf ( "-----------------------------------------\n" ) ;
        }
        if ( graf != NULL ) {  printf ( "Chyba dat - cyklus\n" ) ; exit ( -1 ) ;  }
        exit ( 0 ) ;
}
```