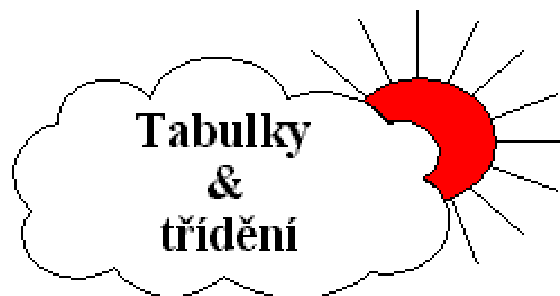
TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta strojní **Katedra aplikované kybernetiky**

doc. Ing. Vladimír V ě c h e t , CSc.

ALGORITMY A DATOVÉ STRUKTURY



Liberec 1997

© **doc. Ing. Vladimír Věchet, CSc.**
Recenzoval : doc. RNDr. Ing. Miloslav Košek, CSc.

ISBN 80 - 7083 - 227 - 4

OBSAH

1. ÚVOD 4
2. TABULKY 5
   2.1 Tabulky s přímým a sekvenčním výběrem 6
   2.2 Uspořádané tabulky 9
   2.3 Rozptýlené tabulky 16
3. TŘÍDĚNÍ 21
   3.1 Porovnání různých metod třídění 22
   3.2 Třídění transpozicí 26
   3.3 Třídění výběrem 30
   3.4 Třídění vkládáním 39
   3.5 Třídění sléváním a poziční třídění 43
   3.6 Principy vnějšího třídění 47

Literatura 51

Příloha 52

Motto : "Mýlit se je lidské. Ale něco dokonale zašmodrchat je možné pouze pomocí počítače"

Murphyho pátý zákon spolehlivosti

1. ÚVOD

Učební text je určen studentům Technické univerzity v Liberci, fakulty strojní, oboru automatizované systémy řízení ve strojírenství. Pokrývá pouze část obsahu předmětu "Algoritmy a datové struktury", vyučovaném v tomto oboru studia. Předkládaný učební text byl psán s cílem zabránit nářkům studentů o nedostupnosti jiné (a jistě obšírnější a lepší) literatury, nikoliv však s cílem nahradit obsah přednášek. Bylo by proto nepříjemné akceptovat tuto skutečnost ex post, a proto považuji za vhodné umístit takovou zmínku přímo do úvodu.

Zápis algoritmu v nějakém symbolickém jazyku je jistě dobrou metodou pro ověření jeho správnosti. Proto se předpokládá, že čtenář bude samostatně řešit úlohy pro cvičení a algoritmy bude zapisovat v symbolickém jazyku. Vzhledem k návaznosti předmětů vyučovaných v oboru automatizované systémy řízení ve strojírenství doporučuji jazyk C a takto jsou některé vzorové příklady uváděny i v textu. Nicméně ani zápis algoritmu v symbolickém jazyku nedává stoprocentní jistotu, že je vše vyřešeno do všech důsledků a v jakémkoliv kontextu - přesně podle pravidla, že "úplně dokonalý je pouze takový program, který se již vůbec nepoužívá". I to je třeba mít na paměti a provést nezbytná ošetření hlavně u argumentů předávaných zapsaným funkcím.

Za pečlivé přečtení rukopisu skript a korektury děkuje autor doc. RNDr. Ing. Miloslavu Koškovi, CSc.

2. TABULKY

Tabulky patří mezi abstraktní datové typy, velmi často používané při programování. Z hlediska dalších potřeb stačí, pokud definujeme tabulku jako množinu prvků, které lze identifikovat pomocí tzv. **klíčů**. Pro prvky této množiny se v běžné terminologii používá názvu věty a tedy s každou větou je spojen klíč, který slouží právě k identifikaci věty. Pokud takový klíč je přímo součástí věty, označuje se jako **interní klíč**, zatímco zbylá část věty tvoří tzv. **výplň**. Lze však vytvořit i samostatnou tabulku obsahující klíče a s každým klíčem spojit ukazatel na vlastní větu. Pak mluvíme o tzv. **externích klíčích**. Je tedy dána alespoň jedna množina klíčů identifikujích dané věty, tj. dvě různé věty nemohou mít stejnou hodnotu klíče. Pokud již takových množin klíčů je více, mluvíme o **primárních, sekundárních, ... klíčích**.

Základní operací nad tabulkou je tzv. **vyhledávání** (angl. searching). Vyhledávací algoritmus je takový algoritmus, který pro dané *k* nalezne větu, jejíž klíč je *k* a vrací buď větu, nebo obecněji může vracet jen ukazatel na tuto větu. Představíme-li si jako tabulku např. telefonní seznam, je zřejmé, že primárním klíčem je jméno účastníka, sekundárním jeho příjmení, terciálním adresa účastníka. Kdybychom prováděli vyhledávání pouze podle primárního klíče, tak bychom asi po vyhledávacím algoritmu požadovali, aby vracel všechny věty (účastníky) se shodným primárním klíčem - těch může být více.

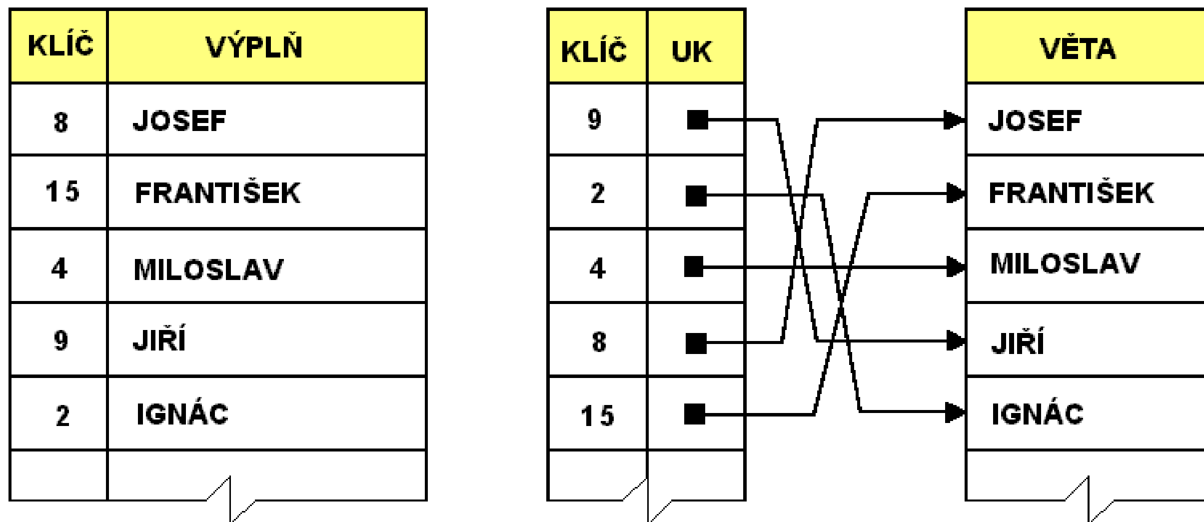
Zatím ponechme stranou otázku, jakým způsobem je tabulka organizována (jak je implementovaná v symbolickém jazyku) - jako pole, soubor, nebo jako abstraktní datový typ seznam, strom, graf apod. S ohledem na jednoduchost a názornost grafického znázorňování předpokládejme, že je tabulka organizována jako pole a aplikace jiné organizace tabulky nechť je buď náplní uvedených cvičení, nebo je třeba ji řešit kontextuálně (tzn. pro jaký konkrétní případ je tabulka použita). Pak např. rozdíl mezi interním a externím klíčem můžeme schematicky znázornit podle obr. 1.

Typickými operacemi s tabulkou jsou :

- ➔ čtení (vyhledání) věty s udaným klíčem ,
- ➔ přidání nové věty do tabulky ,
- ➔ aktualizace věty s udaným klíčem ,
- ➔ odebrání věty s udaným klíčem.

Frekvence používání těchto operací nebývají stejné, někdy se některé operace aplikují se zanedbatelně malou frekvencí, nebo vůbec (třeba odebrání věty). Podle základní operace s tabulkou, kterou je vyhledávání, se rozlišují tyto druhy tabulek :

- tabulky s přímým výběrem ,
- tabulky se sekvenčním výběrem ,
- uspořádané tabulky ,
- rozptýlené tabulky .



obr. 1

2.1 Tabulky s přímým a sekvenčním výběrem

U **tabulek s přímým výběrem** existuje jednoznačné zobrazení množiny klíčů na množinu paměťových míst, které jsou k dispozici pro tuto tabulku. Je tedy k dispozici nějaká transformační funkce $h(k)$, která pro každou hodnotu klíče k určí místo v paměti, kde je věta s klíčem k uložena.

Vhodnou transformační funkci poskytuje např. mapovací funkce pro pole - to je známo již ze základů programování. Tak např. v Pascalu deklaraci ve tvaru

var TABULKA : **array** [KLIC] **of** VETA ;

lze chápat jako tabulku organizovanou jako pole, kde v selektoru tohoto pole je indexový výraz jako jednoznačný externí klíč pro věty. Pochopitelně typ KLIC musí být ordinální typ. Nebo

```
typedef enum {
        PO, UT, ST, CT, PA, SO, NE
} DEN ;
typedef char *VETA ;
```

```
                    VETA  tabulka[ ]  =  {
                            "PAVEL", "PETR", "HUGO", "VIT", "IVO", "JAN", "MILA"
                    } ;
....................
printf ( "V nedeli ma sluzbu %s\n",  tabulka[NE] ) ;
....................
```

odpovídá také tabulce s přímým výběrem.

Tabulky s přímým výběrem se vytvářejí zpravidla v případech, kdy množina klíčů je málo početná a všechny operace s tabulkou se realizují jako známé operace s polem (při respektování všech omezení vztažených k poli - viz dále u rozptýlených tabulek).

Tabulky se sekvenčním výběrem jsou posloupnosti vět, ve kterých se vyhledávání provádí postupným prohledáváním této posloupnosti a porovnáváním zadaného klíče s klíčem vět tak dlouho, pokud nedojde ke shodě nebo vyčerpání všech vět. Tabulku se sekvenčním výběrem můžeme organizovat jako pole různým způsobem, např. :

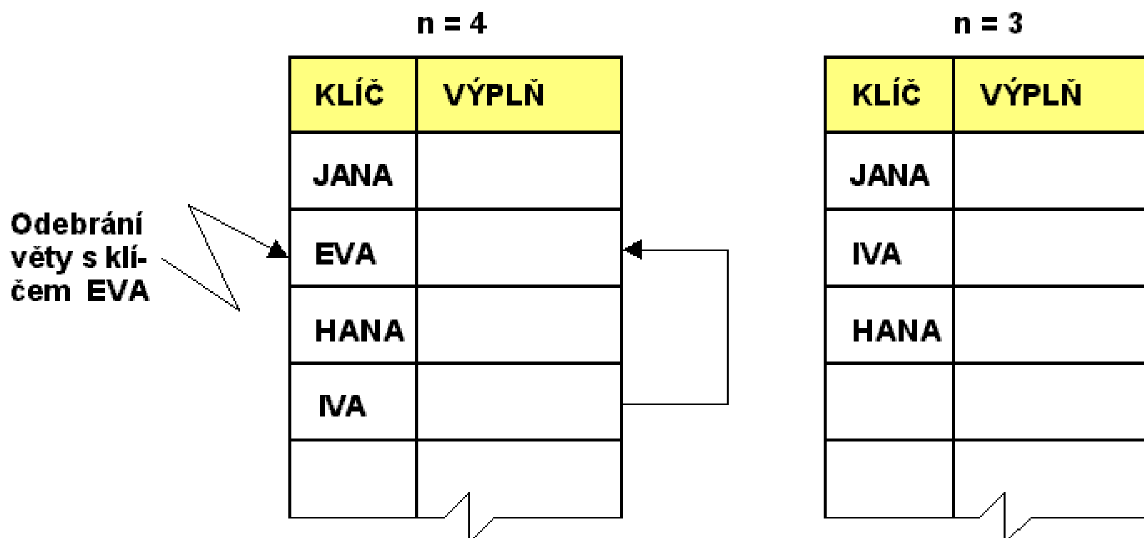
```
#define    MAX    100    /*  Maximální počet vět v tabulce  */
.....................
typedef struct   {
          unsigned    klic  ;
          VYPLN      vypln ;
} VETA ;
....................
int   n = 0  ;     /*  Aktuální počet vět v tabulce, inicializace */
VETA    tab[MAX] ;
```

Operaci vyhledání věty se zadaným klíčem budeme provádět postupným prohledáváním posloupnosti vět a porovnáváním zadaného klíče s klíčem vět. Při shodě zadaného klíče s klíčem věty bude výsledkem index složky pole jako ukazatel na vyhledanou větu, -1 v případě, že věta se zadaným klíčem v tabulce není (tab jako proměnná s paměťovou třídou **extern**):

```
int search ( unsigned k )  {
          for ( int i=0; i<n; i++ ) if ( k = = tab[i].klic ) return ( i ) ;
          return ( -1 ) ;
}
```

Operaci odebrání věty se zadaným klíčem můžeme implementovat tak, že nejprve prohledáme posloupnost vět a v případě, že věta se zadaným klíčem v tabulce je, tak na její místo přesuneme poslední větu posloupnosti a hodnotu *n* zmenšíme o 1 (viz obr 2). Operaci přidání

věty do tabulky chápejme tak, že věta s daným klíčem se připojí na konec posloupnosti vět, ovšem jen v případě, kdy tabulka již neobsahuje větu se stejným klíčem.



obr 2

Je logické, že čas potřebný pro vyhledání věty se zadaným klíčem je u tabulek se sekvenčním výběrem závislý na umístění této věty v posloupnosti vět. Tento čas bude nejmenší, když vyhledávaná věta bude první větou posloupnosti vět a naopak největší, když je poslední v této posloupnosti. Předpokládejme, že *p(i)* je pravděpodobnost vyhledávání věty s *i*-tým klíčem v posloupnosti *m* vět, tzn.

$$\sum_{i=1}^{m} p(i) = 1.$$

Pak průměrný počet porovnání *v* při vyhledávání bude

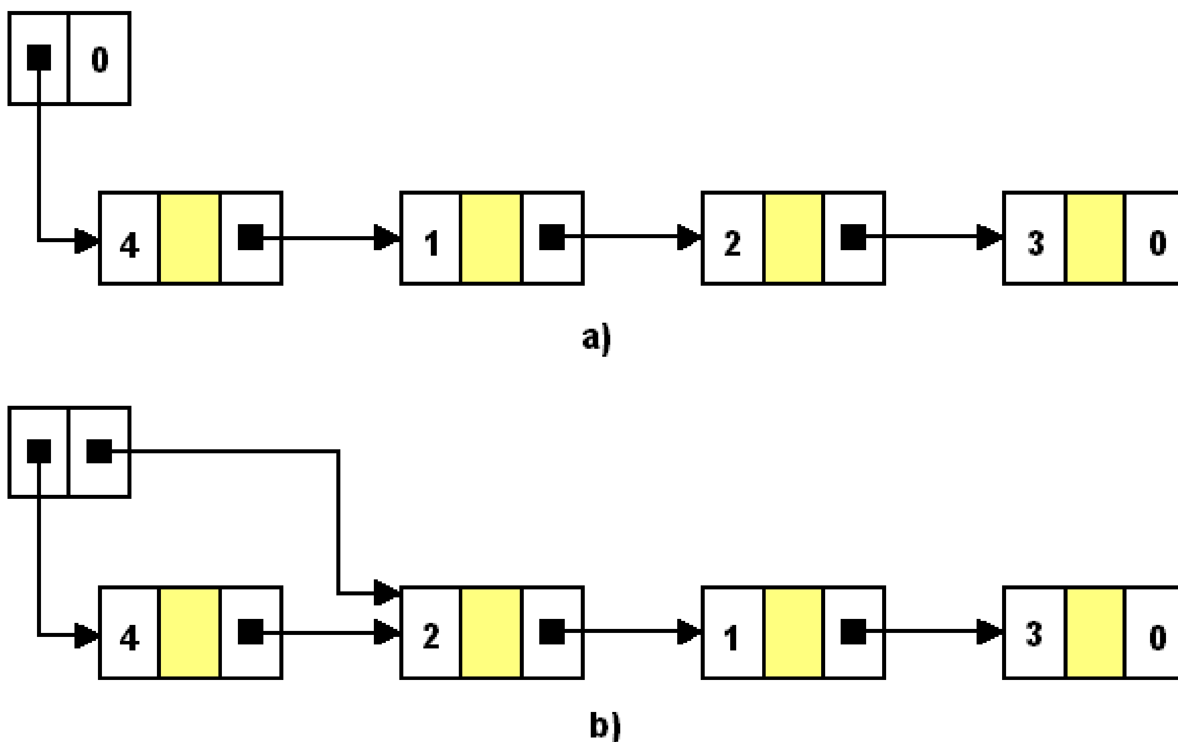
$$v = \sum_{i=1}^{m} i\, p(i)$$

a *v* bude zřejmě nejmenší, když

$$p(1) \geq p(2) \geq \ldots\ldots\ldots\ldots\ldots \geq p(m)$$

Potíž ovšem spočívá v tom, že hodnoty *p(i)* nejsou zpravidla známy a priori, lze je však odhadnout jako četnosti vyhledávání věty s *i*-tým klíčem při mnohonásobně opakovaném vyhledávání. V tom případě je však výhodné, upravit při každém vyhledávání posloupnost vět s ohledem na zmenšení času potřebného pro vyhledávání. Jednou takovou metodou je tzv. **transpoziční metoda**, jejíž princip spočívá v tom, že vyhledaná věta si v posloupnosti vět vymění svoje místo s větou, která ji bezprostředně předchází (viz příklad na obr. 3). V tomto příkladě je tabulka se sekvenčním výběrem organizovaná jako lineární seznam, nové věty se budou připojovat na konec seznamu a vyhledání spočívá v nastavení vnitřního ukazatele seznamu

na větu s udaným klíčem a následné transpozici. Je-li aktuální stav znázorněn schematem podle obr. 3a, pak po vyhledání věty s klíčem 2 je situace znázorněna obr. 3b (jsou uvažovány celočíselné klíče, výplň věty je vyznačena vyplněním příslušného políčka).



obr. 3

Cvičení

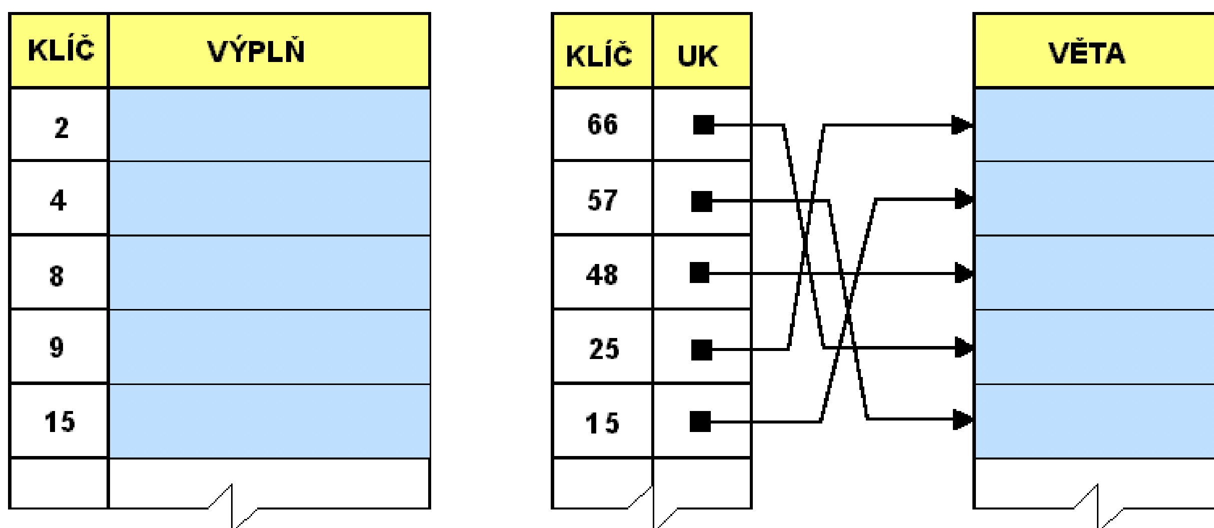
1. Napište program, který přečte zadaný vstupní text a vytiskne tabulku četnosti výskytu jednotlivých písmen textu. Použijte tabulku s přímým výběrem organizovanou jako pole.
2. Uvažujme tabulku se sekvenčním výběrem, organizovanou jako lineární seznam. Zapište v symbolickém jazyku operace s takovou tabulkou.
3. Aplikujte transpoziční metodu na tabulku se sekvenčním výběrem, organizovanou jako pole.

2.2 Uspořádané tabulky

Uspořádané tabulky jsou takové tabulky, u kterých je na množině klíčů identifikujících věty definováno nějaké uspořádání, které je přeneseno i na posloupnost vět, pomocí níž je taková tabulka zobrazena. U takových tabulek lze pak efektivněji provést operaci vyhledávání vět s daným klíčem. Příklady uspořádaných tabulek jsou na obr. 4.

Použijeme pro příklad dříve uvedené :

```
#define    MAX    100    /*  Maximální počet vět v tabulce  */
......................
typedef struct   {
          unsigned    klic  ;
          VYPLN       vypln ;
}  VETA ;
......................
int   n = 0  ;    /* Inicializace tabulky */
VETA  tab[MAX] ;
```



obr. 4

Pak z hlediska dalších potřeb budeme uspořádání definovat následovně (není to však nutné - viz obr. 4) :

```
tab[i].klic   <   tab[i + 1].klic   ,    0  <=  i  <  n  - 1 ,
```

což znamená vzestupné setřídění.

Vyhledávání v uspořádaných tabulkách lze podstatně urychlit aplikací metody binárního vyhledávání : porovnáme zadaný klíč s klíčem věty uprostřed tabulky a podle výsledku porovnání hledáme větu stejnou metodou v první nebo druhé polovině tabulky. V našem příkladě by bylo možné odpovídající funkci zapsat např. následovně :

```
int binary_search  ( unsigned  k, VETA *tab, int n )  {
          int d = 0, h = n - 1, str ;
```

```
        while ( d <= h )  {
            str = ( d + h ) / 2 ;
            if ( tab[str].klic < k )        d = str + 1 ;
            else if ( tab[str].klic > k )  h = str + 1 ;
                 else                       return ( str ) ;
        }
        return ( -1 ) ;
    }
```

Takové funkce je třeba zapisovat kontextuálně. Kdybychom např. použili definice

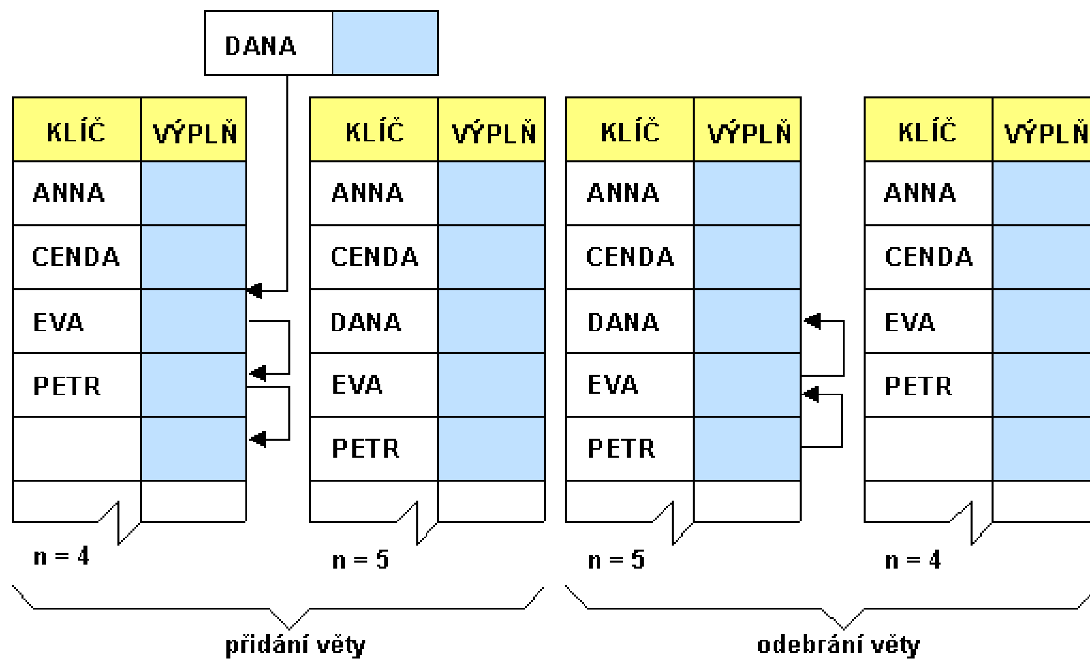
```
    typedef  struct   {
        char     klic[20] ;
        VYPLN vypln ;
    } VETA ;
```

tak funkce

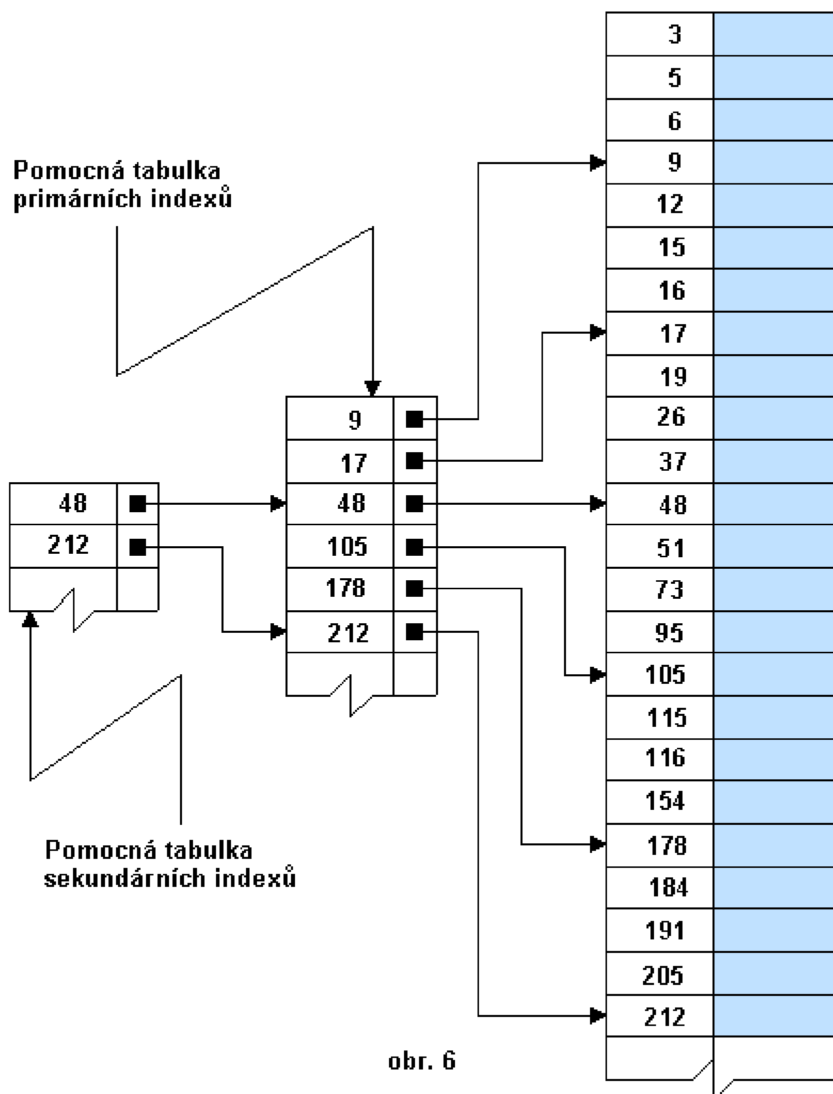
```
    VETA *binary_search ( char k[ ], VETA *tab, int n )   {

        VETA *d = &tab[0], *h = &tab[n-1], *str ;
        int     p ;

        while ( d <= h )    {
            str = d + ( h - d ) / 2 ;
            if ( ( p = strcmp ( str->klic, k ) ) < 0 )  d = str + 1 ;
            else     if ( p > 0 )  h = str - 1 ;
                     else        return ( str ) ;
        }
        return ( NULL ) ;
    }
```

vrací pointer na vyhledanou větu, resp. NULL, pokud věta se zadaným klíčem v tabulce není.

Zvětšení rychlosti vyhledávání u uspořádaných tabulek organizovaných jako pole však jde na úkor zmenšení rychlosti operací přidání a odebrání vět s daným klíčem. Při přidání nové věty do tabulky musíme podle zadaného klíče nejprve vyhledat místo pro vložení, které je nutno uvolnit posunem zbývajících vět, zatímco při odebrání věty se zadaným klíčem je nutné po vyhledání věty postupně uvolňovaná místa obsazovat následujícími větami (viz příklady na obr. 5). Proto také taková organizace tabulek je výhodná v případech, že se aktualizace tabulek (operace přidání nebo odebrání vět) vyskytuje jen velmi zřídka a převažuje operace vyhledávání.

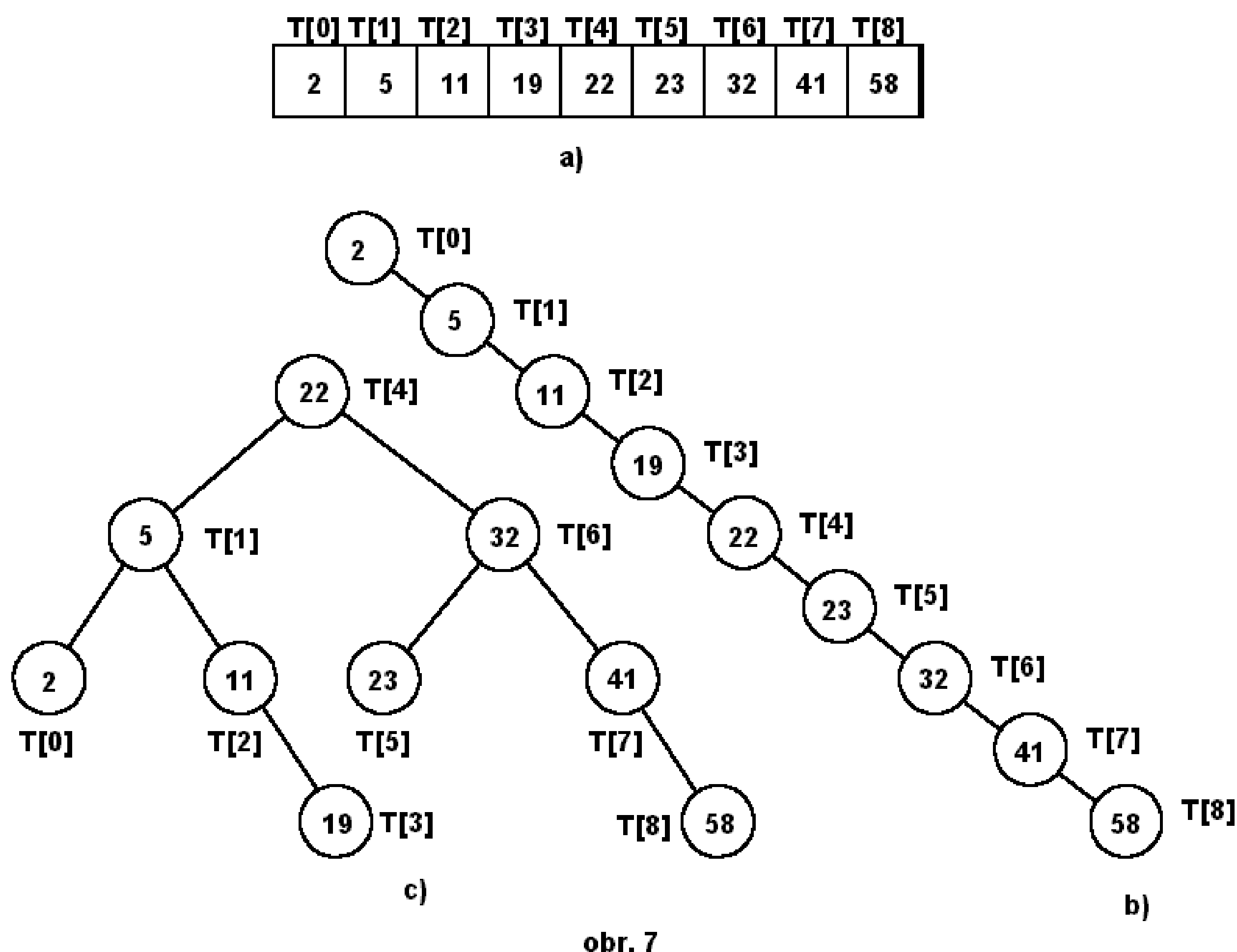


obr. 5



obr. 6

Jiná je technika vyhledávání v tzv. **tabulkách s indexsekvenčním výběrem**. Zde je k uspořádané tabulce vytvořena separátní **pomocná tabulka indexů**, přičemž každý prvek této pomocné tabulky obsahuje klíč k_p s ukazatelem na větu v uspořádané tabulce, jejíž klíč má hodnotu k_p. Prvky v pomocné tabulce jsou též uspořádány podle hodnot k_p. Zrychlení vyhledávání pak plyne ze skutečnosti, že nejprve se prohledají tabulky pomocných indexů, které jsou relativně krátké a tím se určí i relativně krátký úsek tabulky, ve kterém se následně vyhledá věta se zadaným klíčem. V příkladě na obr. 6 prvky pomocné tabulky primárních indexů referencují každou čtvrtou větu uspořádané tabulky. V případě rozsáhlých tabulek lze pro efektivní vyhledávání použít více pomocných tabulek indexů (viz pomocná tabulka sekundárních indexů na obr. 6). K nevýhodám uspořádaných tabulek uvedených již dříve je nutné v tomto případě navíc připočítat i větší nároky na paměť pro pomocné tabulky indexů.



obr. 7

Uspořádané tabulky lze organizovat též jako binární stromy. Je-li v nějakém uzlu takového binárního stromu obsažena věta s klíčem k, tak v levém podstromu tohoto uzlu jsou všechny klíče menší než k a v pravém podstromu jsou všechny klíče větší než k. Takových

binárních stromů však lze k dané posloupnosti vět vytvořit mnoho. Uvažujme příklad podle obr. 7. Podle obr. 7a nechť je uspořádaná tabulka organizována jako pole (pro zjednodušení jsou uvedeny jen celočíselné klíče). Jedna z možností, jak organizovat tuto tabulku jako binární strom je na obr. 7b (každý uzel má prázdný levý podstrom). Aktualizace takových tabulek je snadná - stačí změnit jen odpovídající ukazatel bez přesunů, ale vyhledávání bude pomalé, neboť se bude jednat o prostý sekvenční výběr. Vyberme proto prostřední prvek originálního pole jako kořen binárního stromu a všechny prvky v první polovině originálního pole (kde klíče jsou menší než u prostředního prvku) vytvářejí rekursivně levý podstrom, zatímco všechny prvky v druhé polovině originálního pole (kde jsou klíče naopak větší než u prostředního prvku) vytvářejí rekursivně pravý podstrom. Výsledkem bude tzv. **vyvážený binární strom**, u kterého lze efektivně implementovat jak vyhledávání, tak i aktualizaci.

Poznámka : Strom je vyvážený tehdy a jen tehdy, když se výšky dvou podstromů každého uzlu liší nejvíce o 1 (často se takové stromy nazývají po svých objevitelích jako AVL-stromy). Přitom výška stromů je maximální úroveň jeho listů (též hloubka stromu). Existuje také konvence, že výška prázdného stromu je definována jako -1. Snadno pak nahlédneme, že binární strom podle obr. 7c je vyvážený, což však neplatí pro binární strom podle obr. 7b.

Pokud bylo řečeno, že u uspořádaných tabulek organizovaných jako binární strom lze efektivně implementovat i operace aktualizace tabulek, tak to neznamená, že takové operace lze vyjádřit jednoduchými algoritmy. Právě naopak, a proto v takových případech lze odkázat na speciální literaturu, např. [3]. Potíž spočívá v tom, že po provedených aktualizacích tabulky by tato měla být organizována znovu jako vyvážený binární strom, aby tak byla zachována rychlost operací vyhledávání. Z tohoto důvodu se provádí transformace tak, aby průchody metodou inorder transformovaného a originálního stromu dávaly shodné výsledky a transformovaný strom byl vyvážený.

Cvičení

1. Uvažujme uspořádanou tabulku organizovanou jako kruhový seznam (jeho prvky tedy obsahují věty tabulky). Nechť proměnná *zac* stále referencuje ten prvek kruhového seznamu, se kterým je spojena věta s minimální hodnotou klíče. Další proměnná *vu* nechť při inicializaci referencuje tentýž prvek jako *zac*. Při každém vyhledávání se nastavuje *vu* na hodnotu referencující prvek seznamu obsahující vyhledanou větu, nebo v případě, že věta s udaným klíčem v tabulce není, tak *vu* se nastavuje na *zac*. Napište funkci pro takové vyhledávání v tabulce, přičemž tato funkce bude používat předané hodnoty *vu* pro zmenšení počtu porovnání při vyhledávání.

2. Uspořádaná tabulka je organizovaná jako pole. Promyslete si algoritmus pro vyhledávání užívající Fibonacciho čísla (tzv. Fibonacciho vyhledávání). Nechť fib(i) je funkce, vracející Fibonacciho čísla a pro jednoduchost uvažujme tabulku obsahující pouze celočíselné klíče :

```
unsigned  fibsearch ( int k, int *tab, int n )   {

        int j = 1 , p, f1, f2, t ;

        while ( fib(j++) < n+1 ) ;                          // Tyto dva řádky je
        p = n - fib(j-3) ; f1 = fib(j-3) ; f2 = fib(j-4) ;  //  vhodné modifikovat !!!!
        while ( k != tab[p] )
                if ( p < 0 || k > tab[p] )    {
                        if ( f1 == 1 ) return ( -1 ) ;
                        else   { p += f2 ; f1 -= f2 ; f2 -= f1 ; }
                }
                else   {
                        if ( !f2 )  return ( -1 ) ;
                        else  {  p -= f2 ; t = f1 - f2 ; f1 = f2 ; f2 = t ;  }
                }
        return ( p ) ;
}
```

Ověřte si funkci tohoto algoritmu a porovnejte jej s binárním vyhledáváním. Poté modifikujte inicializační část tohoto algoritmu za účelem efektivnějšího určení Fibonacciho čísel a zapište lepší verzi uvedené funkce.

3. Uvažujme indexsekvenční tabulku s jednou pomocnou tabulkou indexů. Ukazatelé v poli pomocné tabulky indexů odkazují vždy na 10 pozic originálního pole, přičemž 2 ze všech 10 pozic jsou neobsazené a určené pro eventuelní rozšiřování, resp. doplňování tabulky.

a) Proveďte inicializaci takové tabulky ze vstupního souboru, ve kterém jsou věty uspořádány vzestupně podle hodnot klíčů. Je dobré mít ve výplni přídavný člen, který indikuje, zda odpovídající pozice pole tabulky je obsazena větou či nikoliv. To pak umožňuje např. snadné odebrání z tabulky.

b) Implementujte základní operace s takovou tabulkou.

4. Vstupní soubor obsahuje věty uspořádané vzestupně podle hodnot klíčů. Napište program pro inicializaci uspořádané tabulky organizované jako vyvážený binární strom.

2.3 Rozptýlené tabulky

Rozptýlené tabulky představují určitý kompromis mezi tabulkami s přímým a sekvenčním výběrem. U takových tabulek máme opět k dispozici vhodnou transformační funkci, která každé hodnotě klíče *k* přiřazuje nějaký ukazatel na větu s udaným klíčem - index pole nebo adresu umístění věty. Rozptýlené tabulky si pak do určité míry zachovávají rychlost přímého výběru, jsou tedy organizovány za účelem minimalizace počtu operací potřebných pro vyhledávání v tabulce. Princip lze vysvětlit na následujícím příkladě.

Předpokládejme, že náš inventář může obsahovat nanejvýše 100 položek, z nichž každou budeme identifikovat čtyřmístným celým kladným číslem (to bude použito jako klíč věty). Kdybychom chtěli vytvořit tabulku s přímým výběrem organizovanou jako pole, tak by toto pole mělo 10 000 složek, z nichž by bylo využitých nanejvýše 100. Použijeme proto poslední dvě číslice klíče jako indexu složky pole obsahující větu s udaným klíčem. Jinými slovy, zavedeme transformační funkci ve tvaru

$$h(k) = k \mod 100 \quad , \quad \left[\, h(k) = k \ \% \ 100, \right]$$

která nám pro každou hodnotu klíče *k* vrací celé číslo z intervalu <0,99>. Pak ovšem stačí deklarovat pole o 100 složkách (viz obr. 8).

	KLÍČ	VÝPLŇ
0	5900	
1		
2	9402	
3		
…		
46	8446	
47	8147	
48		
49	2249	
…		
96	3596	
97	1097	
98		
99	0099	

obr. 8

Tato dobrá myšlenka má však jednu vadu. Může totiž dojít k případu, že pro dvě různé hodnoty klíčů k_i a k_j bude $h(k_i) = h(k_j)$. Pak říkáme, že dochází k tzv. **kolizím** (anglicky hash collision nebo hash clash a odtud také často používaný název pro tyto tabulky - **hash tables**). Poznamenejme hned, že volba transformační funkce je tím lepší, čím méně takových kolizí připouští. Tak např. podle obr. 8 by došlo ke kolizi při vkládání věty s klíčem 0547, protože odpovídající složka pole je již obsazena větou s klíčem 8147. Zbývá tedy ještě vyřešit problém kolizí, což je možné dvojím způsobem.

První možností je využít **metody otevřeného adresování**, kterou můžeme v našem příkladě interpretovat následovně. Předpokládejme, že do naší tabulky dle obr. 8 chceme vložit větu s klíčem 0547. Užitím naší transformační funkce dostaneme pozici (index) $h(0547) = 47$, ovšem tato pozice je již obsazena větou s klíčem 8147, a proto musíme větu s klíčem 0547 umístit na jinou pozici. Umístěme tedy větu s klíčem 0547 na nejbližší volnou pozici v poli - v našem příkladě 48. Pokud ovšem věta s klíčem 0547 byla takto do tabulky vložena, tak při následném vkládání věty s klíčem, kde hodnota transformační funkce bude 48 (např. věty s klíčem 6648) se použije pozice 50.

Pro vkládání či vyhledávání vět s daným klíčem tedy použijeme tzv. retransformační funkci $rh(i)$ - i je index. V našem příkladě jsme použili

$$rh(i) = (i+1)\,\mathrm{mod}\,100$$

Pokud tedy pozice $h(k)$ je obsazena větou s jiným klíčem, tak $rh[h(k)]$ udává další možnou pozici věty s udaným klíčem. Je-li i tato pozice pole obsazena větou s jiným klíčem, tak použijeme $rh\{rh[h(k)]\}$ atd.

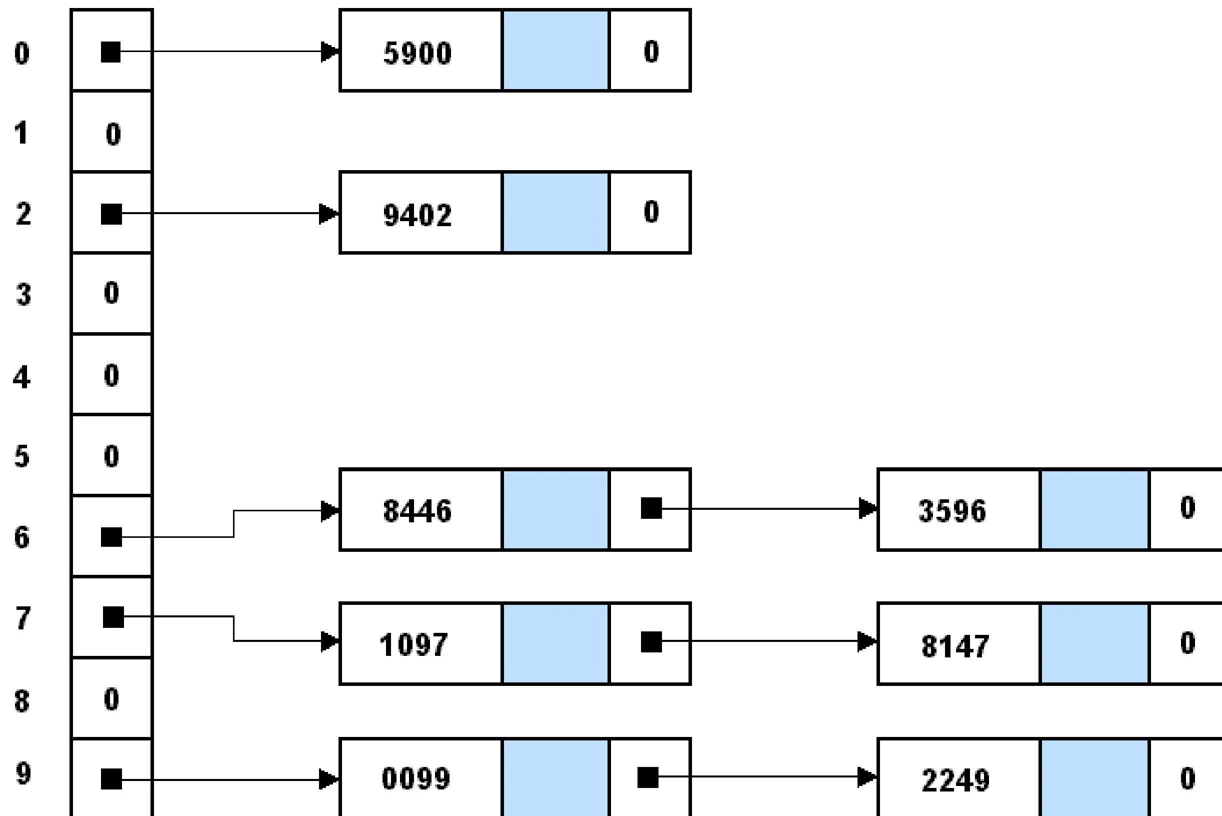
Druhou možností je tzv. **metoda řetězení**, jejíž princip je následující. Předpokládejme, že hodnoty transformační funkce jsou celá čísla z intervalu <0, n-1> a do pole o n složkách budeme ukládat ukazatele na začátky spojových seznamů, jejichž prvky budou věty s klíči se stejnými hodnotami transformační funkce. Zvolíme-li v našem příkladě transformační funkci ve tvaru

$$h(k) = k \mod 10$$

tak tabulku podle obr. 8 znázorníme obr. 9.

Zápis operací s takovou tabulkou velmi závisí na tom, pro jaký konkrétní případ tabulku používáme. Jako příklad (ve fiktivním kontextu) lze napsat vyjádření :

```
.....................
#define  M  10
typedef struct  {
        unsigned klic;
        VYPLN    vypln ;
} VETA ;
typedef struct prvek {
        VETA    veta ;
        struct prvek *spoj ;
} PRVEK ;
```



obr. 9

```
PRVEK *tab[M] ;
void init ( void ) { for ( int i = 0 ; i < M ; ) tab[i++] = NULL ; }
unsigned h ( unsigned k ) { return ( k % 1000 ) ; }
PRVEK *SearchAndInsert ( unsigned k , VYPLN v ) {
        PRVEK *p, *q = NULL ;
        int i = h(k) ;
        for ( p = tab[i] ; p ! = NULL ; q = p , p = p->spoj )
                if ( p->veta.klic = = k ) return ( p ) ;
        p = (PRVEK *) malloc ( sizeof ( PRVEK ) ) ;
        p->veta.klic = k ; p->veta.vypln = v ; p->spoj = NULL ;
        if ( q ) q->spoj = p
        else   tab[i] = p ;
        return ( p ) ;
}
```

Při použití rozptýlených tabulek je velmi důležitá vhodná volba transformační funkce. Kdyby byly všechny použité klíče známy předem (což se však stává zřídka), tak lze určit, která z transformačních funkcí je nejlepší. Většina obecných transformačních funkcí používá **metodu**

dělení, kdy je celočíselný klíč dělen rozsahem tabulky a zbytek po celočíselném dělení se bere jako hodnota transformační funkce :

$$h(k) = k \mod p$$

V uvedeném příkladě byla volena hodnota p = 100. Ovšem kdyby všechny použité hodnoty klíčů měly shodné dvě poslední číslice, tak $h(k_i) = h(k_j)$ pro každé k_i a k_j a pak takovou transformační funkci bychom nepoužili. Obecně se proto pro celočíselné klíče doporučuje volit p jako prvočíslo. Kromě již zmíněné metody dělení se používá i tzv. **metoda překrývání**, kdy se číslice klíče rozdělují do skupin, se kterými se nejčastěji provádí operace exkluzivního součtu (tj. 1, jsou-li oba bity různé, nebo 0, jsou-li oba bity shodné). Tak např. když je klíč vnitřně interpretován binárním řetězcem

010111001010110

a hodnota transformační funkce má být v binární representaci 5 dvojkových číslic, tak výsledkem exkluzivního součtu se 3 binárními řetězci

01011 , 10010 , 10110

bude 01111 (tj. dekadicky 15) jako hodnota transformační funkce pro tento klíč.

Neceločíselné klíče musí být nejprve konvertovány na celá čísla a pak lze vybrat vhodnou transformační funkci. Tak např. klíčem je znakový řetězec obsahující pouze písmena. Nechť A je interpretováno jako 01, čtrnácté písmeno N jako 14 atd. Pak klíč HELLO bude reprezentovat celé číslo 0805121215. Jakmile jsme klíč převedli na celočíselnou hodnotu, tak můžeme použít metod pro celočíselné klíče.

Cvičení

1. Uvažujme následující implementaci rozptýlené tabulky, kde případné kolize jsou řešeny metodou otevřeného adresování :

```
...............................
#define  M   100
..............................
typedef struct {
        unsigned  klic ;
        VYPLN     vypln ;
} VETA ;
............................
VETA tab[M] ;
...........................
```

```
#define  PRAZDNE  0xFFFFU
void init ( void ) { for ( int i = 0 ; i < M ; ) tab[i++].klic = PRAZDNE ; }
unsigned h ( unsigned k ) { return ( k % 100 ) ; }
unsigned rh ( unsigned k ) { return ( ( k + 2 ) % 100 ) ; }
unsigned search ( unsigned k, VYPLN v ) {
        int i ;
        for ( i = h(k) ; tab[i].klic ! = k && tab[i].klic ! = PRAZDNE ; i = rh(i) )  ;
        if ( tab[i].klic  = = PRAZDNE )  {
                tab[i].klic = k ; tab[i].vypln = v ;
        }
        return ( i ) ;
}
```

Přitom PRAZDNE je speciální celočíselná hodnota, která indikuje neobsazenou pozici v poli tab.

Určete, ve kterých případech nebude funkce search pracovat správně (neukončená smyčka) a proveďte odpovídající korektury.

2. Uvažujme retransformační funkci ve tvaru

$$rh(i) = (i + c) \mod n,$$

kde c je konstanta. Přesvědčte se, že pokud c a n jsou relativní prvočísla, tak hodnoty vrácené retransformační funkcí pokrývají všechny pozice tabulky.

3. U rozptýlené tabulky jsou případné kolize řešeny metodou řetězení. Oproti funkci uvedené v textu zapište zvlášť operaci vyhledávání a zvlášť operaci vkládání věty. Dále napište funkci pro odebrání věty z takové tabulky.

3. TŘÍDĚNÍ

Výsledkem třídění (angl. sorting) je nějakým způsobem uspořádaná množina prvků. Této programovací techniky se velmi často používá, především pro urychlení procesu vyhledávání (viz předchozí kapitola).

Nechť je dána nějaká posloupnost *n* prvků *r*(1)*,* *r*(2)*,* ....*,* *r*(*n*), nazývaných **věty**. Pro takové posloupnosti budeme používat termínu **soubor** (neztotožňujte zcela oba zavedené pojmy např. s datovými typy **file** a **record** známými z Pascalu). S každou větou *r*(*i*) je spojen klíč *k*(*i*), který je obvykle součástí věty (viz opět předchozí kapitola). U **souborů setříděných podle klíče** bude pro pro každé $i < j$ věta s klíčem *k*(i) předcházet větu s klíčem *k*(*j*). Budeme však připouštět, aby se v souboru vyskytovaly dvě nebo více vět se stejným klíčem.

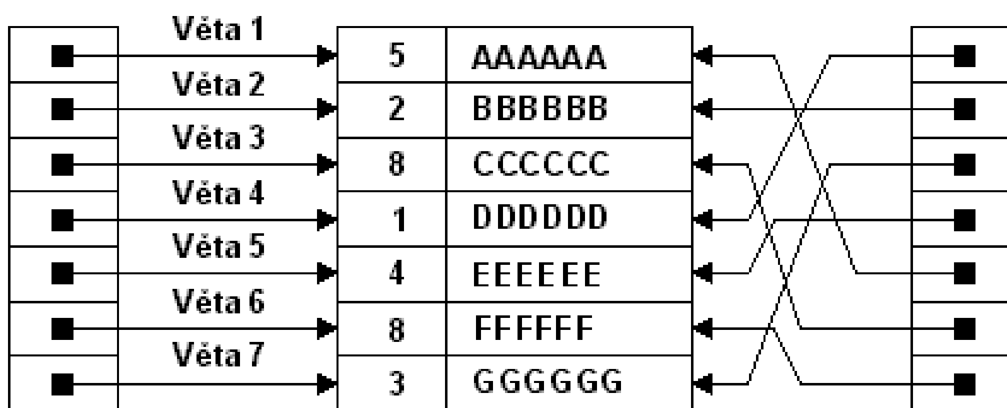
	Klíč	Výplň věty
Věta 1	5	AAAAAA
Věta 2	2	BBBBBB
Věta 3	8	CCCCCC
Věta 4	1	DDDDDD
Věta 5	4	EEEEEE
Věta 6	8	FFFFFF
Věta 7	3	GGGGGG

Originální soubor

Klíč	Výplň věty
1	DDDDDD
2	BBBBBB
3	GGGGGG
4	EEEEEE
5	AAAAAA
8	CCCCCC
8	FFFFFF

Setříděný soubor

a)



Originální tabulka ukazatelů

b)

Setříděná tabulka ukazatelů

obr. 10

Třídit můžeme buď věty samotné (viz obr. 10a), nebo jen nějakou vnější tabulku ukazatelů na věty (tzv. třídění adres - viz obr. 10b). Pro vysvětlení vlastního třídícího algoritmu budeme vždy zjednodušeně předpokládat, že každá věta obsahuje pouze jedinou položku , která tvoří třídící klíč. Ostatně případné další položky věty (výplň) se účastní třídění pouze pasivně (výplň se na vlastním třídění nepodílí). Je tudíž logické vysvětlovat různé metody třídění na poli s celočíselnými složkami.

3.1 Porovnání různých metod třídění

Při výběru metody třídění nelze brát zřetel pouze na složitost zdrojového textu programu (algoritmu třídění). Rozhodující význam má zpravidla strojový čas potřebný pro třídění a potom též požadavky na paměť. Oba tyto požadavky jsou obvykle protichůdné, přičemž větší význam mívá spotřeba strojového času. Je logické, že pokud je počet vět v souboru malý, tak různé složité algoritmy budované za účelem minimalizace spotřeby času a požadavků na paměť, budou horší než velmi jednoduché a ze všeobecného hlediska málo efektivní techniky třídění. Nebo, když by se měl program pro třídění provést pouze jednou (nebo jen několikrát) a programátor má k dispozici dostatek strojového času i paměti, tak asi nebude užitečné ztrácet čas implementací některé efektivní, ale složité metody třídění.

Když porovnáváme různé metody třídění z hlediska spotřeby strojového času, tak nemůže být rozhodující skutečný čas v nějakém konkrétním případě. Ten se bude měnit podle použitého počítače, použitím různých programů pro jednu a tu samou metodu třídění, ale i podle skladby originálního souboru. Proto se porovnávání různých metod třídění provádí následujícím způsobem.

Předpokládejme nejprve, že na základě provedené analýzy je dán čas t potřebný pro třídění jako funkce počtu vět n, dejme tomu ve tvaru

$$t(n) = 0{,}01n^2 + 10n$$

Tento čas nechť vyjadřuje počet časových jednotek potřebných pro setřídění souboru o velikosti n vět (viz údaje v tabulce 1). Všimněme si, že pro malá n je počet časových jednotek přibližně úměrný n, pro $n = 1000$ se oba členy výrazu $0.01\, n^2 + 10\, n$ podílejí na spotřebě času stejně, ovšem pro velká n roste spotřeba času zhruba s n^2. Pak říkáme, že čas potřebný pro třídění je $t(n) = O(n^2)$. Obecněji pokud bude $t(n) = O[g(n)]$ (čteme $t(n)$ je řádu O [$g(n)$]), tak

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} < konst$$

vyjadřuje srovnání dvou funkcí. Prakticky to též vyjadřuje, že pro velká n poroste $t(n)$ s $g(n)$. Když v našem příkladě bylo $t(n) = O(n^2)$, tak při velkých hodnotách n,když se n zvětší 2x, tak

$t(n)$ se zvětší 4x. To však pro malá n neplatí - viz uvedený příklad. Kdybychom totiž aplikovali stejnou metodu třídění na soubory velikosti řádově 10, tak $t(n)$ poroste velmi přibližně s n.

Tab. 1

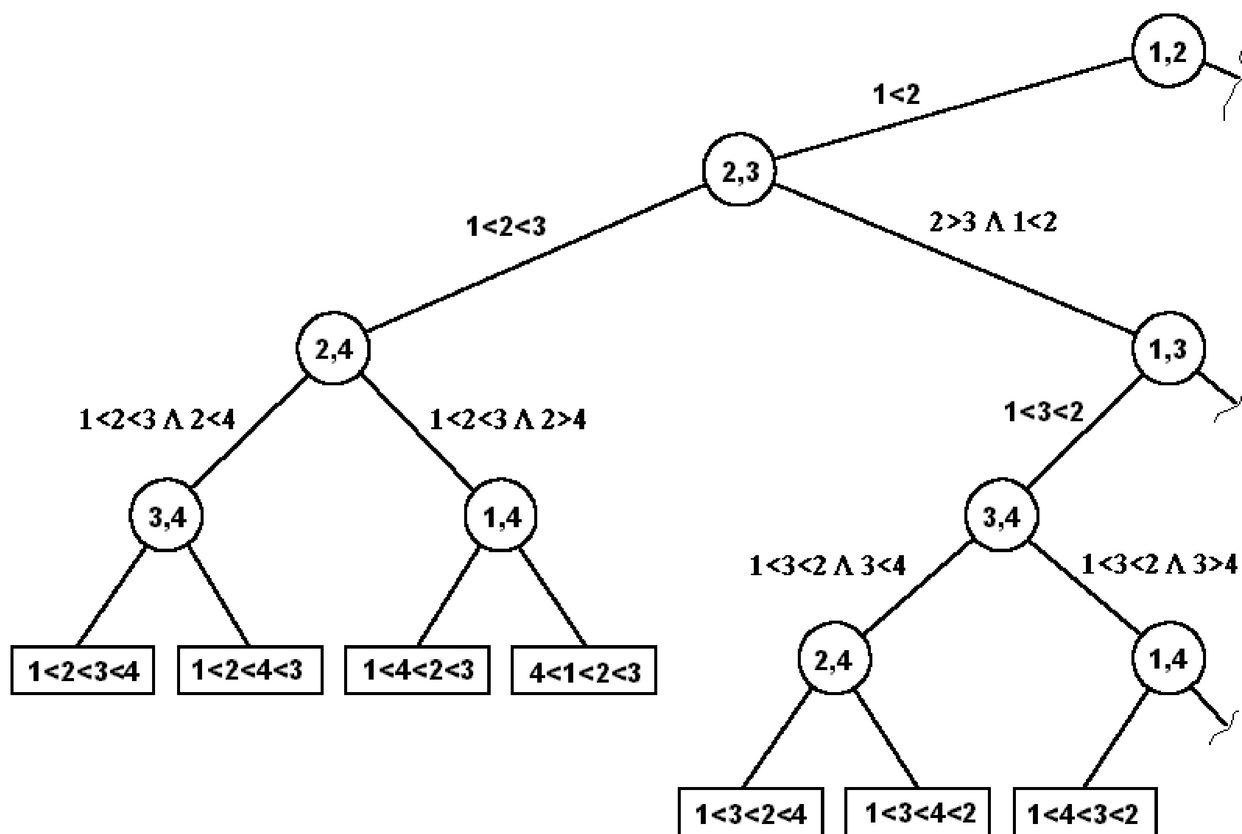
n	a = 0.01 n²	b = 10 n	a + b	(a + b) / n²
10	1	100	101	1.01
50	25	500	525	0.21
100	100	1 000	1 100	0.11
500	2 500	5 000	7 500	0.03
1 000	10 000	10 000	20 000	0.02
5 000	250 000	50 000	300 000	0.01
10 000	1 000 000	100 000	1 100 000	0.01
50 000	25 000 000	500 000	25 500 000	0.01
100 000	100 000 000	1 000 000	101 000 000	0.01
500 000	2 500 000 000	5 000 000	2 505 000 000	0.01

Zdá se, že u optimální techniky třídění by měl být čas $O(n)$. Bohužel však známé a prakticky používané techniky třídění vykazují $O(\ n \log n\ )$ až $O(n^2)$ (základ logaritmů není podstatný). Pro $O(\ n \log n\ )$ když n se zvětší 100x, tak $t(n)$ se pro velká n zvětší 200x, zatímco pro $O(n^2)$ to bude 10 000x. To však v žádném případě neznamená, že by měla být vždy použita technika třídění s $O(\ n \log n\ )$. Předně při malých hodnotách n se může třídící metoda chovat zcela jinak, a proto pro výběr techniky třídění bychom museli znát všechny členy závislosti $t(n)$, které mohou hrát dominantní roli při malých hodnotách n. Dále čas potřebný pro třídění silně závisí i na tom, jak vypadá originální soubor. Tak např. když je originální soubor již "téměř setříděn", může být danou technikou úplně setříděn v čase $O(n)$, zatímco aplikace stejné techniky třídění na originální soubor těch samých vět, jejichž posloupnost je však opačná, může vykazovat $O(n^2)$. Proto dříve uvedené tvrzení o času od $O(\ n \log n\ )$ do $O(n^2)$ je třeba chápat následovně. Neznáme takovou techniku třídění, která by vykazovala čas $O(n)$ pro jakékoli počáteční uspořádání originální posloupnosti vět. Pro nějaké (či nějaká) uspořádání originálního souboru (která však bývají víceméně patologická) může být čas $O(n)$, nicméně pro nějaké (nějaká) uspořádání originálního souboru bude čas třídění alespoň $O(\ n \log n\ )$. Tedy nejen velikost

originálního souboru podmiňuje efektivitu a tudíž i volbu metody třídění. Pokusme se nyní dopracovat k zaklínacímu *O (n log n)*.

Uvažujme nějakou metodu třídění založenou na porovnávání klíčů vždy dvou vět. Pak proces třídění můžeme znázornit pomocí striktně binárního stromu, kde každý uzel jež není listem reprezentuje porovnávání klíčů k_i a k_j. V případě, že $k_i < k_j$ pokračuje se dále levou větví, v opačném případě pravou větví (uvažujme nejdříve, že všechny klíče jsou navzájem různé). Např. pro $n = 4$ je část takového stromu na obr. 11. Porovnáme k_1 a k_2 a pro $k_1 < k_2$ porovnáme dále k_2 a k_3 . Pro $k_2 < k_3$ porovnáme dále k_2 a k_4 , pro $k_2 > k_4$ porovnáme k_1 a k_4 a pak pro $k_1 > k_4$ dospějeme k listu stromu, který reprezentuje setříděnou posloupnost vět : věta s klíčem k_4 , věta s klíčem k_1 , věta s klíčem k_2 a věta s klíčem k_3.

Poznámka : Nenechte se mýlit relacemi na obr. 11 - ve skutečnosti jde o hodnoty klíčů, ne o samotné číselné hodnoty ! Tak např. symbolický zápis 2 >3 Λ 1 < 2 čteme jako "$k_2 > k_3$ a současně $k_1 < k_2$".



obr. 11

Když v procesu třídění neprovádíme žádná redundantní porovnávání (tj. nikdy se neprovádí porovnávání k_i a k_j , pokud taková relace je již známa), tak takový strom musí mít *n!* listů. To dáno počtem všech permutací skupiny *n* různých prvků.

Je zřejmé, že na úrovni m (m = 1, 2, ...) striktně binárního stromu může být nanejvýše 2^m listů, tudíž hloubka stromu jako jeho maximální úroveň musí být alespoň $log_2(n!)$. Proto při aplikaci takové metody třídění lze uvést taková počáteční uspořádání originální posloupnosti, že počet porovnání bude alespoň $log_2(n!)$. Pomocí Stirlingova vzorce

$$n! = n^n e^{-n} \sqrt{2\pi n}(1 + \frac{\delta}{12n}), 0 < \delta_n < 1$$

pak již snadno dokážeme, že

$$\log_2(n!) = O(n \ \log \ n) .$$

Pokud připustíme, že v posloupnosti se mohou vyskytovat věty se shodnými klíči, tak výsledkem aplikace právě uvedených stromů bude setříděná posloupnost, ať již při rovnosti klíčů vybereme levou, nebo pravou větev.

Požadavky na paměť nehrají zpravidla takovou roli jako čas, jsou $O(n)$ až $O(n^2)$. Při vyšších požadavcích na paměť lze využít vnějších pamětí..

Metody třídění se obvykle rozdělují na dvě kategorie, a sice na **metody vnitřního a vnějšího třídění**. Vnitřní třídění se též nazývá tříděním polí, která jsou uložena ve velmi rychlých vnitřních pamětích počítačů. Pokud je však originální soubor natolik rozsáhlý, že se nevejde do vnitřní paměti, tak tříděné údaje jsou uchovávány ve vnějších pamětích a to vyžaduje i jiné, či modifikované metody třídění.

Cvičení

1. Ukažte, že pokud daná metoda třídění vykazuje $O(n \; log_2 \; n)$, vykazuje též $O (n \; log_{10} \; n)$ a naopak (obecněji, že na základě logaritmů nezáleží).
2. Čas potřebný pro třídění je dán vztahem $a \; n^2 + b \; n \; log_2 \; n$, kde a, b jsou dané konstanty. Ukažte, pro jaká n bude mít dominantní roli první člen a pro jaká n bude mít dominantní roli druhý člen (pochopitelně ve vztahu k a, b).
3. Říkáme, že metoda třídění je **stabilní**, když relativní pořadí vět se stejnými hodnotami klíčů zůstává v procesu třídění zachováno. Stabilita třídění je žádoucí zvláště tehdy, když věty jsou již uspořádané podle sekundárních klíčů.

Tak např. když aplikujeme stabilní metodu třídění na originální soubor

KLÍČ :	VÝPLŇ :
5	NOVAK
4	MLADY
5	NOVY
10	STARY

dostaneme setříděný soubor ve tvaru :

KLÍČ :	VÝPLŇ :
4	MLADY
5	NOVAK
5	NOVY
10	STARY

Vyberte si nějakou techniku třídění, která je Vám dosud známa a určete, zda je stabilní či nikoliv.

4. Je dána posloupnost n navzájem různých prvků. Nechť k je nejmenší celé číslo větší nebo rovno $n + log_2\ n - 2$. Ukažte, že pro vyhledání největšího a druhého největšího prvku posloupnosti n různých prvků stačí provést k porovnání.

5. V dalším textu bude provedena klasifikace metod třídění (třídění transpozicí, výběrem, ...). Mnohdy je však velmi obtížné zařadit nějakou konkrétní metodu třídění podle takové klasifikace. Pokuste se řešit toto a následující cvičení a po prostudování celé kapitoly o třídění se k těmto cvičením vraťte a pokuste se o svou klasifikaci uvedených algoritmů.

Napište funkci pro třídění posloupnosti $\{ x_i \}$ různých celočíselných hodnot. Použijme pomocné pole, jehož i-tá složka je počet prvků originální posloupnosti $\{ x_i \}$ menších nebo rovných x_i, např. :

i	1	2	3	4	5	6	7	8
x_i	5	-3	9	6	7	-2	1	0
pomocné pole	5	1	8	6	7	2	4	3
výstupní pole	-3	-2	0	1	5	6	7	9

6. Modifikujte řešení předchozí úlohy pro případ, kdy v originální posloupnosti se mohou vyskytovat shodné prvky. Určete poté, zda Vaše modifikace představuje stabilní metodu třídění, či nikoliv.

7. Nechť je opět dána nějaká posloupnost $\{ x_i \}$ celočíselných hodnot a nechť

$$A \leq X_i \leq B, i = 1,2,.........,n$$

Pokud uvažujeme $B - A + 1 << n$, tak pro třídění použijte pole, jehož složky udávají četnost výskytu x_i v originální posloupnosti. Tato metoda třídění se též nazývá **frekvenční třídění**.

3.2 Třídění transpozicí

Nejjednodušším algoritmem je třídění transpozicí sousedních vět v jejich posloupnosti. Tato technika se i v naší literatuře označuje jako **bubble sort**. U této metody se klíč *k(i)* porovnává s klíčem $k(i+1)$ a je-li $k(i)$ větší než $k(i+1)$, tak si věty $r(i)$ a $r(i+1)$ vymění svoje

místa. Tímto způsobem se prochází celá posloupnost opakovaně a tak dlouho, pokud se vyskytuje nesetříděnost.

Vlastní třídící algoritmus je demonstrován na třídění pole s celočíselnými složkami. Uvažujme např. následující posloupnost celých čísel :

25 57 48 37 12 92 86 33 .

V každém průchodu budeme postupně porovnávat čísla x_i a x_{i+1} (i = 1, 2, ..., 7) a pokud $x_i > x_{i+1}$, tak si obě čísla vymění svoje místa v posloupnosti. Po prvním průchodu tedy obdržíme tuto posloupnost :

25 48 37 12 57 86 33 (92) .

Při prvním průchodu bude největší číslo (92) již zaujímat správnou polohu (tj. stejnou jako v setříděné posloupnosti) - "vybublá" a odtud tedy i název této metody. Ve druhém průchodu stačí proto provést obecně n-2 porovnání, ve třetím n-3 atd. Nanejvýše je nutné provést n-1 takových průchodů (v posledním průchodu porovnat x_1 a x_2). To v našem příkladě dává tyto výsledky (počínaje druhým průchodem) :

25	37	12	48	57	33	(86	92)
25	12	37	48	33	(57	86	92)
12	25	37	33	(48	57	86	92)
12	25	33	(37	48	57	86	92)
12	25	(33	37	48	57	86	92)
12	(25	33	37	48	57	86	92)

Celkem je uvedeno 7 průchodů, ovšem poslední dva průchody byly zbytečné, neboť posloupnost již byla setříděna 4. průchodem. Proto za účelem eliminace zbytečných průchodů se při každém průchodu určuje, zda se provádí alespoň jedna transpozice a třídění se ukončí průchodem, ve kterém nebyla provedena žádná transpozice (v našem příkladě 5. průchodem). Implementace této metody v symbolickém jazyku je ponechána na čtenáři.

Tato technika třídění je relativně jednoduchá, obecně však málo efektivní. v prvním průchodu je třeba provést n-1 porovnání, ve druhém n-2, atd. Obecně je třeba provést p průchodů, $p \le n-1$. Celkový počet porovnání tedy bude

$$(n-1)+(n-2)+...............+(n-p)=\frac{1}{2}(2np-p^2-p)$$

Počet průchodů je řádu $O(n)$, tedy výsledek je řádu $O(n^2)$. Jistě, vynásobíme-li n nějakou konstantou c, vynásobí se čas konstantou menší než c^2 $(p \le n-1)$,ovšem navíc zřejmě při každém průchodu inicializujeme nějakou proměnnou indikující, že při tomto průchodu došlo k transpozici. Pochopitelně počet výměn nemůže být větší než počet porovnání. Pokud bychom takovou metodu aplikovali na originální soubor, který je již setříděn, tak se provede jen n-1 porovnání - $O(n)$. Výhodou této metody jsou malé požadavky na paměť.

Poznamenejme též, že počet průchodů nutných pro úplné setřídění je dán "největší vzdáleností pohybu" nějaké věty směrem k začátku souboru. Tak např. v dříve uvedeném příkladu posloupnosti celých čísel je v originální posloupnosti 33 jako x_8, v setříděné posloupnosti jako x_3 (po pátém průchodu). Počet průchodů lze snížit tak, že dva po sobě jdoucí průchody budou mít opačný směr (tj. zleva doprava, zprava doleva, atd.), čímž lze zvýšit rychlost třídění.

Obecně efektivnější je metoda třídění nazývaná **quicksort**, jejíž princip je následující. Ze souboru vět $r(1), r(2), \dots, r(n)$ vybereme nějakou větu [např. $r(1)$] a označíme ji ρ. Nově uspořádáme posloupnost vět tak, že věta ρ bude na *j-tém* místě této nové posloupnosti těch samých vět, přičemž budou dále splněny tyto podmínky v nově uspořádané posloupnosti :

1) $k(1), k(2), \dots, k(j-1)$ budou menší nebo rovny klíči vybrané věty ρ ,
2) $k(j+1), k(j+2), \dots, k(n)$ budou větší nebo rovny klíči vybrané věty ρ.

Věta ρ tudíž bude na místě, jaké má mít v úplně setříděném souboru. Stejný postup potom aplikujeme na posloupnosti vět $r(1), r(2), \dots, r(j-1)$ a na posloupnosti vět $r(j+1), r(j+2), \dots, r(n)$ atd. Snadno by tedy bylo možné zapsat tento algoritmus jako rekursivní funkci.

Celý postup lze opět demonstrovat na jednoduchém příkladě třídění posloupnosti celých čísel. Uvažujme opět originální posloupnost

25 57 48 37 12 92 86 33 .

Vybereme první prvek ($\rho = x_1 = 25$) a nově uspořádaná posloupnost bude

(12) 25 (57 48 37 92 86 33) .

Opakujme totéž pro posloupnost čísel 12 a posloupnost 57, 48, 37, 92, 86, 33. Posloupnost obsahující jeden prvek je již setříděna, tedy dostaneme

12 25 (57 48 37 92 86 33)

a po transpozicích ve druhé posloupnosti dostaneme :

12 25 (48 37 33) 57 (92 86) .

Postupně pak dostaneme :

12 25 (37 33) 48 57 (92 86) ,
12 25 (33) 37 48 57 (92 86) ,
12 25 33 37 48 57 (92 86) ,
12 25 33 37 48 57 (86) 92 ,
12 25 33 37 48 57 86 92 .

Výběr věty ρ není významný. Algoritmus demonstrovaný výše uvedeným příkladem zachovává po rozdělení posloupnosti vět do dvou dílčích posloupností relativní vzájemnou polohu vět. Tak např. 57 předchází 37 v originální posloupnosti i v dílčí posloupnosti 57, 48, 37, 92, 86, 33. Efektivnější je však provést nové uspořádání (rearrange) podle algoritmu demonstrovaného na naší posloupnosti celých čísel. Šipky odkazují na dva porovnávané prvky a

posouvají se střídavě po každé výměně, která je označena v odpovídajícím řádku hvězdičkou, a to tak dlouho, až referencují stejný prvek.

25	57	48	37	12	92	86	33	
↑							↑	
25	57	48	37	12	92	86	33	
↑						↑		
25	57	48	37	12	92	86	33	
↑					↑			
25	57	48	37	12	92	86	33	*
↑				↑				
12	57	48	37	25	92	86	33	*
	↑			↑				
12	25	48	37	57	92	86	33	
	↑		↑					
12	25	48	37	57	92	86	33	
	↑	↑						
12	25	48	37	57	92	86	33	
	↑							

12	25	48	37	57	92	86	33	*
		↑					↑	
12	25	33	37	57	92	86	48	
			↑				↑	
12	25	33	37	57	92	86	48	*
				↑			↑	
12	25	33	37	48	92	86	57	
				↑		↑		
12	25	33	37	48	92	86	57	
				↑	↑			
12	25	33	37	48	92	86	57	
				↑				

atd.

Techniku třídění quicksort lze efektivně aplikovat na tzv. úplně nesetříděné soubory - pak vykazuje $O(n \log n)$. Úplně nesetříděnými soubory máme na mysli takové, kdy v každé

částečně tříděné posloupnosti bude správná poloha ρ někde uprostřed této posloupnosti, resp. když jsou věty seřazeny v posloupnosti náhodně. Naopak její aplikace na úplně setříděné posloupnosti, resp. téměř setříděné posloupnosti je velmi neefektivní - čas bude $O(n^2)$. Později však bude zřejmější, že bublinové třídění (bubble sort) lze všeobecně považovat za nejhorší z uvedených metod třídění, zatímco quicksort se všeobecně považuje za nejlepší algoritmus vnitřního třídění.

Cvičení

1. Napište funkci pro metodu třídění bubble sort tak, aby dva po sobě následující průchody měly opačný směr.
2. Napište program používající funkci qsort z stdlib.h pro třídění, kdy klíče jsou celočíselné a znakové řetězce.
3. Modifikujte třídění metodou quicksort tak, že pokud nesetříděný úsek posloupnosti bude "již dostatečně krátký", tak na dotřídění takového úseku bude aplikována metoda bubble sort. Určením aktuálního času běhu programu určete takovou hranici pro aplikaci metody bubble sort, aby se taková smíšená strategie vyplatila.

3.3 Třídění výběrem

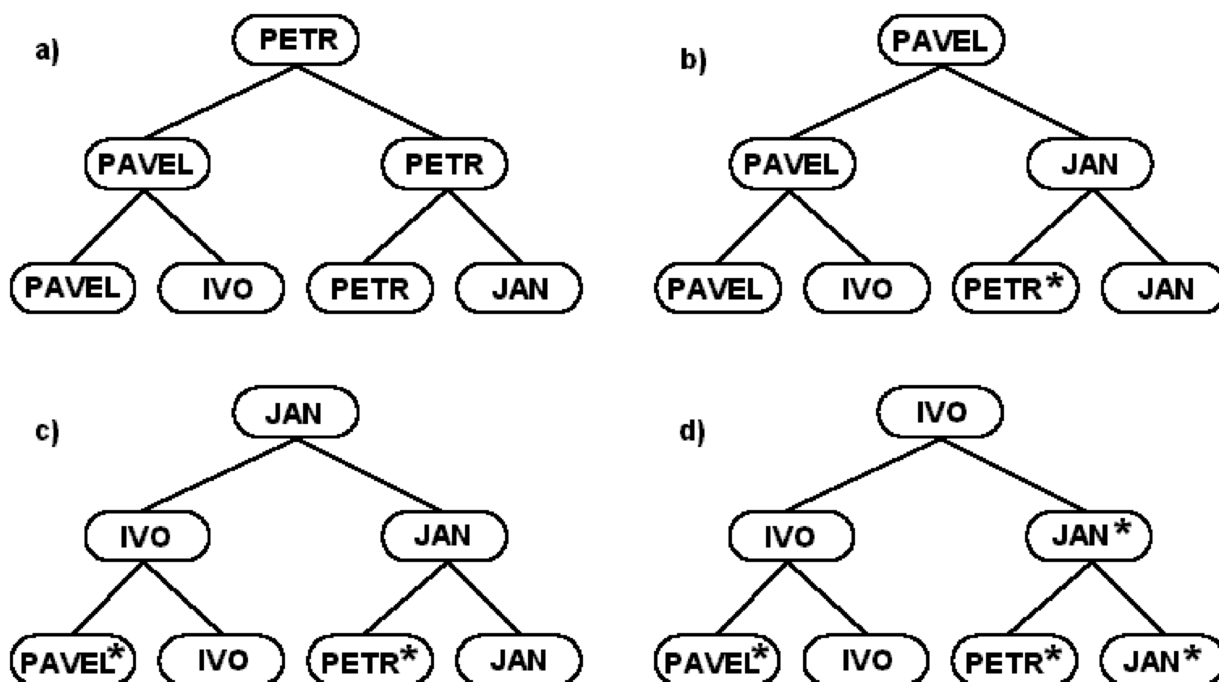
U této metody jsou postupně vybírány věty ze souboru a umísťovány do polohy, jakou mají mít v setříděném souboru. Nejjednodušší metoda tohoto druhu spočívá v postupném výběru věty s největším klíčem, druhým největším klíčem atd. a jejich umisťování do odpovídající polohy v setříděném souboru. Obsahuje-li tříděné pole n složek, pak jedna z mnoha možností zápisu jednoduchého třídění výběrem může vypadat takto :

```
void selectsort ( VETA *x, int p, int q ) {
      int i, j, k, max;
      VETA    y ;
      for ( i = q ; i > p; i-- ) {
            max = x[k=i].klic ;
            for ( j = i-1; j >= p; j-- )
                  if ( x[j].klic > max ) max = x[k=j].klic ;
            y = x[k] ; x[k] = x[i] ; x[i] = y ;
      }
}
```

Třídíme tedy úsek pole od indexu p do q za předpokladu, že $0 \le p < q \le n-1$.

Tuto metodu lze charakterizovat hodnotou $O(n^2)$ a to bez ohledu na to, jak je uspořádána originální posloupnost vět.

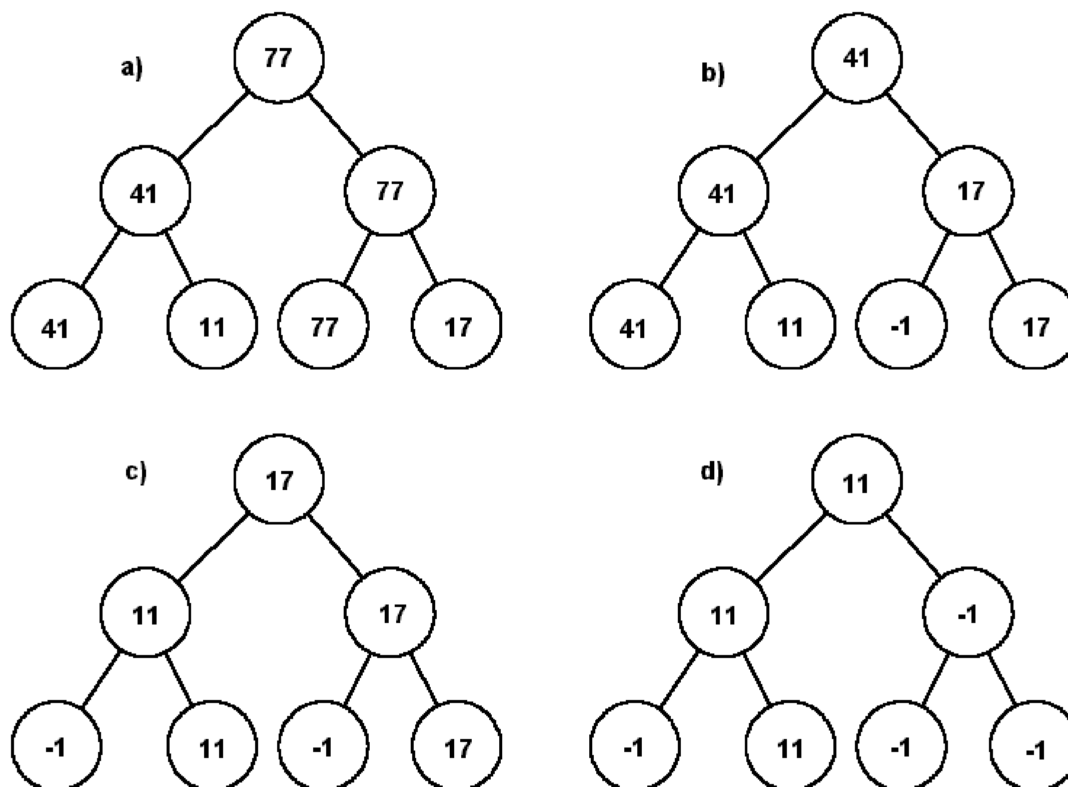
Jiné metody třídění výběrem používají binární stromy - viz vytvoření odpovídajícího binárního stromu pro jeho následný průchod metodou inorder. Další metoda třídění pomocí binárních stromů je obvykle označována jako **turnajové třídění** (tournament sort). Její princip lze vysvětlit na příkladě podle obr. 12. Řekněme, že chceme určit pořadí hráčů z množiny {PAVEL, IVO, PETR, JAN}. Výsledky turnaje nechť jsou znázorněny binárním stromem podle obr. 12a, z něhož vyplývá, že nejlepším hráčem je PETR. Ovšem druhým nejlepším hráčem nemusí být nutně PAVEL, mohl by to být i JAN, který měl tu smůlu, že "narazil" na PETRA hned v prvním kole. Chceme-li určit druhého nejlepšího hráče, označíme vítěze (PETR) hvězdičkou a ten se již na dalším průběhu nepodílí. Teprve vzájemné utkání mezi PAVLEM a JANEM stanoví, že druhým nejlepším hráčem je PAVEL (viz obr. 12b). Podobně vzájemné utkání mezi IVO a JANEM stanoví, že třetím nejlepším hráčem je JAN (obr. 12c). Můžeme si to představit tak, že originální množina {PAVEL, IVO, PETR, JAN} tvoří listy stromu, uzly, které nejsou listy obsahují vítěze utkání "levého" a "pravého syna" a postupným odebíráním obsahu kořene upravovaných binárních stromů získáme posloupnost {PETR, PAVEL, JAN, IVO}.



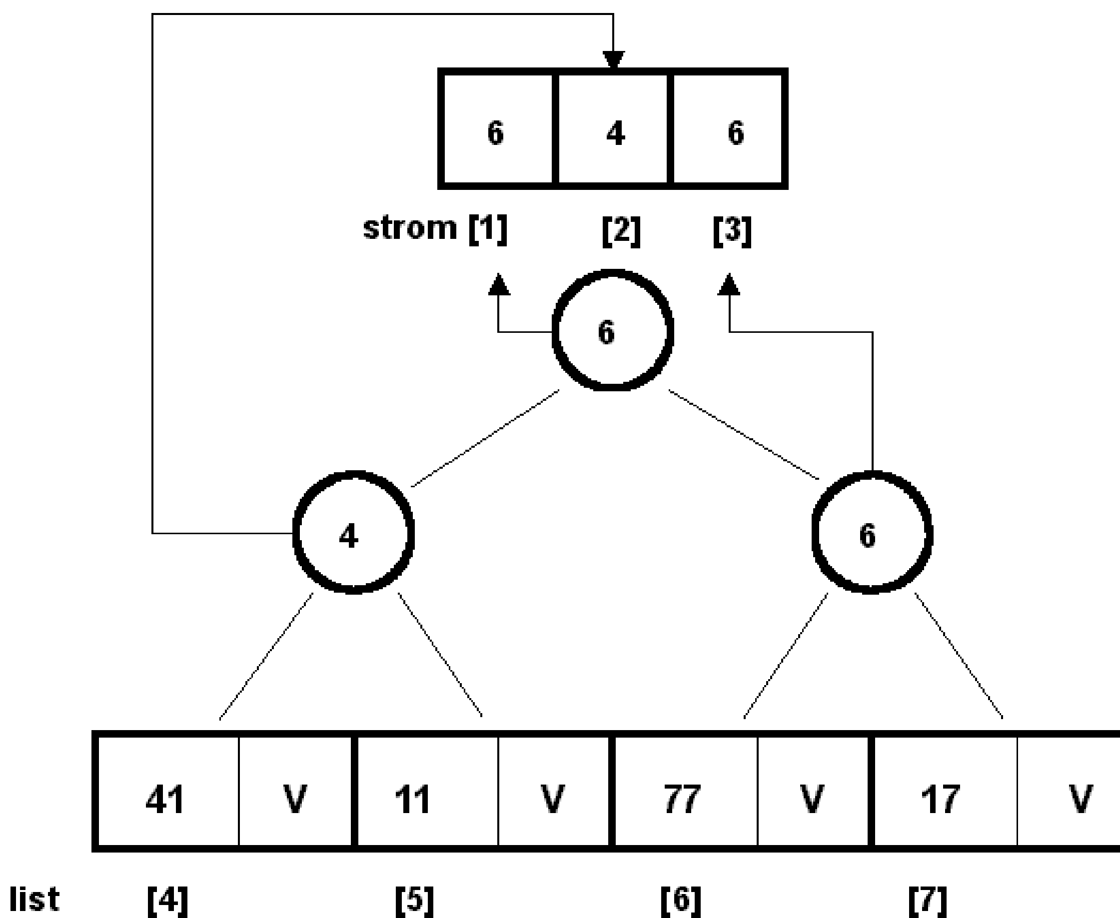
obr. 12

Na stejném principu je založeno turnajové třídění - viz obr. 13. Neuvažujme výplň vět, která se ostatně stejně na vlastním třídění aktivně nepodílí, ale jen klíče, které nechť jsou v našem příkladě celočíselné. Rozdíl oproti předchozímu příkladu spočívá v tom, že do uzlů, které nejsou listy zapíšeme větší z obou klíčů a místo označování hvězdičkou zapíšeme klíč, který je menší než nejmenší klíč v originální posloupnosti (např. v našem příkladu -1). Postupným odebíráním obsahu kořene stromů získáme uspořádanou posloupnost. Analogicky bychom postupovali např. v případě, kdy klíče jsou znakové řetězce sestávající ze znaků ASCII. Pak místo označování hvězdičkou zapíšeme klíč jako prázdný znak.

Použijme implementace binárního stromu pomocí pole a předpokládejme, že počet vět originálního souboru n je mocninou 2. V našem příkladě podle obr. 13 použijeme dvě pomocná pole list a strom podle obr. 14. Po inicializaci bude v poli list originální soubor, kdežto v poli strom indexy odkazující na složky pole list s větší hodnotou klíče.Všimněme si, že kořen našeho triviálního binárního stromu obsahuje 6 a list[strom[1]], tj. list[6] větu s největším klíčem. Dále pokud je i index nějaké složky pole list, tak $j = i/2$ (celočíselné dělení, $j = i \ div \ 2$) je index složky pole strom, obsahující příslušný uzel a pro index složky pole strom $j > 1$ je $k = j/2$ index složky téhož pole, obsahující uzel, jehož synem je daný uzel. Po odebrání kořene stromu dosadíme do složky pole list s indexem strom[1] fiktivní klíč a provedeme odpovídající úpravy (readjust) v poli strom.



obr. 13



obr. 14

Takto popsaný algoritmus má však dvě vady na kráse :

1. Deklarace dvou polí (list a strom) je nutná, snad jen kdybychom aplikovali tento algoritmus na třídění pole se složkami celočíselného typu, tak bychom mohli vystačit jen s jedním polem. Tak, jak byl algoritmus vysvětlen a graficky na jednoduchém příkladě znázorněn, tak jej lze přepsat do symbolického jazyku Pascal. Je však nutné si uvědomit, že v jazyku C má první složka pole index 0 a tudíž odpovídajícím způsobem je nutné transformovat hodnoty indexů na pravé straně uvedených výrazů.
2. Pokud počet vět originálního souboru není mocninou 2, tak zvolíme počet složek pole list jako nejmenší mocninu 2, větší než n (vytváření úplného binárního stromu) a složky pole list, které jsme zavedli navíc, budou obsahovat věty s klíčem menším než je nejmenší klíč v originálním souboru.

Lze ukázat, že tato technika třídění vykazuje $O(n \log n)$. Z hlediska nároků na paměť je zde třeba pomocných polí, přičemž celkový počet složek bude 2 m - 1 (m je nejmenší mocnina 2 větší nebo rovna velikosti originálního souboru n) - to je nepříjemné, zvláště když je např. n = 1025. Pak je třeba navíc zavádět fiktivní věty s klíčem menším než je nejmenší klíč v originálním souboru a v důsledku toho se provádí mnoho neužitečných porovnávání.

Právě zmíněné nedostatky jsou odstraněny u metody třídění pomocí binárních stromů, obvykle zvané **třídění hromadou (heapsort)**, kde pro každou větu originálního souboru je určen pouze jeden uzel binárního stromu a originální pole je současně užito i jako pracovní pole. Hromada (halda) je zpravidla definována jako binární strom, jehož uzly jsou ohodnoceny celými čísly a který má tyto vlastnosti :

➔ každý uzel hromady, který neleží na největší nebo druhé největší úrovni binárního stromu, má dva syny ,

➔ na druhé největší úrovni hromady následují zleva doprava nejprve uzly se dvěma syny, pak případně uzel mající jen levého syna a poté listy,

➔ ohodnocení uzlů hromady je provedeno tak, že ohodnocení libovolného uzlu není větší než ohodnocení kteréhokoliv z jeho následníků.

Uvidíme, že relaci ve třetí podmínce budeme dále interpretovat obráceně. To je však nepodstatné, zrovna tak místo od větších k menším můžeme postupovat od menších k větším prvkům.

Každý uzel bude obsahovat věty s klíčem, který je menší nebo roven klíči věty, obsažené v "otcovi" daného uzlu. Je-li při implementaci hromady pomocí pole x splněno, že

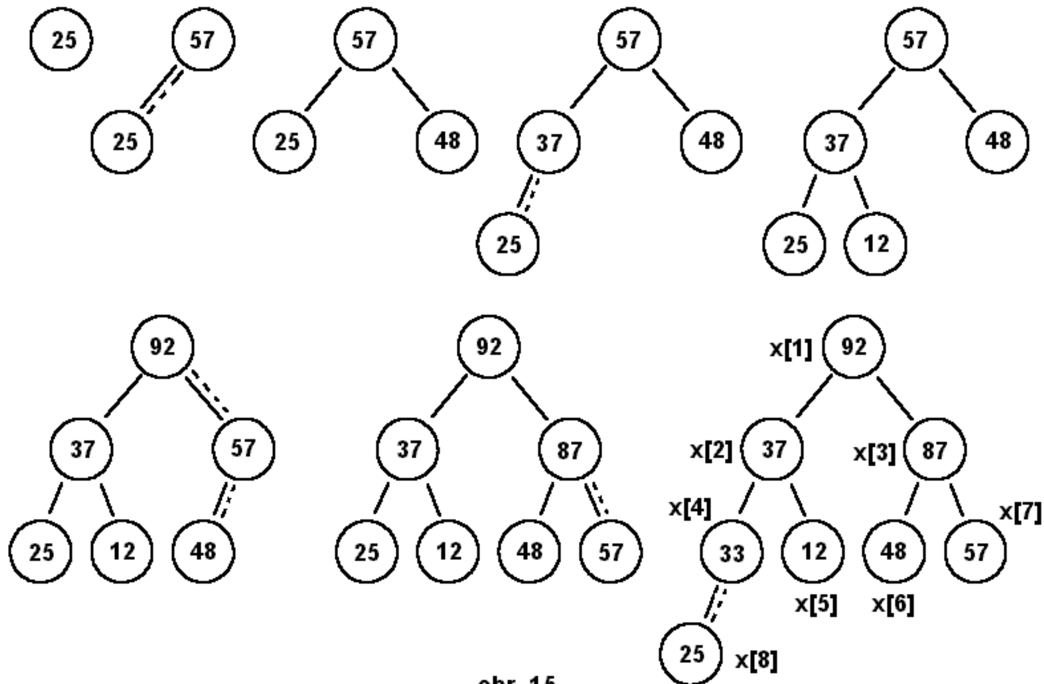
$$x[j].klic \le x[j/2].klic, 1 \le j/2 < j \le n,$$

tak kořen stromu (první element pole x) obsahuje větu s největším klíčem Uvažujeme tedy pole o n složkách, se kterými jsou sdruženy indexy 1 až n a symbol "/" znamená celočíselné dělení.

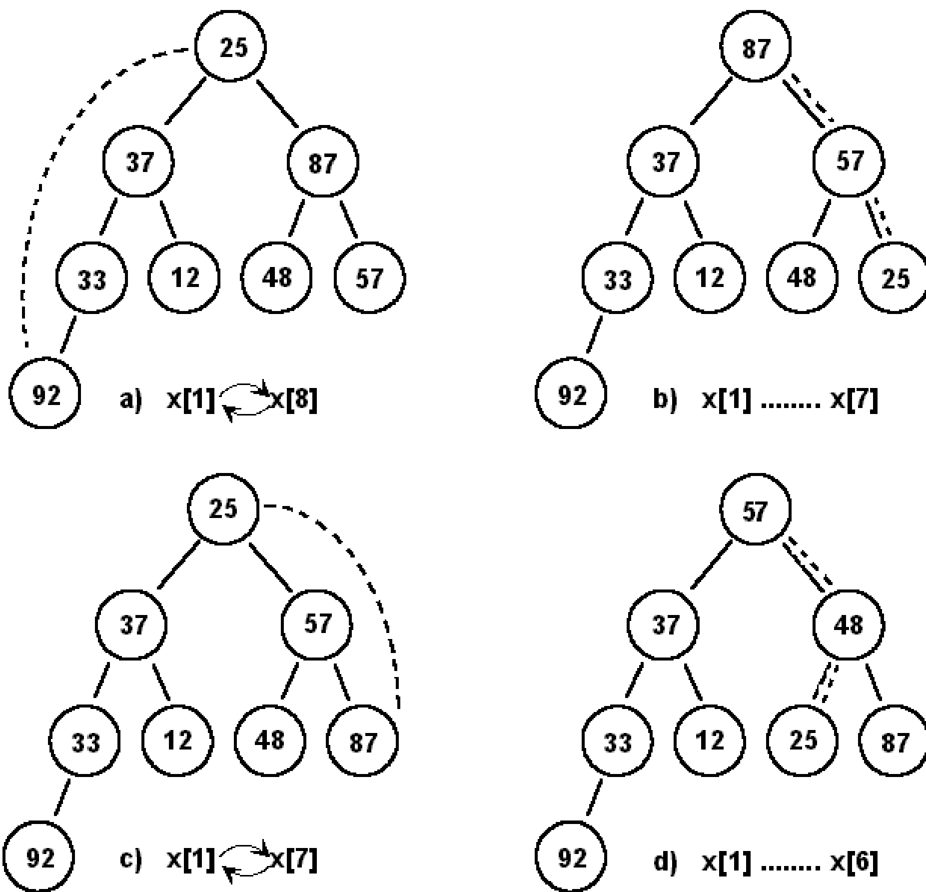
Jako příklad uvažujme celočíselné klíče a pro zjednodušení grafického vyjádření nebudeme zaznamenávat výplně vět. Nechť je tedy dána např. posloupnost

25 57 48 37 12 92 87 33 ,

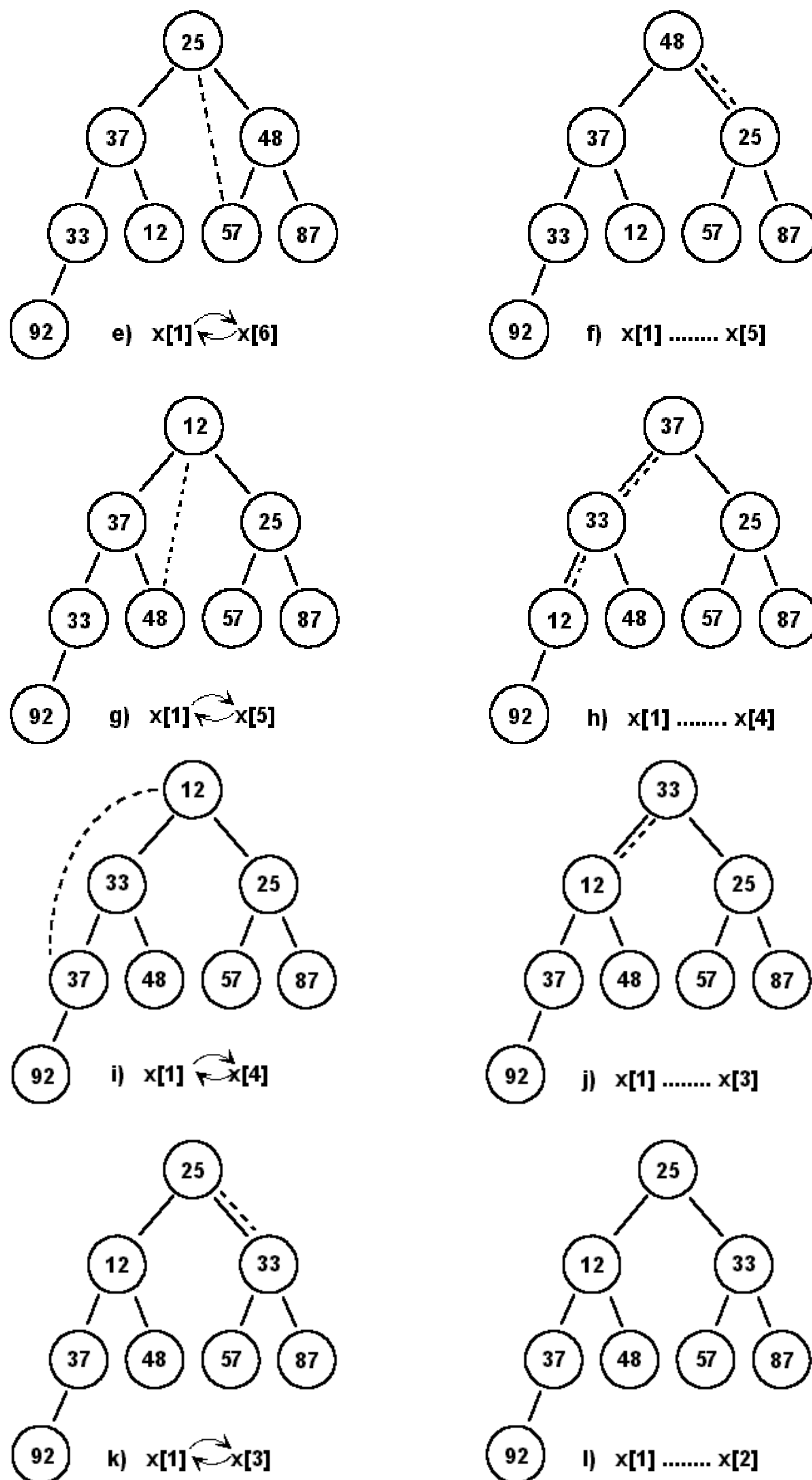
n = 8. Vytvoření hromady v poli x se složkami, kterým jsou přidruženy indexy 1 až 8 je znázorněno na obr. 15. Prováděné výměny jsou na tomto obrázku znázorněny čárkovaně. Zaměníme-li ve výsledku podle obr. 15 x[1] a x[8], bude složka x[8] obsahovat největší element (větu s největším klíčem) - viz obr. 16a. Následně však vytvoříme hromadu v poli x se složkami, kterým jsou přidruženy indexy od 1 do 7 (viz obr. 16b) a zaměníme-li nyní x[1] a x[7] (viz obr. 16c), bude složka x[7] obsahovat druhý největší element a celý postup opakujeme tak dlouho, až v poli x získáme setříděnou posloupnost - viz obr. 16m.



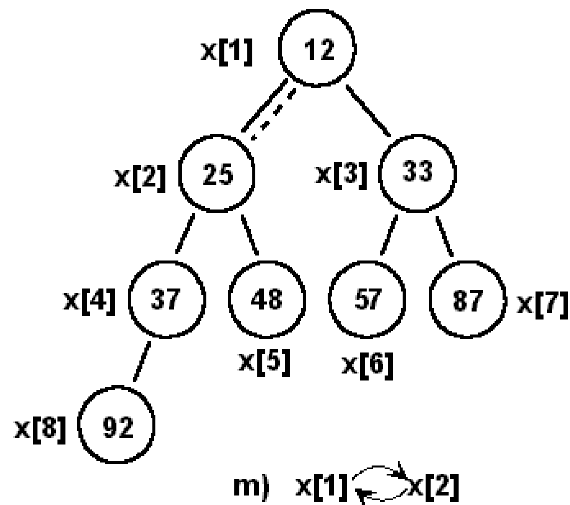
obr. 15



obr. 16



obr. 16 - pokrač.



obr. 16 - dokončení

Je asi logické, vysvětlit celý algoritmus na poli, jehož složky jsou sdruženy s indexy 1 až *n*. Pokud ovšem takový algoritmus implementujeme v jazyku C, musíme s *n* složkami pole sdružit indexy 0 až *n*-1 a dříve uvedenou relaci zapíšeme ve tvaru

$$x[j]klic \le x[(j-1)/2]klic, \ \ 0 \le (j-1)/2 < j \le n-1$$

Pro $n \ge 3$ lze uvedený postup zapsat jako funkci v symbolickém jazyku C např. takto :

```
int heapsort ( VETA *x, int n ) { /* n = počet prvků pole x */
    int i, j, k ;
    VETA y ;

    /* n musí být >= 3 !!! */
    if ( n < 3 ) return ( 1 ) ;
    /* Vytvořeni hromady --> */
    for ( k=1; k<n; k++ ) {
        i = k ; y = x[k] ; j = ( i - 1 ) / 2 ;
        for ( ;; ) {
            if ( y <= x[j] ) break ;
            x[i] = x[j] ; i = j ;
            if ( !i ) break ;
            j = ( j - 1 ) / 2 ;
        }
        x[i] = y ;
    }
```

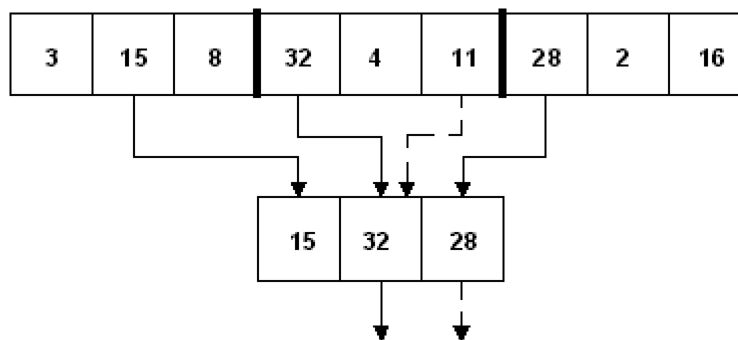
```
        /* Záměna x[k] <-> x[0], úprava --> */
        for ( k = n - 1; k > 0; k-- ) {
                y = x[k] ; x[k] =x[0] ;i = 0 ; j = 1 ;
                if ( x[2] > x[1]  && k - 1 >= 2 ) j = 2 ;
                while ( j <= k-1 )  {
                        if ( x[j] < y ) break ;
                        x[i] = x[j] ; i = j ; j = 2 * i + 1 ;
                        if ( j + 1 <= k - 1 )  if ( x[j+1] > x[j] ) j++ ;
                }
                x[i] = y ;
        }
        return ( 0 ) ;
}
```

Pro velká n je tato metoda efektivní [v nejhorším případě $O(n \log n)$], rovněž tak s ohledem na potřebnou paměť.

Cvičení

1. Ukažte, že potřebný počet porovnání u jednoduchého třídění výběrem je $(n^2-n)/2$ a nezávisí na uspořádání originální posloupnosti vět.

2. Tzv. **kvadratické třídění** výběrem se provádí následovně. Originální posloupnost n vět rozdělíme do $k = \sqrt{n}$ skupin, z nichž každá obsahuje k vět. V každé skupině nalezneme větu s největším klíčem a tyto věty umístíme do pomocného pole. V tomto pomocném poli najdeme větu s největším klíčem a to je věta s největším klíčem i v originální posloupnosti. Tuto větu nahradíme v pomocném poli větou ze stejné skupiny, ve které má nahrazující věta druhý největší klíč. Znovu najdeme v pomocném poli větu s největším klíčem a to bude věta s druhým největším klíčem v originální posloupnosti. Celý postup opakujeme až do úplného setřídění. Schematicky je tato idea znázorněna na obr. 17.



obr. 17

Zapište tuto metodu jako funkci v symbolickém jazyku.

3. Přepište v textu uvedenou funkci pro třídění hromadou pro případ třídění úseku pole od indexu p do q ($0 \le p, p+2 \le q \le n-1$, n je počet složek).

4. V poli x jsou podstromy s kořeny ve druhé a třetí složce pole organizovány jako hromady. Napište funkci, která v tomto poli vytvoří jedinou hromadu.

3.4 Třídění vkládáním

Princip této metody spočívá ve vkládání vět do již setříděného souboru vět. Příkladem jednoduchého vkládání je následující funkce (n je počet vět, třídí se úsek pole vět počínaje indexem p do indexu q) :

```
#define SUCCESS  0
#define ERROR      1
typedef struct  {
        int klic ;
        VYPLN  vypln ;
} VETA ;

int  insertsort ( VETA *x, int n, int p, int q )  {
        int i = p + 1, j ;
        VETA y ;

        if ( p < 0 || p > q || q > n-1 ) return ( ERROR ) ;
        while ( i <= q )  {
                j = i - 1 ; y = x[i++] ;
                while ( j >= p )   {
                   if ( y.klic >= x[j].klic ) break ;
                   x[j+1] = x[j--] ;
                } ;
                x[j+1] = y ;
        }
        return ( SUCCESS ) ;
}
```

Je-li originální soubor již vzestupně setříděn, bude tato jednoduchá metoda vykazovat čas $O(n)$, je-li naopak setříděn sestupně, tak $O(n^2)$.

Při třídění vět v poli podle **Shellova algoritmu** se provádí separátní třídění subpolí originálního pole, přičemž subpole obsahují každou *k*-tou složku originálního pole. Hodnotě *k* se říká inkrement a třídění každého subpole se provádí obvykle výše popsaným jednoduchým tříděním vkládáním. Tak např. pokud složkám pole x jsou přidruženy indexy od 1 do *n* a volíme-li $k = 5$, tak se provádí třídění následujících pěti subpolí :

subpole 1 : x[1] , x[6] , x[11] , ...
subpole 2 : x[2] , x[7] , x[12] , ...
subpole 3 : x[3] , x[8] , x[13] , ...
subpole 4 : x[4] , x[9] , x[14] , ...
subpole 5 : x[5] , x[10], x[15] , ...

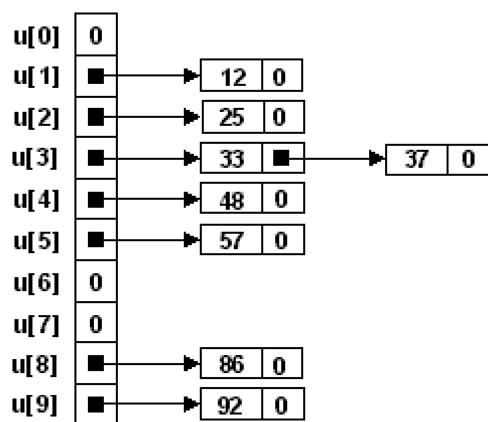
Po provedení třídění těchto subpolí se celý proces opakuje s menší hodnotou inkrementu *k*. Nakonec pro $k = 1$ subpole obsahuje celé pole, které bude setříděno. Jedna z mnoha možností zápisu Shellova třídění v jazyku C je následující :

```
typedef struct {        /*  Počet inkrementů a jejich hodnoty  */
        int pocet ;
        int *hod ;
} INCREMENT ;

typedef struct  {
        int       klic ;          /*   Uvažují se celočíselné klíče  */
        VYPLN  vypln ;
} VETA ;

void  shellsort ( VETA *x, int n, INCREMENT *incr )  {

        int i, j, k, L ;
        VETA y ;

        for ( i = 0; i < incr->pocet; i++ )  {
                /* Inkrement : */  k = incr->hod[i] ;
                for ( j = k; j < n ; j++ )  {
                        y = x[j] ; L = j - k ;
                        while ( L >= 0 )   {
```

```
                    if ( y.klic >= x[L].klic ) break ;
                    x[L+k] = x[L] ; L -= k ;
                } ;
                x[L+k] = y ;
            }
        }
    }
```

Poznamenejme, že při posledním průchodu ($k = 1$) se jedná o jednoduché třídění vkládáním. Toto jednoduché třídění vkládáním je efektivní, pokud je soubor téměř setříděn. Rovněž tak pro malá n je třídění s $O(n^2)$ často efektivnější než třídění s $O(n \log n)$, neboť programy jsou jednoduché a kromě porovnání a výměn se provádí málo jiných akcí. Naproti tomu u třídění s $O(n \log n)$ se všeobecně jedná o složitější programy, kdy se v každém průchodu provádí větší počet operací navíc, aby se redukoval objem operací v následujících průchodech. Protože první inkrement užitý při třídění podle Shellova algoritmu se volí velký, tak jednotlivá subpole jsou malá a jednoduché třídění vkládáním je tudíž dostatečně rychlé. Při každém dalším kroku je sice inkrement menší (tzn. subpole delší), ovšem každý předchozí krok posouvá úplný soubor směrem k téměř setříděnému souboru, na kterém je jednoduché třídění vkládáním též efektivní.

Čas potřebný pro třídění touto metodou závisí na počtu a hodnotách inkrementů. Pokud se použije vhodná posloupnost inkrementů, tak lze ukázat, že pro třídění podle Shellova algoritmu bude čas $O(n \log^2 n)$. Přitom vhodnou posloupností inkrementů se rozumí taková, ve které jsou použité inkrementy relativní prvočísla, tj. kromě 1 by neměly mít společného dělitele. To pak zaručuje, že každý následující průchod "promíchá" vzniklá subpole tak, že úplný soubor je při posledním průchodu téměř úplně setříděn.



obr. 18

Již dříve bylo řečeno, že je často obtížné jednoznačně klasifikovat nějakou metodu třídění. Zpravidla se však ke třídění vkládáním počítá i třídění vypočítáváním adres transformace klíče – **hashing**. U této metody třídění se na každý klíč aplikuje nějaká transformační funkce, jejíž výsledek určuje zařazení věty do nějaké podmnožiny originální množiny tříděných vět. Takové zařazení se pak zpravidla provádí metodou vkládání. Jako příklad uveďme třídění vět, jejichž klíče jsou kladná, dvoumístná celá čísla (výplně nebudeme uvažovat). Zvolíme-li transformační funkci

$h(k) = k/10$ (celočíselné dělení), tak z originální posloupnosti vět vytvoříme 10 lineárních uspořádaných seznamů (některé z nich mohou být prázdné). Tak např. pro posloupnost

25 57 48 37 12 92 86 33

bychom dostali schema podle obr. 18 (porovnejte s kapitolou o rozptýlených tabulkách).

Je vidět, že tato metoda vykazuje přídavné požadavky na dynamicky alokovanou paměť. Pokud by však originální posloupnost byla ve tvaru lineárního seznamu, tak takové přídavné požadavky odpadají.

Čas potřebný ke třídění závisí na volbě transformační funkce ve vztahu k n a konkrétním hodnotám klíčů. Bude mezi $O(n)$ a $O(n^2)$. Tak např. v našem příkladě když bude originální posloupnost deseti dvoumístných celých kladných čísel obsahovat čísla z každé dekády, tak bez ohledu na uspořádání originální posloupnosti bude čas blízko $O(n)$. Naopak pokud tato čísla budou pouze z jedné dekády, tak bude čas dán převážně časem na zařazování vět do uspořádaného seznamu.

Cvičení

1. Uvažujme třídění podle Shellova algoritmu.

- ♦ ukažte, že když s inkrementem k třídíme posloupnost vět, která již byla setříděna s inkrementem j, tak tato posloupnost zůstává setříděna i s inkrementem j,
- ♦ zamyslete se nad tím, proč kromě inkrementu 1 aplikovaného naposledy, by měly být ostatní inkrementy relativní prvočísla. Tak např. doporučované posloupnosti inkrementů (psané v opačném pořadí než jsou aplikovány) jsou

1 , 4 , 13 , 40 , 121 , ...

$(k_{i+1} = 3k_i + 1$, *počet inkrementů* $> \log_3 n < -1,>$ <*značí celou část čísla*),

nebo

1 , 3 , 7 , 15 , 31 , ...

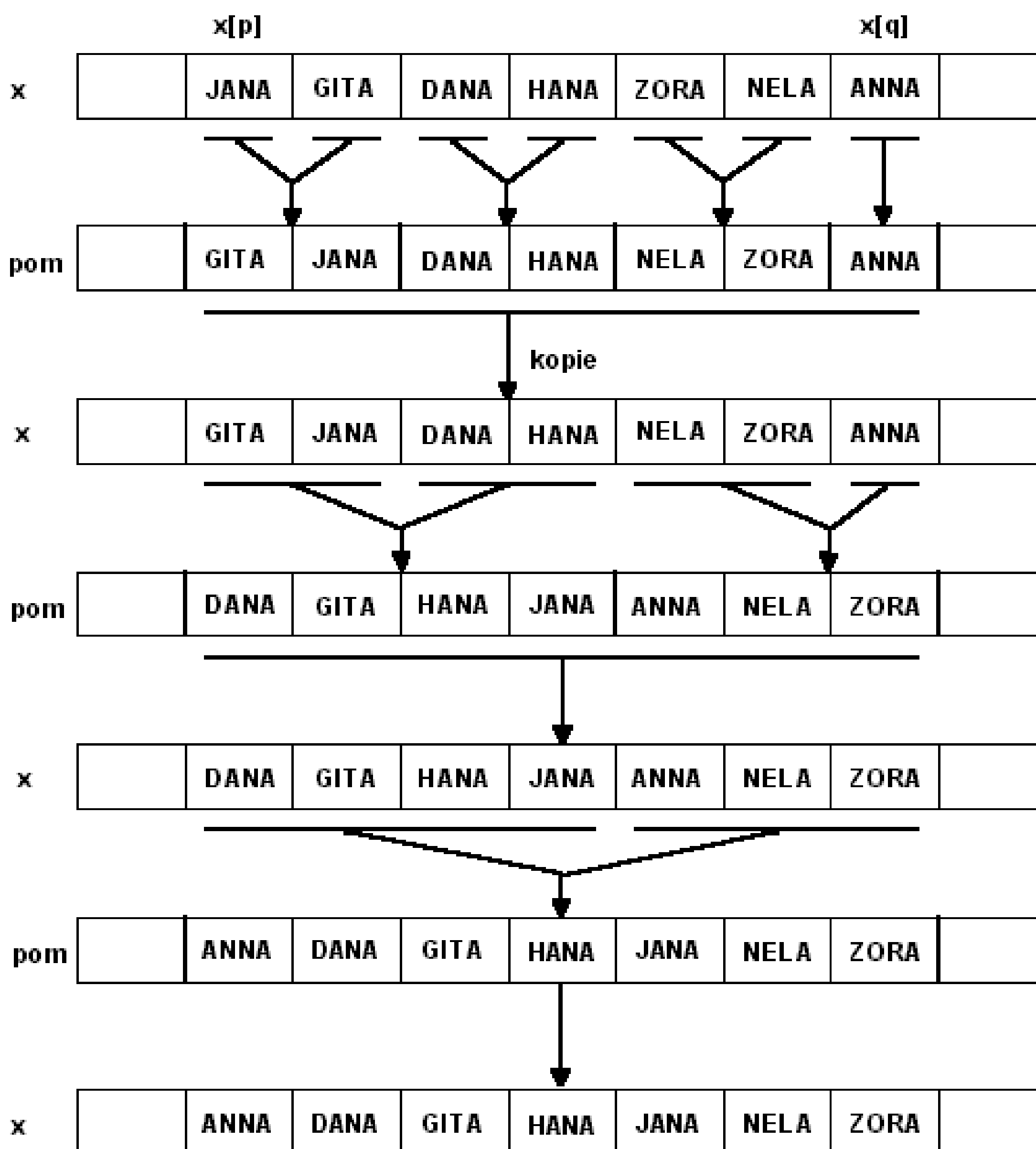
$(k_{i+1} = 2k_i + 1$, *počet inkrementů* $> \log_2 n < -1,>$

2. Napište funkci v jazyku C pro třídění podle Shellova algoritmu v případě, kdy klíče tříděných vět jsou znakové řetězce.

3. Originální posloupnost vět je dána ve tvaru lineárního seznamu implementovaného pomocí dynamických proměnných. Napište funkci v jazyku C, která z daného originálního seznamu vytvoří seznam uspořádaný podle velikostí klíčů vět metodou hashing.

3.5 Třídění sléváním a poziční třídění

Sléváním je označován proces, ve kterém se ze dvou nebo více souborů vytváří jiný setříděný soubor. Tato technika třídění bývá označována jako **merge sort** a její princip lze vysvětlit na následujícím příkladě (viz obr. 19). Tříděné věty jsou v poli x a pro zjednodušení grafického znázornění jsou na obr. 19 zaznamenány pouze klíče vět (v našem příkladě nechť jsou klíče vět znakové řetězce). Rozdělíme originální pole do subpolí o 1 větě (tato subpole jsou ovšem setříděna a priori, když obsahují pouze 1 větu) a sousední, disjunktní subpole sloučíme do



obr. 19

subpolí o 2 větách v pomocném poli pom (ev. doplníme zbývající větu do pole pom) a provedeme kopii pole pom do pole x. Tento proces postupně opakujeme pro slučování (slévání) subpolí dvojnásobné velikosti (ev. menších zbytků) až do úplného setřídění.

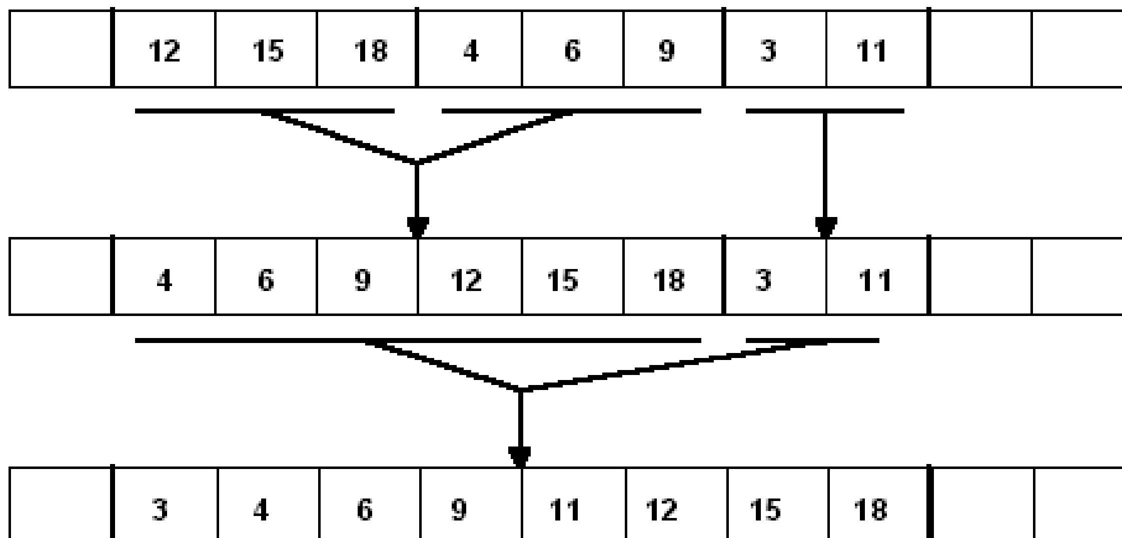
Pro uvedený příklad bychom mohli odpovídající funkci v jazyku C zapsat následovně (samozřejmě lze celý algoritmus vyjádřit lépe, o tom viz dále, ovšem modifikace jsou již ponechány na samostatné cvičení :

```
typedef struct {
        char   klic[5] ;
        VYPLN  vypln ;
} VETA ;
void  mergesort ( VETA *x, int p, int q )  {

        VETA *pom = (VETA *) malloc ( (q-p+1)*sizeof(VETA) ) ;
        int m = 1 ;            /* Začíná se slučováním subpolí obsahujících jednu větu */
        int d1, d2, h1, h2 ; /* Dolní (d) a horní (h) indexy slučovaných polí */
        int i, j, k ;

        while ( m < q-p+1 )  {
                k = 0 ; d1 = p ;
                while ( d1 + m <= q )  {
                        d2 = d1 + m ; h1 = d2 - 1 ; h2 = d2 + m - 1 ;
                        if ( h2 > q ) h2 = q ;
                         i = d1 ; j = d2 ;
                        /* Sléváni */
                         while ( i <= h1 && j <= h2 )
                            pom[k++] = ( strcmp(x[i].klic,x[j].klic) > 0 )?x[j++]:x[i++] ;
                        /* Umístěni zbylých vět : */
                        while ( i <= h1 ) pom[k++] = x[i++] ;
                        while ( j <= h2 ) pom[k++] = x[j++] ;
                        d1 = ++h2 ;
                }
                i = d1 ; /* Ev. doplnění */
                while ( k < q-p+1 ) pom[k++] = x[i++] ;
                /* Kopie */  for ( k = 0 ; k < q-p+1; k++ ) x[k+p] = pom[k] ;
                m *= 2 ; /* Následuje slévání polí dvojnásobné délky */
```

```
        }
        free ( pom ) ;
    }
```

Čas potřebný na toto třídění je O ($n \log n$), z hlediska požadavků na paměť je zde třeba deklarovat pomocné pole.

Oproti uvedenému vyjádření lze třídění sléváním vylepšit několika modifikacemi. Předně místo slévání subpolí v poli x do pomocného pole pom a kopírování pom na x lze v prvním kroku slévat subpole v poli x do pole pom a ve druhém kroku subpole v poli pom do pole x atd. Dále u uvedeného vyjádření dochází ke slévání subpolí stejné velikosti (s vyjímkou posledního). Lze však využít již stávajícího uspořádání a slévat nejdelší subpole, která již vykazují vzestupné uspořádání (viz příklad pro celočíselné klíče na obr. 20). Nutnost použití pomocného pole lze odstranit použitím spojových seznamů.



obr. 20

Stručně je třeba se zmínit též o tzv. **pozičním třídění** (angl. radix sort). Uvažujme m-místné celočíselné kladné klíče

$$a_{m-1}10^{m-1} + a_{m-2}10^{m-2} + \ldots\ldots\ldots a_1 10 + a_0$$

V prvním kroku roztřídíme věty do 10 skupin 0, 1, 2, ... ,9 podle hodnot desítkových číslic a_{m-1} klíče (v případě nutnosti uvažujeme i vedoucí nuly) tak, že všechny věty ve skupině 0 budou mít menší klíče než věty ve skupině 1, atd. Stejným způsobem roztřídíme věty ve druhém kroku podle hodnot desítkových číslic a_{m-2} klíče a to v každé skupině. Po m krocích bude posloupnost setříděna. Při alternativní metodě stejného druhu budeme naopak postupovat od číslic klíče počínaje nejnižším řádem až k nejvyššímu řádu a v každém kroku budeme vytvářet 10 front, přičemž zařazení věty do odpovídající fronty je dáno hodnotou aktuální desítkové číslice.

Vyprázdněním front získáme novou posloupnost vět, na kterou aplikujeme stejný postup podle desítkových číslic na řádu o 1 vyšším. Posloupnost bude setříděna v *m* krocích. Celý postup demonstrujme na posloupnosti dvoumístných celých čísel. Originální posloupnost :

92 32 81 21 89 47 52 36

1. krok (fronty vytvořené na bázi číslic na 0-tém řádu) :

	čelo		konec
fronta[0]			
fronta[1]	81		21
fronta[2]	92	32	52
fronta[3]			
fronta[4]			
fronta[5]			
fronta[6]	36		
fronat[7]	47		
fronta[8]			
fronta[9]	89		

Vyprázdnění fronty, nová posloupnost :

81 21 92 32 52 36 47 89

2. krok (fronty vytvořené na bázi číslic na prvním řádu) :

	čelo	konec
fronta[0]		
fronta[1]		
fronta[2]	21	
fronta[3]	32	36
fronta[4]	47	
fronta[5]	52	
fronta[6]		
fronta[7]		
fronta[8]	81	89
fronta[9]	92	

Setříděná posloupnost :

21 32 36 47 52 81 89 92 .

Implementace v symbolickém jazyku je ponechána na cvičení. Čas potřebný pro toto třídění závisí na počtu číslic klíčů (m) a počtu tříděných vět ($n = q - p + 1$). Přibližně bude čas $O(m.n)$, a proto takové třídění bude efektivní, pokud klíče nebudou příliš velké. Jsou zde i

přídavné požadavky na paměť. Pro velká m je výhodnější aplikovat toto třídění na číslice klíče na nejvyšších řádech a pak použít např. jednoduché vkládání pro dotřídění. V případě, že u většiny vět jsou číslice klíčů na nejvyšších řádech různé, tak se eliminují neefektivní kroky prováděné na bázi řádově nejnižších číslic klíčů.

Cvičení

1. Vylepšete funkci pro třídění v poli sléváním na základě modifikací původního algoritmu, popsaných v textu.
2. Uvažujme následující metodu slévání dvou setříděných polí x a y do pole z. Nechť n_x (n_y) je počet vět v poli x (y) a dále předpokládejme, že $n_x >> n_y$. Rozdělíme pole x do n_y+1 přibližně stejně dlouhých subpolí. Porovnáme klíč první věty subpole y (dejme tomu y[0].klic) s klíčem první věty druhého subpole x. Pokud je menší, tak binárním vyhledáváním v prvním subpoli x najdeme i takové, že

$$x[i].klic \le y[0].klic \le x[i+1].klic$$

Všechny složky prvního subpole až do x[i] a pak y[0] zkopírujeme do z. Tento proces opakujeme s y[1], y[2], ... , y[j] takovou, že klíč věty y[j] je větší než klíč první věty druhého subpole. Zkopírujeme zbývající věty prvního subpole a první větu druhého subpole do z. Pak porovnáme klíč věty y[j] s klíčem první věty třetího subpole atd. Tato metoda se nazývá **binární slévání**.

Zapište tento algoritmus v jazyku C.
3. Poziční třídění bude aplikováno jen na r desítkových číslic klíčů na nejvyšších řádech (r je daná konstanta). Pro dotřídění užijte metody jednoduchého vkládání. Zapište odpovídající funkci(e) v jazyku C.

3.6 Principy vnějšího třídění

Tříděných vět může být mnoho a nemusí se vejít do vnitřní paměti. Uvažujme proto nyní, že tříděné věty jsou organizovány jako sekvenční soubory.

Nejpoužívanější metodou vnějšího třídění je pak **slévání** (viz předchozí kapitola). Využijme stávajícího uspořádání vět originálního souboru a místo slévání dvou posloupností vět stejné délky (co do počtu vět) budeme slévat tzv. **běhy**. Běh je posloupnost *vět* $r(i)$*,* $r(i+1)$*, ... ,* $r(j)$, kde

$$k(l) \le k(k+1), l = i, i+1, ..., j-1,$$

$$k(i-1) > k(i)$$
$$k(j) > k(j+1)$$

Přitom $k(i)$ je klíč spojený s i-tou větou. Tak např. originální soubor (uvažujeme jen klíče celočíselného typu)

2 8 71, 49, 5 51 64, 15 29

obsahuje 4 běhy (oddělené čárkami).

Použijeme dva pomocné soubory *f1* a *f2* a v první fázi střídavě rozdělíme běhy obsažené v souboru *f* do souborů *f1* a *f2* (tzv. distribuční fáze) Ve druhé fázi sléváme běhy ze souborů *f1* a *f2* do souboru *f*. Obě fáze se opakují tak dlouho, až soubor *f* obsahuje právě jeden běh a je tudíž setříděn. Počet vstupních běhů v distribuční fázi může být větší než součet výstupních běhů obsažených v *f1* a *f2*, např. :

Originální posloupnost :

f	:	2	8,	5	7	12,	9	15,	13	14

Distribuční fáze :

f1	:	2	8	9	15	
f2	:	5	7	12	13	14

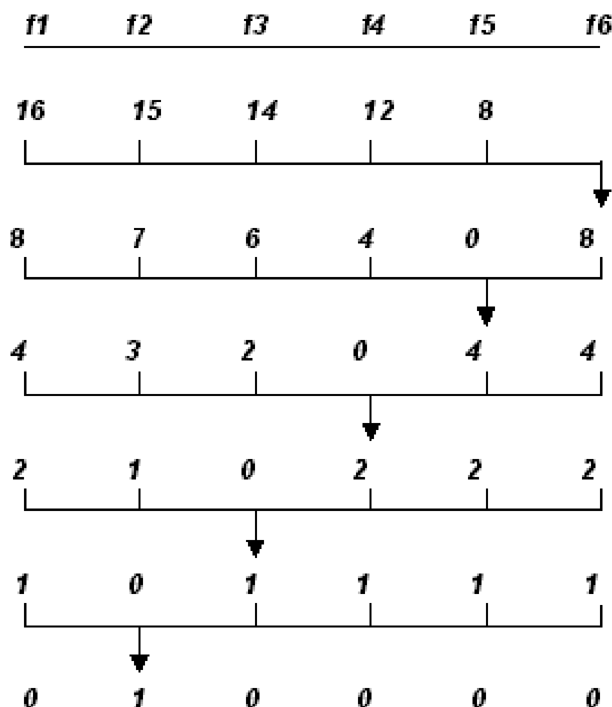
Fáze slévání :

f	:	2	5	7	8	9	12	13	14	15

V distribuční fázi byly jednotlivé běhy *f* střídavě přenášeny do *f1* a *f2* (1., 3., 5., ... běh na *f1*, 2., 4., 6., ... běh na *f2*), což může vést k tomu, že počty běhů obsažených v *f1* a *f2* jsou diametrálně odlišné. Pak ve fázi slévání po vyčerpání jednoho ze souborů *f1* a *f2* přeneseme zbývající běhy druhého souboru do souboru *f*.

V literatuře je popsaná technika třídění zpravidla označována jako **přirozené slévání** (natural merge). Její zápis v symbolickém jazyku je třeba vázat na konkrétní hardwarovou platformu a určitou implementaci jazyka C. Proto je v příloze pouze ukázka jednoho možného zápisu (určeno čtenáři k vylepšení) pod ULTRIX v.4.1 a vyšší na počítačích DEC.

Efektivnost takového třídění je dána počtem tzv. přechodů (každý přechod zahrnuje fázi distribuce a slévání), ve kterých se vždy provádí kopírování celé množiny údajů. Vylepšení lze proto dosáhnout distribucí běhů do více než dvou souborů (**vícecestné slévání**). Další vylepšení představuje tzv. **polyfázové třídění**, jehož princip lze vysvětlit na následujícím příkladě podle obr. 21.



obr. 21

Počáteční běhy jsou distribuovány na *f1* (16 běhů), *f2* (15 běhů), *f3* (14 běhů), *f4* (12 běhů) a *f5* (8 běhů). Originální soubor obsahoval tedy 65 běhů. 8 běhů z *f1, f2, f3, f4* a *f5* se slévá do *f6*, pak 4 běhy z *f1, f2, f3, f4 a f6* do *f5* atd., až *f2* bude obsahovat právě jeden běh. Aby algoritmus pracoval skutečně efektivně, tak je třeba zajistit správnou distribuci počátečních běhů. Podrobnosti o zmíněných metodách třídění najde čtenář v lit. [3].

Dále je třeba si uvědomit, že programy pro vnější třídění nemohou (obvykle) vyčerpat celou vnitřní paměť a je výhodné využít pro třídění i volnou část rychlé vnitřní paměti. Toho lze dosáhnou **kombinací metod vnitřního a vnějšího třídění**. Prakticky to znamená, že přizpůsobená technika vnitřního třídění se využívá pro distribuci počátečních běhů tak, že tyto běhy budou mít délku podle velikosti dostupné vnitřní paměti. Jako nejvhodnější metoda vnitřního třídění se ukazuje třídění hromadou.

Cvičení

1. Podle použité harwarové platformy a konkétní implementace jazyka C zapište svou funkci(e) pro třídění sekvenčního souboru sléváním.
2. Přesvědčte se, že při polyfázovém třídění s distribucí běhů na 2 soubory je ideální, když počet počátečních běhů je roven součtu dvou po sobě následujících Fibonacciho čísel, které udávají distribuci počátečních běhů. Tak např. je-li počet počátečních běhů roven 21, tak ideální

distribuce těchto běhů je 13 a 8. V případě, když počet počátečních běhů není roven tomuto počtu, bylo by třeba simulovat existenci hypotetických prázdných běhů, které spolu se skutečnými dávají ideální počet. Při distribuci běhů má více souborů se pak totéž týká Fibonacciho čísel vyšších řádů.

Literatura

[1] MÜLLER, K. : Programovací jazyky. Praha, ČVUT 1981.

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

[3] WITRH, N. : Algoritmy a štruktúry údajov. Bratislava, Alfa 1989.

Příloha

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
/*  Ev.  pro  SIZE a typ VETA např. -->
#include <fmerge.h>
*/

#define SIZE  2000

typedef struct {
        int    klic ;
        char   vypln[5] ;          /*  Například .....              */
} VETA ;

typedef struct {
        int   dscr ;               /* Deskriptor souboru            */
        VETA  *buffer ;            /* Vyrovnavací paměť             */
        int   i ;                  /* Aktuální věta                 */
        int   n ;                  /* Počet vět                     */
} BUFFER ;

static int EOR ;                   /* Indikátor konce běhu          */
static int pocet ;                 /* Počet běhů                    */
static BUFFER *f1, *f2, *f ;


/* Zkopíruj větu z x do y a přiřaď hodnotu proměnné EOR  */

void kopievety ( BUFFER *x, BUFFER *y ) {
```

```
        y->buffer[y->i] = x->buffer[x->i++]  ;

        f ( x->i == x->n )  {
                x->n = read ( x->dscr, (char *)x->buffer, SIZE*sizeof(VETA) ) ;
                x->n /= sizeof(VETA) ; x->i = 0 ;
        }
        EOR = ( x->n == 0 )  ? 1 : y->buffer[y->i].klic > x->buffer[x->i].klic ;
        y->i++ ;
        if ( y->i == SIZE ) {
                write ( y->dscr, (char *)y->buffer, SIZE*sizeof(VETA)) ;
                y->i = 0 ;
        }
}



/* Zkopíruj běh z  x  do  y */

void kopiebehu ( BUFFER *x, BUFFER *y )  {

        do kopievety ( x, y ); while ( !EOR ) ;
}

/* Distribuce běhů */

void distribuce ( void )  {

        do  {
                kopiebehu ( f, f1 ) ;
                if ( f->n != 0 ) kopiebehu ( f, f2 ) ;
        } while ( f->n != 0 ) ;
        write ( f1->dscr, (char *)f1->buffer, f1->i*sizeof(VETA) ) ;
        write ( f2->dscr, (char *)f2->buffer, f2->i*sizeof(VETA) ) ;
}
```

```
/* Slévani běhu */

void slevanibehu ( void )  {

        if ( f1->buffer[f1->i].klic < f2->buffer[f2->i].klic ) {
                kopievety ( f1, f ) ;
                if ( EOR ) { kopiebehu ( f2, f ) ; pocet++ ; }
        }
        else {
                kopievety ( f2, f ) ;
                if ( EOR ) { kopiebehu ( f1, f ) ; pocet++ ; }
        }
}

/* Slévání z  f1 a  f2  do  f */

void slevani ( void ) {

        while ( f1->n != 0 && f2->n != 0 ) slevanibehu () ;
        while ( f1->n != 0 ) {  kopiebehu ( f1, f ) ; pocet++ ;  }
        while ( f2->n != 0 ) {  kopiebehu ( f2, f ) ; pocet++ ;  }
        write ( f->dscr, (char *)f->buffer, f->i*sizeof(VETA) ) ;
}

int prirozeneslevani ( char *s ,                /* Tříděný originální soubor        */
                       char *s1, char *s2       /* Pomocné soubory                  */)  {


    f  = (BUFFER *) malloc ( sizeof ( BUFFER ) ) ;
    f1 = (BUFFER *) malloc ( sizeof ( BUFFER ) ) ;
    f2 = (BUFFER *) malloc ( sizeof ( BUFFER ) ) ;
    f->buffer = (VETA *) malloc ( SIZE * sizeof ( VETA ) ) ;
    f1->buffer= (VETA *) malloc ( SIZE * sizeof ( VETA ) ) ;
    f2->buffer= (VETA *) malloc ( SIZE * sizeof ( VETA ) ) ;
    f1->dscr=  creat ( s1, 0600 ) ;
    f2->dscr = creat ( s2, 0600 ) ;
```

```
    if ( (f->dscr = open ( s, O_RDWR )) < 0 ) return (-1) ;
    do {
        lseek ( f->dscr, 0L, SEEK_SET ) ;
        f->n = read ( f->dscr, (char *)f->buffer, SIZE*sizeof(VETA)) ;
        f->n /= sizeof(VETA) ; f->i = 0 ;
        f1->dscr = open ( s1, O_RDWR | O_TRUNC ) ;
        f1->i = 0 ;
        f2->dscr=  open ( s2, O_RDWR | O_TRUNC ) ;
        f2->i = 0 ;
        distribuce () ;
        lseek ( f->dscr, 0L, SEEK_SET ) ; f->i = 0 ;
        lseek ( f1->dscr, 0L, SEEK_SET ) ; lseek ( f2->dscr, 0L, SEEK_SET ) ;
        f1->n = read ( f1->dscr, (char *)f1->buffer, SIZE*sizeof(VETA)) ;
        f1->n /= sizeof(VETA) ; f1->i = 0 ;
        f2->n = read ( f2->dscr, (char *)f2->buffer, SIZE*sizeof(VETA)) ;
        f2->n /= sizeof(VETA) ; f2->i = 0 ;
        pocet = 0 ; slevani () ;
        close ( f1->dscr ) ; close ( f2->dscr ) ;
    } while ( pocet != 1 ) ;
    close ( f->dscr ) ;
    remove ( s1 ) ; remove ( s2 ) ;
    free( f->buffer ) ; free( f1->buffer ) ; free ( f2->buffer ) ;
    free ( f ) ; free ( f1 ) ; free ( f2 ) ;
    return (0) ;
}
```