

3. Sortări

3.1. Conceptul de sortare

- **Obiectivele** fundamentale ale acestui capitol sunt:
 - (1) Furnizarea unui **set extins de exemple** referitoare la utilizarea structurilor de date introduse în capitolul 1.
 - (2) Evidențierea **influenței** profunde pe care adoptarea unei anumite **structuri** o are asupra **algoritmului** care o utilizează și asupra **tehnicilor de programare** care implementează algoritmul respectiv.
- **Sortarea** este domeniul ideal al **studiului**:
 - (1) **Construcției** algoritmilor.
 - (2) **Performanțelor** algoritmilor.
 - (3) **Avantajelor** și **dezavantajelor** unor algoritmi față de alții în accepțiunea unei aplicații concrete.
 - (4) **Tehnicilor de programare** aferente diferiților algoritmi.
- Prin **sortare** se înțelege în general **ordonarea** unei mulțimi de elemente, cu scopul de a facilita **căutarea ulterioară** a unui element dat.
 - Sortarea este o activitate **fundamentală** cu caracter **universal**.
 - Spre exemplu în cartea de telefoane, în dicționare, în depozitele de mărfuri și în general în orice situație în care trebuiesc căutate și regăsite obiecte, sortarea este prezentă.
- În cadrul acestui capitol se presupune că sortarea se referă la anumite elemente care au o structură articol definită după cum urmează [3.1.a]:

```
typedef struct {
    int cheie;
    ... alte câmpuri;
} tip element; /*3.1.a*/
```

- Câmpul `cheie` precizat, poate fi neesențial din punctul de vedere al informației înregistrate în articol, partea esențială a informației fiind conținută în celelalte câmpuri.
- Din punctul de vedere al **sortării** însă, `cheie` este cel mai important câmp întrucât este valabilă următoarea **definiție a sortării**.

- Fiind dat un șir de elemente aparținând tipul mai sus definit:

$$a_1, a_2, \dots, a_n$$

- Prin **sortare** se înțelege **permutarea** elementelor șirului într-o anumită ordine:

$$a_{k1}, a_{k2}, \dots, a_{kn}$$

- Astfel încât șirul cheilor să devină **monoton crescător**, cu alte cuvinte să avem

$$a_{k1}.cheie \leq a_{k2}.cheie \leq \dots \leq a_{kn}.cheie$$

- Tipul câmpului `cheie` se presupune a fi **întreg** pentru o înțelegere mai facilă, în realitate el poate fi însă orice tip **scalar**.
- O **metodă de sortare** se spune că este **stabilă** dacă după sortare, ordinea relativă a elementelor cu chei egale coincide cu cea inițială
 - Această stabilitate este esențială în special în cazul în care se execută sortarea după mai multe chei.
- În cazul **operației de sortare**, **dependența** dintre **algoritmul** care realizează sortarea și **structura de date** prelucrată este profundă.
- Din acest motiv **metodele de sortare** sunt clasificate în **două mari categorii** după cum elementele de sortat:
 - (1) Sunt înregistrate ca și **tablouri** în **memoria centrală** a sistemului de calcul, ceea ce conduce la **sortarea tablourilor** numită **sortare internă**.
 - (2) Sunt înregistrate într-o **memorie externă**, ceea ce conduce la **sortarea fișierelor** (secvențelor) numită și **sortare externă**.

3.2. Sortarea tablourilor

- Tablourile se înregistrează în **memoria centrală** a sistemelor de calcul, motiv pentru care **sortarea tablourilor** se mai numește și **sortare internă**.
- **Cerința fundamentală** care se formulează față de metodele de sortare a tablourilor se referă la **utilizarea cât mai economică a zonei de memorie disponibile**.

- Din acest motive pentru început, prezintă interes numai algoritmi care realizează **sortarea "in situ"**, adică chiar în zona de memorie alocată tabloului.
- Pornind de la această restricție, în continuare algoritmi vor fi clasificați în funcție de eficiența lor, respectiv în funcție de timpul de execuție pe care îl necesită.
- **Aprecierea cantitativă** a eficienței unui **algoritm de sortare** se realizează prin intermediul unor **indicatori specifici**.
 - (1) Un prim indicator este **numărul comparațiilor de chei** notat cu **C**, pe care le execută algoritmul în vederea sortării.
 - (2) Un alt indicator este **numărul de atribuiri de elemente**, respectiv **numărul de mișcări de elemente** executate de algoritm, notat cu **M**.
 - Ambii indicatori depind de numărul total n al elementelor care trebuie sortate.
- În cazul unor algoritmi de sortare simpli bazați pe așa-zisele **metode directe de sortare** atât C cât și M sunt proporționali cu n^2 adică sunt $O(n^2)$.
- Există însă și **metode avansate de sortare**, care au o complexitate mult mai mare și în cazul cărora indicatorii C și M sunt de ordinul lui $n \cdot \log_2 n$ ($O(n \cdot \log_2 n)$).
 - Raportul $n^2 / (n \cdot \log_2 n)$, care ilustrează câștigul de eficiență realizat de acești algoritmi, este aproximativ egal cu 10 pentru $n = 64$, respectiv 100 pentru $n = 1000$.
- Cu toate că ameliorarea este substanțială, **metodele de sortare directe** prezintă interes din următoarele **motive**:
 - (1) Sunt foarte potrivite pentru explicitarea principiilor majore ale sortării.
 - (2) Procedurile care le implementează sunt scurte și relativ ușor de înțeles.
 - (3) Deși metodele avansate necesită mai puține operații, aceste operații sunt mult mai complexe în detaliile lor, respectiv metodele directe se dovedesc a fi superioare celor avansate pentru valori mici ale lui n .
 - (4) Reprezintă punctul de pornire pentru metodele de sortare avansate.
- Metodele de sortare care realizează sortarea **"in situ"** se pot clasifica în trei mai categorii:
 - (1) Sortarea prin **inserție**.
 - (2) Sortarea prin **selecție**.
 - (3) Sortarea prin **interschimbare**.
- În prezentarea acestor metode se va lucra cu tipul element definit anterior, precum și cu următoarele notații [3.2.a].

```
/*definire număr elemente de sortat*/
#define n ...
```

```

/*definire structură element*/
typedef struct tip_element {
    int cheie;
    ... alte câmpuri;
} tip_element;

/*definire tablou de sortat*/
tip_element a[n];
-----

```

3.2.1. Sortarea prin inserție

- Această metodă este larg utilizată de jucătorii de cărți.
 - Elementele (cărțile) sunt în mod conceptual divizate într-o **secvență destinație** $a_1 \dots a_{i-1}$ și într-o **secvență sursă** $a_i \dots a_n$.
 - În fiecare pas începând cu $i=2$, elementul i al tabloului (care este de fapt primul element al secvenței sursă), este luat și transferat în secvența destinație prin **inserarea** sa la locul potrivit.
 - Se incrementează i și se reia ciclul.
- Astfel la început se sortează primele două elemente, apoi primele trei elemente și așa mai departe.
- Se face precizarea că în pasul i , primele $i-1$ elemente sunt deja sortate, astfel încât sortarea constă numai în a insera elementul $a[i]$ la locul potrivit într-o secvență deja sortată.
- În termeni formali, acest algoritm este precizat în secvența [3.2.1.a].

```

-----
/*Sortarea prin inserție*/
tip_element a[n];

pentru i=2 to n
    *inserează a[i] la locul potrivit în a[1]...a[i];
    □
/*pentru*/
-----

```

- **Selectarea** locului în care trebuie inserat $a[i]$ se face parcurgând **secvența destinație** deja sortată $a[1], \dots, a[i-1]$ de la dreapta la stânga și comparând elementul de inserat $a[i]$ cu elementele secvenței. În acest scop se utilizează indicatorul j .
 - **Simultan** cu parcurgerea, se realizează și **deplasarea spre dreapta** cu o poziție a fiecărui element testat până în momentul îndeplinirii condiției de oprire.
 - În acest mod se face loc în tablou elementului care trebuie inserat.

- Oprirea parcurgerii se realizează pe primul element $a[j]$ care are cheia mai mică sau egală cu $a[i]$.
- Dacă un astfel de element $a[j]$ nu există, oprirea se realizează pe $a[1]$ adică pe prima poziție (Fig.3.2.1.a).

Sortarea prin inserție

Tabloul a:

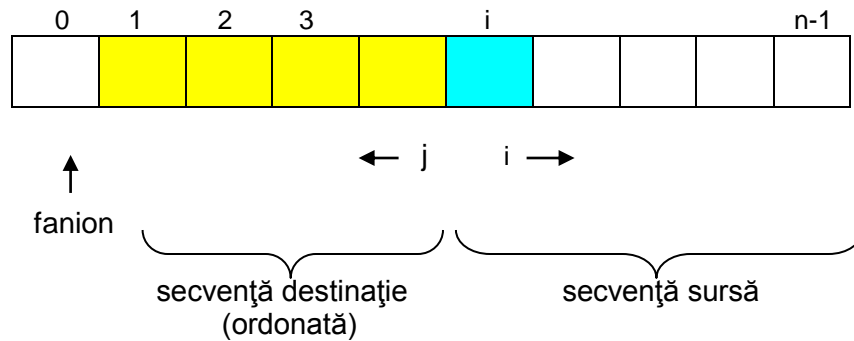


Fig.3.2.1.a. Modelul sortării prin inserție

- Acest caz tipic de repetiție cu **două condiții de terminare** readuce în atenție **metoda fanionului** (&1.4.2.1).
 - Pentru aplicarea ei se introduce **elementul auxiliar** $a[0]$ care se asignează inițial cu $a[i]$.
 - În felul acesta, cel mai târziu pentru $j=0$, condiția de a avea cheia lui $a[j]$ "mai mică sau egală" cu cheia lui $a[i]$ se găsește îndeplinită și nu mai trebuie verificată valoarea indicelui j (≥ 0).
 - Inserția propriu-zisă se realizează pe poziția $j+1$.
- Algoritmul care implementează sortarea prin inserție apare în [3.2.1.b] iar profilul său temporal în figura 3.2.1.b.

/*Sortare prin inserție*/

```

void SortarePrinInsertie                                     /*3.2.1.b*/
{
    tip_element a[n]; /*tablou de sortat*/
    int i,j;          /*definire indici de lucru*/
    tip_element temp; /*definire variabilă interschimbare*/
    /*parcurește secvență sursă stânga-dreapta*/
    for (i=2; i<n; i++)
    {
        temp=a[i]; /*inițializare interschimbare*/
        a[0]=a[i]; /*poziționare fanion*/
        j=i-1; /*inițializare indicator secvență destinație*/
        /*parcurește secvență destinație dreapta-stânga*/
        while (a[j].key>temp.key)
    }
}
  
```

```

{
    a[j+1]=a[j]; /*deplasare element curent spre
                  dreapta*/
    j=j-1;      /*avans indicator secvență destinație*/
} /*while*/
a[j+1]=temp;   /*inserție element curent*/
} /*for*/
} {SortarePrinInsertie}

```

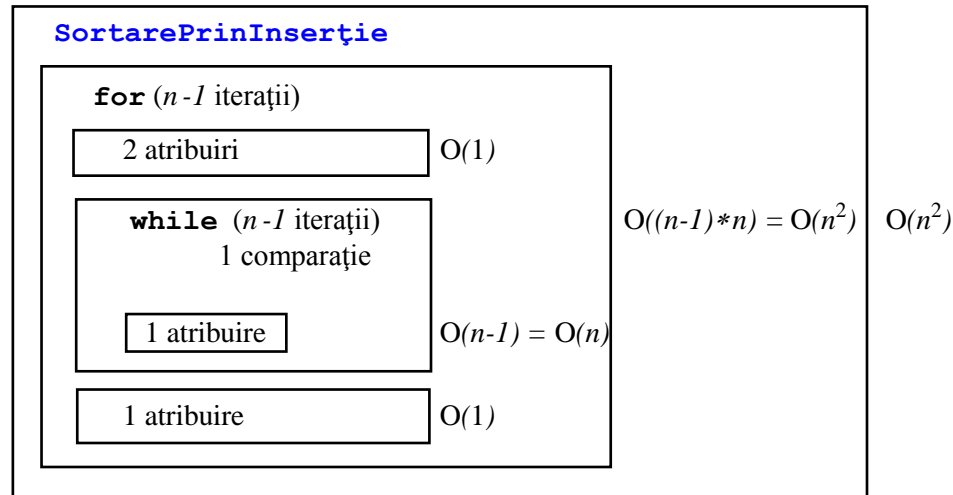


Fig.3.2.1.b. Profilul temporal al algoritmului de sortare prin inserție

- După cum se observă, algoritmul de sortare conține un **ciclu exterior** după i care se reia de $n-1$ ori (bucula **for**).
- În cadrul fiecărui ciclu exterior se execută un **ciclu interior** de lungime variabilă după j , până la îndeplinirea condiției (bucula **while**).
- În pasul i al ciclului exterior **for**, numărul minim de reluări ale ciclului interior este 0 iar numărul maxim de reluări este $i-1$.

3.2.1.1. Analiza sortării prin inserție

- În cadrul celui de-al i -lea ciclu **for**, numărul C_i al **comparațiilor de chei** executate în bucla **while**, depinde de ordinea inițială a cheilor, fiind:
 - Cel puțin 1 (secvența ordonată).
 - Cel mult $i-1$ (secvența ordonată invers).
 - În medie $i/2$, presupunând că toate permutările celor n chei sunt egal posibile.
 - Întrucât avem $n-1$ reluări ale lui **for** pentru $i = 2 \dots n$, parametrul C are valorile precizate în [3.2.1.c].
-

$$C_{\min} = \sum_{i=2}^n 1 = n - 1$$

$$C_{\max} = \sum_{i=2}^n (i-1) = \sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2} \quad [3.2.1.c]$$

$$C_{\text{med}} = \frac{C_{\min} + C_{\max}}{2} = \frac{n^2 + n - 2}{4}$$

- Numărul de **atribuiri de elemente** M_i în cadrul unui ciclu **for** este $C_i + 3$.
 - **Explicația:** la numărul C_i de atribuiri executate în cadrul ciclului interior **while** de tip $a[j+1] = a[j]$ se mai adaugă 3 atribuiri ($\text{temp} = a[i]$, $a[0] = \text{temp}$ și $a[i+1] = \text{temp}$).
 - Chiar pentru numărul minim de **comparații** de chei (C_i egal cu 1) cele trei atribuiri rămân valabile.
- În consecință, parametrul M ia următoarele valori [3.2.1.d].

$$M_{\min} = 3 \cdot (n - 1)$$

$$\begin{aligned} M_{\max} &= \sum_{i=2}^n (C_i + 3) = \sum_{i=2}^n (i + 2) = \sum_{i=1}^{n+2} i - (1 + 2 + 3) = \\ &= \frac{(n+2) \cdot (n+3)}{2} - 6 = \frac{n^2 + 5 \cdot n - 6}{2} \end{aligned} \quad [3.2.1.d]$$

$$M_{\text{med}} = \frac{M_{\min} + M_{\max}}{2} = \frac{n^2 + 11 \cdot n - 12}{4}$$

- Se observă că atât C cât și M sunt de ordinul lui n^2 ($O(n^2)$).
- Valorile **minime** ale indicatorilor rezultă când a este **ordonat**, iar valorile **maxime**, când a este **ordonat invers**.
- Sortarea prin inserție este o **sortare stabilă**.
- În secvența [3.2.1.e] se prezintă o variantă C ușor modificată a acestei metode de sortare.

/*Sortarea prin inserție - varianta C*/

```
insertie(int a[],int n)                                /*3.2.1.e*/
{
    /*parcurește secvența sursă dreapta-stânga*/
    for(int i=n-2;i>=0;i--)
```

```

{
    a[n]=a[i];          /*setare fanion pe poziția a[n]*/
    int j=i+1; /*inițializare indicator secvență destinație*/
    /*parcursere secvență destinație stânga-dreapta*/
    while(a[j]<a[n])
    {
        a[j-1]=a[j]; /*deplasare element curent spre
                        dreapta*/
        j++;          /*avans indicator secvență destinație*/
    }
    a[j-1]=a[n];      /*inserție element pe poziția a[j-1]*/
}
} /*insertie*/

```

- Relativ la secvența [3.2.1.e] se fac următoarele observații:
 - Implementarea este varianta "în oglindă" față de varianta pseudocod.
 - Tabloul de n elemente este $a[0] \dots a[n-1]$.
 - **Secvența sursă** este $a[0] \dots a[i]$.
 - **Secvența destinație** (cea ordonată) este $a[i+1] \dots a[n-1]$.
 - **Fanionul** este poziționat pe poziția n a tabloului a în locația $(a[n])$.
 - În procesul de căutare a locului de inserție, în pasul curent, se parcurge cu indicele j secvența destinație, de la stânga la dreapta respectiv de la poziția $i+1$ până la găsirea locului inserției sau până la n .
 - Elementele întâlnite care sunt mai mici ca și cheia de inserat se mută cu o poziție spre stânga până la îndeplinirea condiției (Fig. 3.2.1.c).

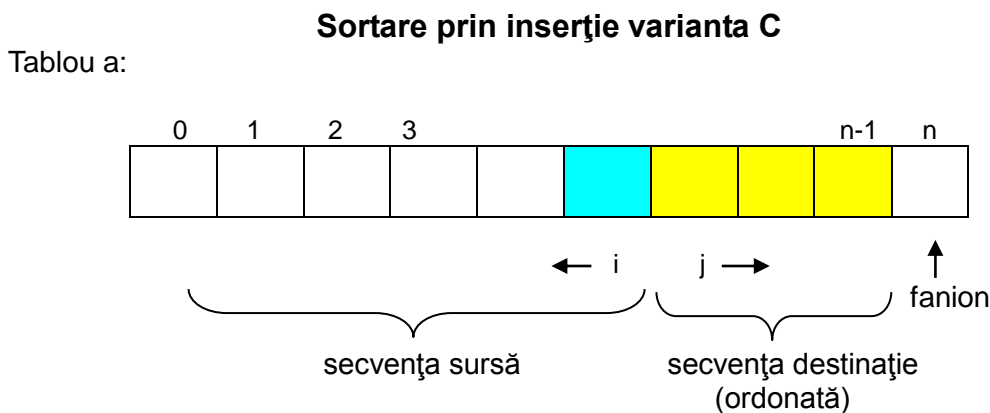


Fig.3.2.1.c. Model al algoritmului de sortare prin inserție varianta C

3.2.1.2. Sortarea prin inserție binară

- Algoritmul de **sortare prin inserție** poate fi îmbunătățit pornind de la observația că **secvența destinație** este deja **sortată**.

- În acest caz **căutarea locului de inserție** se poate face mult mai rapid utilizând tehnica **căutării binare**, prin împărțiri succesive în părți egale a intervalului de căutare.
- Algoritmul modificat se numește **inserție binară** [3.2.1.f].

/*Sortare prin inserție binară - Varianta pseudocod*/

```

PROCEDURE SortarePrinInserțieBinară          /*[3.2.1.f]*/
{
    tip_element a[n]; /*tablou de ordonat*/
    tip_indice i,j;    /*indici ordonare tablou*/
    tip_indice stanga,dreapta,m; /*indici căutare binară*/
    tip_element temp; /*variabilă interschimbare*/
    /*parcurgere secvență sursă stânga-dreapta*/
    pentru i=2 la n
        temp=a[i]; /*inițializare element de interschimbare*/
        stanga=1; dreapta=i-1; /*setare indici căutare binară*/
        /*căutare binară*/
        cât timp (stanga<=dreapta)
            m=(stanga+dreapta)DIV 2;
            if (a[m].cheie>temp.cheie)
                dreapta=m-1;
            else
                stanga=m+1;
        □ /*cât timp*/
        /*deplasare spre dreapta a elementelor secvenței
        destinație de la dreapta-stânga până la locul de
        inserție*/
        pentru j=i-1 la stanga a[j+1]=a[j];
        a[stanga]=temp; /*inserție element curent*/
        □ /*pentru*/
    } /*SortarePrinInserțieBinară*/

```

/*Sortare prin inserție binară - Varianta C */

```

void sortare_prin_inserție_binara()          /*[3.2.1.f]*/
{
    tip_element a[n]; /*tablou de ordonat*/
    int i,j; /*indici ordonare tablou*/
    int stanga,dreapta,m; /*indici căutare binară*/
    tip_element temp; /*variabilă interschimbare*/

    /*parcurgere secvență sursă stânga-dreapta*/
    for (i=2; i<=n; i++)
    {
        temp=a[i]; /*inițializare element de interschimbare*/
        stanga=1; dreapta=i-1; /*setare indici căutare binară*/
        /*căutare binară*/
        while (stanga<=dreapta)
        {
            m=(stanga+dreapta)/2; /*acces mijloc interval*/
            if (a[m].cheie>temp.cheie)
                dreapta=m-1; /*selecție interval stânga*/
            else
                stanga=m+1; /*selecție interval dreapta*/
        } /*while*/
        /*deplasare spre dreapta a elementelor secvenței

```

```

destinație de la dreapta-stânga până la locul de
insertie*/
for(j=i-1;j>=stanga;j--) a[j+1]=a[j];
a[stanga]=temp; /*insertie element curent*/
} /*for*/
} /*sortare_prin_insertie_binara*/

```

3.2.1.3. Analiza sortării prin inserție binară

- În cazul **sortării prin inserție binară**, poziția de inserat este găsită dacă

$$a[j].cheie \leq x.cheie \leq a[j+1].cheie ,$$

adică intervalul de căutare ajunge de dimensiune 1.

- Dacă intervalul inițial este de **lungime** i sunt necesari $\lceil \log_2(i) \rceil$ pași pentru determinarea locului inserției.
- Întrucât procesul de sortare presupune parcurgerea prin metoda înjumătățirii binare a unor secvențe de lungime i (care conțin i chei), pentru $i=1, 2, \dots, n$, **numărul total de comparații** C efectuate în bucla **WHILE** este cel evidențiat de expresia prezentată în [3.2.1.g].

$$C = \sum_{i=1}^n \lceil \log_2 i \rceil \quad [3.2.1.g]$$

Această sumă se poate aproxima prin **integrala**:

$$C = \int_1^n \log_2 x \cdot dx = x \cdot (\log_2 x - c) \Big|_1^n = n \cdot (\log_2 n - c) + c \quad [3.2.1.h]$$

$$c = \log_2 e = 1 / \ln 2 = 1.44269$$

- Numărul **comparațiilor** de chei este independent de ordinea inițială a cheilor.
 - Acesta este un caz de comportament **anormal** al unui algoritm de sortare.
- Din **nefericire** beneficiile **căutării binare** se răsfrâng **numai** asupra **numărului de comparații** și **nu** asupra **numărului de mișcări**.
- De regulă, **mutarea cheilor** și a informațiilor asociate necesită **mai mult timp** decât **compararea a două chei**.
 - Astfel întrucât M rămâne de ordinul lui n^2 , performanțele acestei metode de sortare **nu** cresc în măsura în care ar fi de așteptat.

- De fapt, re-sortarea unui tablou gata sortat, utilizând inserția binară, consumă mai mult timp decât inserția cu căutare secvențială.
- În ultimă instanță, **sortarea prin inserție nu** este o metodă potrivită de sortare cu ajutorul calculatorului, deoarece inserția unui element presupune deplasarea **poziției cu poziție** în tablou a unui număr de elemente, deplasare care este neeconomică.
 - Acest **dezavantaj** conduce la ideea dezvoltării unor algoritmi în care mișcările să afecteze un număr mai redus de elemente și să se realizeze pe **distanțe mai mari**.
- Sortarea prin **selecție** reprezintă un pas înainte în acest sens.

3.2.2. Sortarea prin selecție

- **Sortarea prin selecție** folosește procedeul de a **selecta** elementul cu cheia minimă și de a schimba între ele poziția acestui element cu cea a primului element.
 - Se repetă acest procedeu cu cele $n-1$ elemente rămase, apoi cu cele $n-2$, etc. terminând cu ultimele două elemente.
- Această metodă este oarecum opusă **sortării prin inserție** care presupune la fiecare pas **un singur element al secvenței sursă** și **toate elementele secvenței destinație** în care se caută de fapt locul de inserție.
- **Selecția în schimb** presupune **toate elementele secvenței sursă** dintre care selectează pe cel cu **cheia cea mai mică** și îl depozitează ca **element următor al secvenței destinație**.

```

/*Sortarea prin selecție - pas rafinare 1 - varianta
pseudocod*/
tip_element a[n];                                /*[3.2.2.a]*/
pentru i=0 la n-1
    *găsește cel mai mic element al lui a[i]...a[n] și
      asignează pe min cu indicele lui;
    *interschimbă pe a[i] cu a[min];
  □

```

- Modelul principal al sortării prin selecție apare în figura 3.2.2.a.

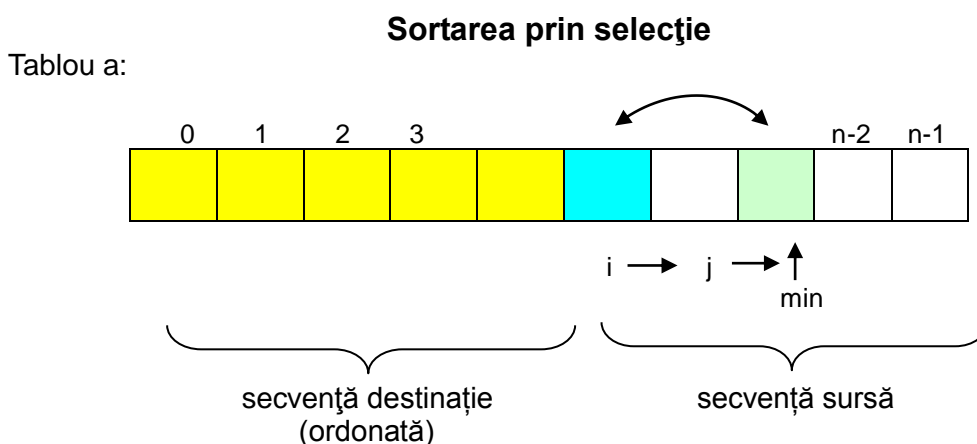


Fig.3.2.2.a. Modelul sortării prin selecție

- În urma procesului de detaliere rezultă algoritmul prezentat în [3.2.2.b] al cărui profil temporal apare în figura 3.2.2.b.

```
-----  
/* Sortare prin selecție - varianta C */  
  
void sortare_prin_selectie()                                /*3.2.2.b*/  
{  
    int i,j;                                                /*indici de ordonare*/  
    int min;                                                 /*cel mai mic (cmm) element curent*/  
    tip_element temp; /*variabilă interschimbare*/  
    tip_tablou a[n]; /*tablou de sortat*/  
  
    /*parcursere secvență sursă stânga-dreapta*/  
    for (i=0; i<=n-2; i++)  
    {  
        /*selectează cel mai mic element curent*/  
        min=i; temp=a[i]; /*inițializare cmm element curent*/  
        for (j=i+1; j<=n-1; j++)  
            if (a[j].cheie<temp.cheie)  
            {  
                min=j; temp=a[j]; /*marcare cmm element curent*/  
            } /*for*/  
        a[min]=a[i]; a[i]=temp; /*interschimbare a[i]<->a[min]*/  
    } /*for*/  
} /*sortare_prin_selectie*/  
-----
```

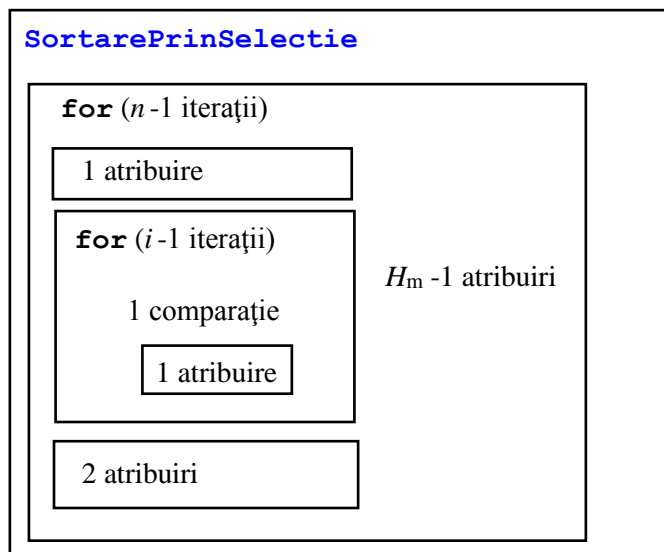


Fig.3.2.2.b. Profilul temporal al sortării prin selecție

3.2.2.1. Analiza sortării prin selecție

- Numărul **comparațiilor** de chei C este independent de ordinea inițială a cheilor. El este **fix** fiind determinat de derularea celor două bucle **for** încuibate.

$$C = \sum_{i=1}^{n-1} (i-1) = \sum_{i=1}^{n-2} i = \frac{n^2 - 3 \cdot n + 2}{2} \quad [3.2.2.c]$$

- Numărul minim al **atribuirilor** este de cel puțin 3 pentru fiecare valoare a lui i, (temp=a[i], a[min]=a[i], a[i]=temp), de unde rezultă:

$$M_{\min} = 3 \cdot (n-1) \quad [3.2.2.d]$$

- Acest minim poate fi atins efectiv, dacă inițial cheile sunt deja **sortate**.
- Pe de altă parte, dacă cheile sunt **sortate inițial în ordine inversă**, M_{\max} se determină cu ajutorul **formulei empirice** [3.2.2.e] [Wi76].

$$M_{\max} = \left\lfloor \frac{n^2}{4} \right\rfloor^{(1)} + 3 \cdot (n-1) \quad [3.2.2.e]$$

- **Valoarea medie** a lui M **nu** este media aritmetică a lui M_{\min} și M_{\max} , ea obținându-se printr-un **raționament probabilistic** în felul următor:

- Se consideră o secvență de m chei.
- Fie o **permutare oarecare** a celor m chei notată cu k_1, k_2, \dots, k_m .
- Se determină numărul termenilor k_j având proprietatea de a fi **mai mici** decât toți termenii precedenți k_1, k_2, \dots, k_{j-1} .
- Se adună toate valorile obținute pentru cele $m!$ permutări posibile și se împarte suma la $m!$
- Se demonstrează că rezultatul acestui calcul este $H_m - 1$, unde H_m este **suma** parțială a **seriei armonice** [3.2.2.f] [Wi76]:

$$H_m = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{m} \quad [3.2.2.f]$$

- Această valoare reprezintă **media numărului de atribuiri** ale variabilei `temp`, executate în procesul de sortare pentru o secvență de m elemente în bucla **for** interioară, deoarece `temp` se atribuie ori de câte ori se găsește un element mai mic decât toate elementele care-l preced.
- Ținând cont și de atribuiri `temp=a[i]`, `a[min]=a[i]` și `a[i]=temp`, **valoarea medie a numărului total de atribuiri** la o trecere pentru m termeni este H_m+2 .
- Se demonstrează că deși **seria este divergentă**, o **sumă parțială** a sa se poate calcula cu ajutorul formulei [3.2.2.g]:

$$H_m \approx \ln m + \gamma + \frac{1}{2 \cdot m} - \frac{1}{12 \cdot m^2} + \frac{1}{120 \cdot m^4} \quad [3.2.2.g]$$

unde $\gamma = 0.5772156649\dots$ este constanta lui **Euler** [Kn76].

- Pentru un m suficient de mare valoarea lui H_m se poate aproxima prin expresia:
-

$$H_m \approx \ln m + \gamma \quad [3.2.2.h]$$

- Tot acest raționament este valabil la **o trecere** pentru o secvență de m chei.
 - Întrucât în procesul de sortare se parcurg succesiv secvențe care au respectiv lungimea $m=n, n-1, n-2, \dots, 1$, fiecare dintre ele necesitând în medie H_m+2 atribuiri, numărul mediu total de atribuiri M_{med} va avea expresia:
-

$$M_{\text{med}} \approx \sum_{m=1}^n (H_m + 2) \approx \sum_{m=1}^n (\ln m + \gamma + 2) = n \cdot (\gamma + 2) + \sum_{m=1}^n \ln m \quad [3.2.2.i]$$

- Suma de termeni discreți, poate fi aproximată cu ajutorul calculului integral [3.2.2.j].
-

$$\int_1^n \ln x \cdot dx = x \cdot (\ln x - 1) \Big|_1^n = n \cdot \ln(n) - n + 1 \quad [3.2.2.j]$$

- Această aproximare conduce la rezultatul final [3.2.2.k]:
-

$$M_{\text{med}} \approx n \cdot (\ln m + \gamma + 1) + 1 = O(n \cdot \ln n)$$

[3.2.2.k]

-
- În concluzie, **algoritmul bazat pe selecție** este de **preferat** celui bazat **pe inserție**, cu toate că în cazurile în care cheile sunt ordonate, sau aproape ordonate, sortarea prin inserție este mai rapidă.
 - Optimizarea performanței sortării prin selecție poate fi realizată prin reducerea numărului de mișcări de elemente ale tabloului.
 - Sedgewik [Se88] propune o astfel de variantă în care în loc de a se memora de fiecare dată elementul minim curent în variabila `temp`, se reține doar indicele acestuia, mutarea urmând a se realiza doar pentru ultimul element găsit, la părăsirea ciclului **for** interior [3.2.2.l].
-

/*Sortare prin selecție optimizată - Varianta C*/

```
void selectie_optimizata()                                /*3.2.2.1*/
{
    int i,j;      /*indici de ordonare*/
    int min;      /*indice cel mai mic (cmm) element curent*/
    tip_element temp; /*variabilă interschimbare*/
    tip_tablou a[n]; /*tablou de sortat*/

    /*parcursere secvență sursă stânga-dreapta*/
    for (i=0; i<=n-2; i++)
    {
        min=i; /*inițializare indice cmm element curent*/
        /*selecție cmm element curent*/
        for (j=i+1; j<=n; j++)
            if (a[j].cheie<temp.cheie) min=j; /*marcare cmm
                                                element curent*/

        /*interschimbare a[i]<->a[min]*/
        temp=a[min]; a[min]=a[i]; a[i]=temp;
    } /*for*/
} /*selecție_optimizată*/
```

- **Măsurătorile experimentale** efectuate însă asupra acestei variante **nu** evidențiază vreo ameliorare nici chiar pentru valori mari ale dimensiunii tabloului de sortat.
 - Explicația rezidă în faptul ca atribuirile care necesită în plus accesul la componenta unui tablou se realizează practic în același timp ca și o atribuire normală.

3.2.3. Sortarea prin interschimbare. Sortările bubblesort și shakersort

- Clasificarea metodelor de sortare în diferite familii ca **inserție**, **interschimbare** sau **selecție** nu este întotdeauna foarte bine definită.
 - Paragrafele anterioare au analizat algoritmi care deși implementează inserția sau selecția, se bazează pe fapt pe interschimbare.

- În acest paragraf se prezintă o **metodă de sortare** în care **interschimbarea** a două elemente este caracteristica dominantă.
- **Principiul de bază al sortării prin interschimbare** este următorul:
 - Se compară și se interschimbă perechile de elemente alăturate, până când toate elementele sunt sortate.
- Ca și la celelalte metode, se realizează **tregeri repetate** prin tablou, de la capăt spre început, de fiecare dată deplasând cel mai mic element al mulțimii rămase spre capătul din stânga al tabloului.
- Dacă se consideră tabloul în poziție verticală și se asimilează elementele sale cu niște **bule de aer** în interiorul unui **lichid**, fiecare bulă având o **greutate** proporțională cu valoarea cheii, atunci fiecare trecere prin tablou se soldează cu **ascensiunea** unei bule la nivelul specific de greutate.
- Din acest motiv această metodă de sortare este cunoscută în literatură sub denumirea de **bubblesort** adică **sortare prin metoda bulelor**.
- Modelul principal al sortării bubblesort apare în fig. 3.2.3.a iar algoritmul aferent acestei metode apare în secvența [3.2.3.a].

Sortarea bubblesort

Tabloul a:

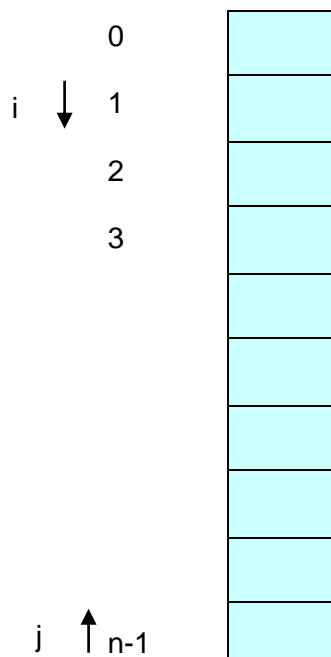


Fig.3.2.3.a. Modelul principal al sortării bubblesort

```
/*Sortarea prin interschimbare Bubblesort - varianta 1 -
pseudocod*/
```

```
PROCEDURE Bubblesort;
```

```
/*[3.2.3.a]*/
```



```

{
    tip_element a[n]; /*tablou de sortat*/
    tip_indice i,j;; /*indici de sortare*/

    /*parcuregere de sus în jos*/
    pentru i=0 la n-1
        /*parcuregere de jos în jos*/
        pentru j=n-1 spre i
            if(a[j-1].cheie>a[j].cheie)
                *interschimbă a[j] cu a[j-1]
        □ /*pentru*/
    □ /*pentru*/
    /*Bubblesort*/
}
-----
/*Sortarea prin interschimbare Bubblesort - varianta 1 - C*/

void bubblesort() /*3.2.3.a*/
{
    tip_element a[n]; /*tablou de sortat*/
    int temp; /*variabilă interschimbare*/
    int i,j; /*indici de sortare*/

    /*parcuregere de sus în jos*/
    for(i=0;i<n;i++)
    {
        /*parcuregere de jos în jos*/
        for(j=n-1;j>=i;j--)
            if (a[j-1].cheie>a[j].cheie)
            {
                temp=a[j-1]; a[j-1]=a[j]; /*interschimbare*/
                a[j]=temp;
            }
    }
} /*bubblesort*/
}
-----

```

- Profilul temporal al algoritmului de sortare prin interschimbare este prezentat în figura 3.2.3.b și el conduce la o estimare a încadrării performanței algoritmului în ordinul $O(n^2)$.

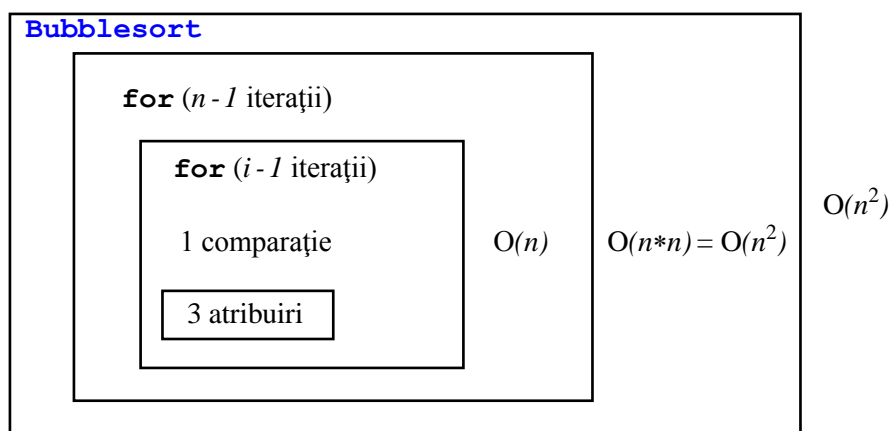


Fig.3.2.3.b. Profilul temporal al sortării prin interschimbare

- Se pot observa trei elemente importante:
- (1) În multe cazuri ordonarea se termină **înainte** de a se parcurge toate iterațiile buclei **for** exterioare.
 - În acest caz, restul iterațiilor sunt fără efect, deoarece elementele sunt deja ordonate.
 - În consecință, o modalitate evidentă de **îmbunătățire** a algoritmului bazată pe această observație este aceea prin care se memorează **dacă a avut sau nu loc** vreo interschimbare în cursul unei treceri.
 - Și în acest caz este însă necesară o **ultimă trecere**, fără nici o interschimbare care marchează finalizarea algoritmului.
 - O variantă a sortării bubblesort bazată pe observația (1) apare în [3.2.3.b]. Această variantă este binecunoscută programatorilor datorită în special simplității sale.

{Sortarea prin interschimbare - Varianta 2 - pseudocod}

```
PROCEDURE Bubblesort1;                                /*[3.2.3.b]*/

tip_element a[n]; /*tablou de sortat*/
tip_indice i;     /*indice sortare*/
boolean modificat; /*variabilă semnalare schimbare*/

repetă
    modificat=false; /*inițializare semnalare schimbare*/
    pentru i=0 la n-2
        dacă (a[i].cheie>a[i+1].cheie)
            *interschimba a[i] cu a[i+1];
            modificat=true /*semnalare modificare*/
            □ /*dacă*/
        □ /*pentru*/
    până când NOT modificat
    □ /*repetă*/
/*Bubblesort1*/
```

/*Sortarea prin interschimbare(varianta 2)- C*/

```
typedef int boolean;
#define true (1)
#define false (0)

void bubblesort1()                                /*3.2.3.b*/
{
    tip_element a[n]; /*tablou de sortat*/
    tip_element temp; /*variabilă interschimbare*/
    int i;            /*indice sortare*/
    boolean modificat; /*variabilă semnalare modificare*/

    do {
        modificat=false; /*inițializare semnalare modificare*/
        for(i=0;i<=n-2;i++)
            if (a[i].cheie>a[i+1].cheie)
```

```

    {
        /*interschimbare*/
        temp=a[i]; a[i]=a[i+1]; a[i+1]=temp;
        modificat=true; /*semnalare modificare*/
    }
} while(modificat);
} /*bubblesort1*/
/*-----*/

```

- (2) Îmbunătățirea analizată, poate fi la rândul ei perfecționată, memorând nu faptul că a avut loc sau nu o interschimbare ci **indicele k al ultimei schimbări**.

- Este evident faptul că toate perechile de elemente situate sub acest indice k (care au indici mai mici) sunt deja ordonate, în consecință trecerile următoare pot fi terminate la acest indice în loc să fie terminate la indicele predeterminat ca limită i.

- (3) La o analiză atentă se poate observa o **asimetrie** particulară:

- Un element **ușor** plasat la capătul **greu** al tabloului este readus la locul său într-o singură trecere.

- Spre exemplu tabloul:

12 18 22 34 65 67 83 04

va fi sortat cu ajutorul metodei bubblesort îmbunătățite într-o singură trecere.

- În schimb un element **greu** plasat la capătul **ușor** al tabloului va fi readus spre locul său doar cu câte o poziție la fiecare trecere.

- Spre exemplu tabloul:

83 04 12 18 22 34 65 67

va necesita șapte treceri în vederea sortării.

- Această neobișnuită asimetrie, sugerează o a treia îmbunătățire: **alternarea sensurilor de parcurgere ale trecerilor consecutive**.
- Algoritmul care include aceste îmbunătățiri se numește **shakersort** (sortare prin amestecare). Modelul principal al acestei sortări apare în figura 3.2.3.c iar implementarea este prezentată în secvența [3.2.3.c].

Sortarea shakersort

Tabloul a:

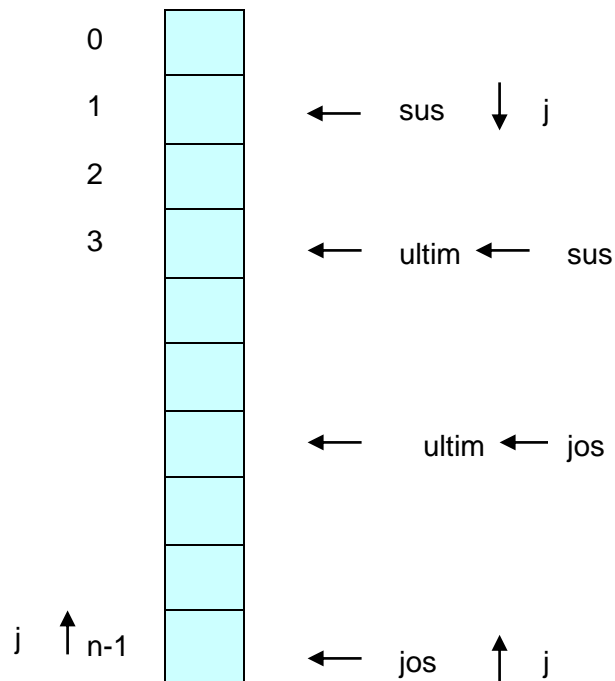


Fig.3.2.3.c. Modelul principal al sortării shakersort

/*Sortarea prin interschimbare(varianta 3) implementare C*/

```
void shakersort() /*3.2.3.c*/
{
    tip_element a[n]; /*tablou de sortat*/
    tip_element temp; /*variabilă interschimbare*/
    int j; /*indice de parcurgere*/
    int sus, jos, ultim; /*indici de limitare*/

    /*inițializare indici de limitare*/
    sus=1; jos=n-1; ultim=n-1;
    /*ordonare tablou*/
    do {
        /*parcurgere de jos în sus*/
        for(j=jos; j>=sus; j--)
            if (a[j-1].cheie>a[j].cheie)
            {
                temp=a[j-1]; a[j-1]=a[j]; a[j]=temp;
                ultim=j; /*setare indice ultima schimbare*/
            } /*for*/
        sus=ultim+1; /*actualizare limita sus*/
        /*parcugere de sus în jos*/
        for(j=sus; j<=jos; j++)
            if (a[j-1].cheie>a[j].cheie)
            {
                temp=a[j-1]; a[j-1]=a[j]; a[j]=temp;
            }
    } while (sus<jos);
}
```

```

        ultim=j; /*setare indice ultima schimbare*/
    } /*for*/
    jos=ultim-1; /*actualizare limita jos*/
} while (!(sus>jos));
} /*shakersort*/
/*-----*/

```

3.2.3.1. Analiza sortărilor bubblesort și shakersort

- Numărul comparațiilor la **algoritmul bubblesort** este constant și are valoarea:

$$C = \sum_{i=1}^{n-1} (i-1) = \frac{n^2 - 3 \cdot n + 2}{2} \quad [3.2.3.d]$$

- Valorile minimă, maximă și medie ale **numărului de mișcări** sunt:

$$M_{\min} = 0$$

$$M_{\max} = 3 \cdot C = \frac{3}{2} \cdot (n^2 + 3 \cdot n + 2) \quad [3.2.3.e]$$

$$M_{\text{med}} = \frac{3}{4} (n^2 + 3 \cdot n + 2)$$

- Analiza metodei îmbunătățite **shakersort** arată că $C_{\min} = n-1$.
 - Pentru ceilalți indicatori, Knuth ajunge la un **număr mediu de treceri** proporțional cu $n - k_1 \sqrt{n}$ și la un **număr mediu de comparații** de chei proporțional cu $C_{\text{med}} = 1/2 (n^2 - n(k_2 + 1 \ln n))$.
- Trebuie însă remarcat faptul că toate **îmbunătățirile propuse nu afectează** în nici un fel **numărul de interschimbări**. Ele reduc numai numărul de verificări redundante.
- Din păcate însă **interschimbarea a două chei** este mult mai costisitoare ca timp decât compararea lor, prin urmare toate aceste îmbunătățiri atent studiate au un efect **mult mai redus** decât s-ar aștepta în mod intuitiv.
- Analiza comparativă a performanțelor algoritmilor de sortare prezentați, scoate în evidență următoarele:
 - **Sortarea prin interschimbare** este **mai puțin performantă** decât sortările prin **inserție** sau **selecție**, astfel încât utilizarea ei nu este recomandabilă.
 - Algoritmul **shakersort** poate fi utilizat în mod avantajos în cazurile în care elementele sunt aproape sortate, caz însă destul de rar întâlnit în practică.

- Se poate demonstra că **distanța medie** pe care fiecare element al unui tablou de dimensiune n , o parcurge în procesul de sortare este de $n/3$ locuri.
- Deoarece în metodele prezentate până acum (cu excepția sortării prin selecție), fiecare element își modifică doar cu un singur loc poziția la fiecare pas elementar, este necesar un număr de treceri proporțional cu n^2 .
- O **îmbunătățire** efectivă a performanței trebuie să aibă în vedere deplasarea elementelor pe distanțe mai mari într-un singur pas.

3.2.4. Sortarea prin inserție cu diminuarea incrementului. Sortarea shellsort

- **D.L. Shell** a propus în 1959, o perfecționare a **metodei de sortare prin inserție** directă.
- **Ideea** acestei metode numite **sortare prin inserție cu diminuarea incrementului** este următoarea:
 - La început, toate articolele care sunt despărțite prin câte **4 poziții**, sunt grupate și sortate separat prin metoda inserției.
 - Acest proces se numește **sortare-4**.
 - În exemplul din fig.3.2.4, unde se sortează 8 elemente, s-au format 4 grupe de elemente separate prin câte 4 poziții.
 - După această primă trecere, sunt grupate elementele sunt separate prin câte **două poziții** și din nou sunt sortate prin inserție.
 - Acest proces se numește **sortare-2**.
 - În final, la cea de-a treia trecere, elementele sunt sortate obișnuit (**sortare-1**).
 - Se precizează faptul că fiecare **k-sortare** este de fapt o **sortare prin inserție** la care **pasul este k** (nu 1 ca la inserția normală).

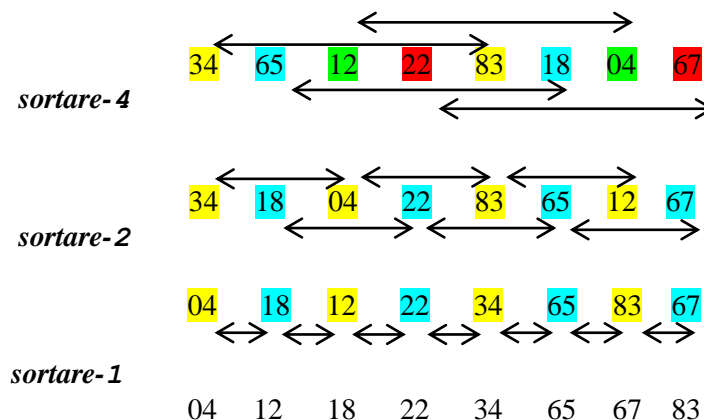


Fig. 3.2.4. Sortare prin inserție cu diminuarea incrementului

- Deși la prima vedere această metodă care presupune câteva treceri asupra tuturor elementelor, nu pare foarte performantă, totuși la o analiză mai atentă, fiecare trecere realizează relativ puține modificări ale pozițiilor elementelor.
- Este evident că metoda conduce la sortarea elementelor și că fiecare pas profită de cei anteriori deoarece fiecare **sortare- i** combină grupuri deja **sortate- j** .
- Este posibilă **orice secvență de incremenți descrescători** atâta timp cât **ultimul increment este egal cu unitatea**.
 - În cel mai rău caz, procesul de sortare se va realiza în întregime în acest pas.
- Este mai puțin evident, dar practica demonstrează că o secvență de incremenți descrescători care **nu** sunt puteri ale lui 2 asigură o eficiență superioară procesului de sortare.

3.2.4.1. Analiza metodei de sortare shellsort

- **Analiza metodei shellsort** pune probleme deosebite din punct de vedere matematic, multe din ele încă nerezolvate.
 - În particular, **nu** se cunoaște nici măcar **secvența de incremenți** cea mai potrivită.
 - Ceea ce este deosebit de interesant este faptul că **incremenții nu** trebuie să fie unii **multipli** altora.
- Pentru o eficiență sporită a sortării este de dorit ca între diferitele lanțuri de parcurgere să aibă loc cât mai multe interacțiuni.
- De asemenea este valabilă următoarea **teoremă** pe care de fapt se bazează metoda:
 - Dacă o secvență **sortată- k** este **sortată- i** ea rămâne și **sortată- k** .
 - Cu alte cuvinte, procesul de sortare cu diminuarea incrementului este **cumulativ**.
- **Knuth** indică drept cele mai potrivite secvențe de incremenți cele prezentate în [3.2.4.c] respectiv [3.2.4.d] (furnizate în ordine crescătoare):

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

$h_t, h_{t-1}, \dots, h_k, h_{k-1}, \dots, h_1$

[3.2.4.c]

unde $h_{k-1} = 3 \cdot h_k + 1$, $h_t = 1$ și $t = \lfloor \log_3 n \rfloor - 1$

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

[3.2.4.d]

unde $h_{k-1} = 2 \cdot h_k + 1$, $h_t = 1$ și $t = \lfloor \log_2 n \rfloor - 1$

-
- Pentru ultima secvență, analiza matematică a sortării a n elemente cu **metoda Shellsort** demonstrează necesitatea unui efort proporțional cu $n^{1.2}$ ($O(n^{1.2})$).
 - În [3.2.4.e] se prezintă o variantă de implementare acestei metode.
 - Algoritmul utilizează incremenți bazați pe formula [3.2.4.c], unde incrementul maxim se calculează funcție de dimensiunea tabloului în prima buclă **do-while** [Se88].
 - Incremenții descrescători se calculează automat la reluarea fiecărui ciclu de sortare, respectiv a doua buclă **do-while**.
 - Avem de fapt o suită de **sortări prin inserție** cu incrementul variabil descrescător h .

/*Sortarea Shellsort (Varianta Sedgewick) - implementare C*/

```
void shellsort()                                /*3.2.4.e*/
{
    tip_element a[n]; /*tablou de sortat*/
    tip_element temp; /*variabilă interschimbare*/
    int i,j;          /*indici de ordonare*/
    int h;             /*increment curent*/

    /*calcul valoare increment maxim*/
    h=1; /*inițializare increment*/
    do {
        h=3*h+1;
    } while (!(h>n));

    /*sortare shellsort*/
    do {
        h=h/3; /*calcul valoare increment curent*/
        /*sortare prin inserție cu incrementul h*/
        for(i=h+1;i<=n;i++)
        {
            temp=a[i]; j=i;
            while((a[j-h].cheie>temp.cheie)&&(j>h))
            {
                a[j]=a[j-h]; /*deplasare element curent*/
                j=j-h;
            } /*while*/
            a[j]=temp; /*inserție element curent*/
        } /*for*/
    } while (!(h==1));
} /*shellsort*/
```

3.2.5. Sortarea prin metoda ansamblelor. Sortarea heapsort

- Metoda **sortării prin selecție** se bazează pe selecția repetată a celei mai mici chei dintre n elemente, apoi dintre cele $n-1$ rămase, etc.

- Este evident că determinarea celei mai mici chei dintre n elemente necesită $n-1$ comparații, dintre $n-1$ elemente necesită $n-2$ comparații, etc.
- Activitatea de selecție poate fi **îmbunătățită**, dacă la fiecare trecere se vor reține **mai multe informații** și nu doar elementul cu cheia cea mai mică.
 - Astfel spre exemplu din $n/2$ comparații se poate determina **cea mai mică cheie a fiecărei perechi de elemente**.
 - Din alte $n/4$ comparații, **cea mai mică cheie a fiecărei perechi de chei mici** deja determinate și așa mai departe.
 - În final, utilizând doar $n/2 + n/4 + \dots + 4 + 2 + 1 = n-1$ comparații, se poate construi un **arbore de selecții** având drept rădăcină cheia cea mai mică (fig.3.2.5.a).
 - Arborele de selecții este de fapt un **arbore binar parțial ordonat**.

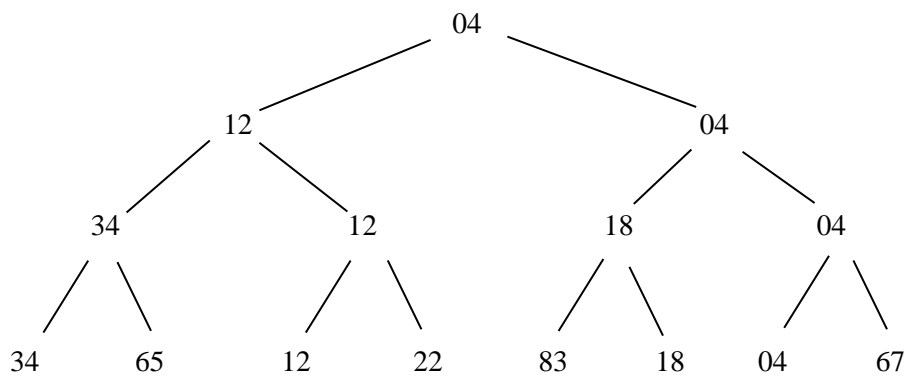


Fig.3.2.5.a. Arbore de selecții

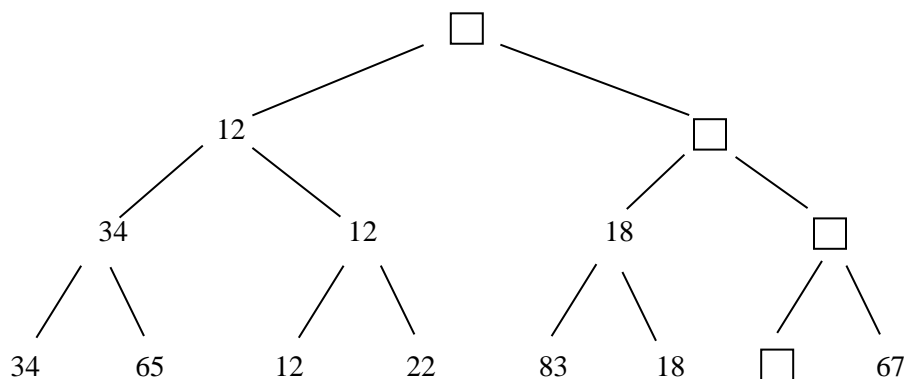


Fig. 3.2.5.b. Selecția traseului celei mai mici chei

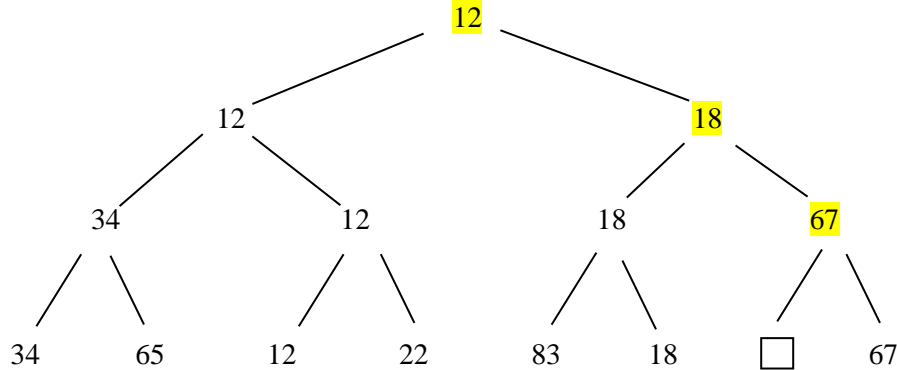


Fig. 3.2.5.c. Completarea locurilor libere

- Cum se poate utiliza acest arbore la sortare?
 - Se extrage cheia cea mai **mică** din rădăcina arborelui.
 - În continuare se parcurge în **sens invers** drumul urmat de cheia cea mai mică și se elimină succesiv această cheie (Fig. 3.2.5.b).
 - Pe acest parcurs cheia se înlocuiește (Fig. 3.2.5.c):
 - (1) Cu un **loc liber**, la baza structurii arbore.
 - (2) Cu **elementul ramurii alternative** în cazul unui nod intermediar.
- Din nou, elementul care va răzbate spre **rădăcina arborelui** va fi cel cu **cheia cea mai mică** din cele rămase, element care poate fi extras și procesul se repetă.
- După n astfel de pași de selecție, s-au extras succesiv cele n elemente ale mulțimii în ordine crescătoare, arborele devine vid și procesul de sortare este încheiat.
- Trebuie notat faptul că fiecare din cei n pași de selecție necesită **numai** $\log_2 n$ comparații, adică un număr de comparații egal cu înălțimea arborelui.
- În consecință, **procesul de sortare integrală** necesită:
 - Un număr de n pași pentru construcția arborelui.
 - Un număr de operații elementare de ordinul lui $n \cdot \log_2 n$ pentru sortarea propriu-zisă.
- Aceasta este o îmbunătățire considerabilă față de metodele directe care necesită un efort de ordinul $O(n^2)$ și chiar față de Shellsort care necesită $O(n^{1.2})$.
- Este evident faptul că în cazul metodei de sortare bazată pe structura arbore, **complexitatea** pașilor de sortare individuali crește.
- De asemenea, în vederea reținerii unei cantități sporite de informație, trebuie concepută o **structură de date** aparte care să permită organizarea eficientă a informației.

- Respectiva **structură de date** trebuie să respecte următoarea **specificație**:
 - (1) Să elimine **locurile goale**, care pe de o parte sporesc dimensiunea arborelui, iar pe de altă parte sunt sursa unor comparații care nu sunt necesare.
 - (2) Arborele să fie reprezentat utilizând locații de memorie pentru n elemente și nu pentru $2n - 1$ elemente așa cum rezultă din figurile 3.2.5.a, b, c.
- Aceste probleme au fost rezolvate de către **J. Williams**, creatorul **metodei de sortare heapsort** (*sortare de ansamblu*).
- Metoda în sine, reprezintă o realizare de **excepție** printre metodele convenționale de sortare și utilizează o reprezentare specială a unui **arbore binar parțial ordonat**, numită **"heap"** sau **"ansamblu"**.
- Un **ansamblu** ("heap") este definit ca o secvență de chei $h_{stanga}, h_{stanga+1}, \dots, h_{dreapta}$ care se bucură de proprietățile [3.2.5.a]:

$$\begin{aligned}
 h_i &\leq h_{2i} & \text{pentru toți } i = stanga, \dots, dreapta/2 & \quad [3.2.5.a] \\
 h_i &\leq h_{2i+1}
 \end{aligned}$$

- Un **ansamblu** poate fi asimilat cu un **arbore binar parțial ordonat** și reprezentat printr-un **tablou**.
- Spre exemplu, ansamblul h_1, h_2, \dots, h_{15} poate fi asimilat cu arborele binar din figura 3.2.5.d și poate fi reprezentat prin tabloul h în baza următoarei tehnici:
 - (1) Se **numeratează** elementele ansamblului, nivel cu nivel, de sus în jos și de la stânga la dreapta.
 - (2) Se **asociază** elementelor ansamblului, locațiile unui **tablou** de elemente h , astfel încât elementului h_i al ansamblului îi corespunde locația $h[i]$ din tablou.

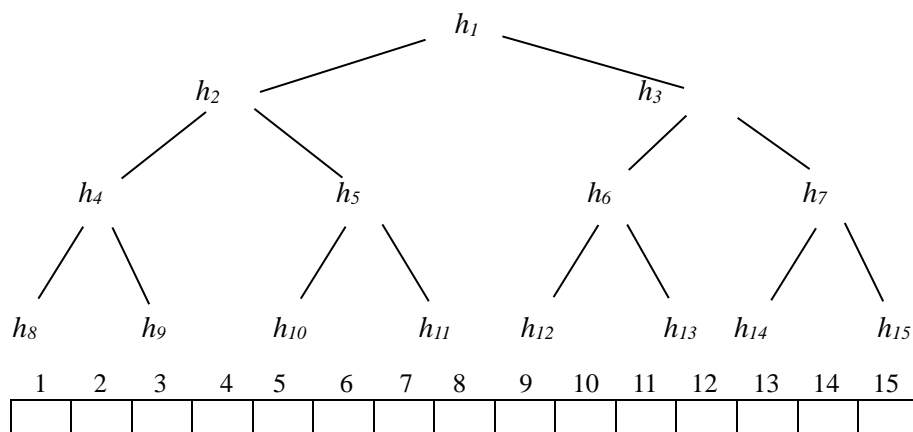


Fig. 3.2.5.d. Reprezentarea unui ansamblu printr-un tablou liniar h

- Un ansamblu se bucură de proprietatea că **primul** său element este cel mai mic dintre toate elementele ansamblului adică $h_1 = \min(h_1, \dots, h_n)$.
- Se presupune un **ansamblu parțial** $h_{s+1}, h_{s+2}, \dots, h_d$ definit prin indicii $s+1$ și d .
 - Acestui ansamblu i se adaugă la stânga, pe poziția h_s un nou element x , obținându-se un **ansamblu extins spre stânga** h_s, \dots, h_d .
- În figura 3.2.5.e.(a) apare ca exemplu ansamblul h_2, \dots, h_7 , iar în aceeași figură (b), ansamblul extins spre stânga cu un element $x=34$.
 - Noul ansamblu se obține din cel anterior plasând pe x în vârful ansamblului și deplasându-l "în jos" de-a lungul drumului indicat de componentele cele mai mici, care în același timp urcă.
 - Astfel valoarea 34 este mai întâi schimbată cu valoarea 04, apoi cu valoarea 12, generând structura din figura amintită.
 - Se poate verifica cu ușurință că această deplasare conservă condițiile care definesc un **ansamblu** [3.2.5.a].

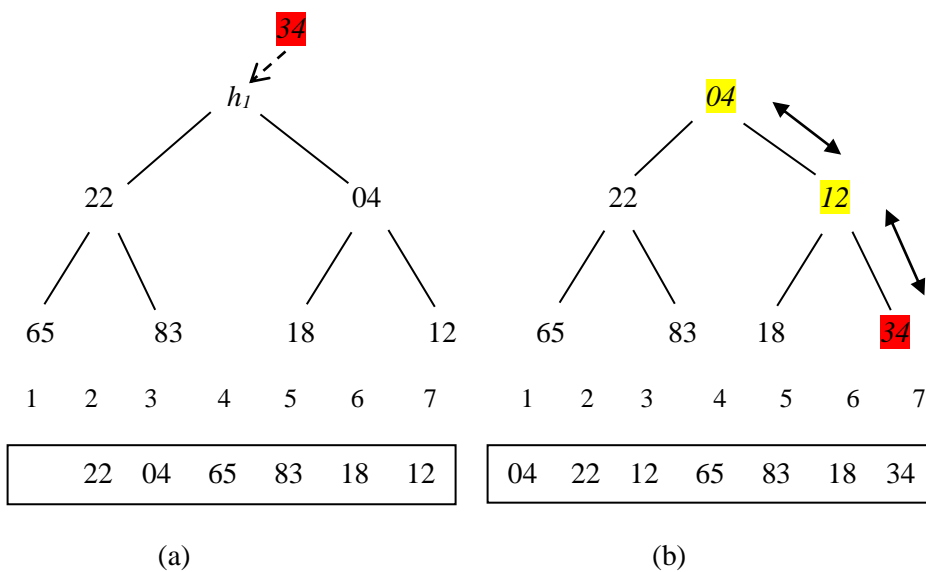


Fig. 3.2.5.e. Deplasarea unei chei într-un ansamblu

- Notând cu i și j indicii elementelor care se interschimbă, și presupunând că x a fost introdus pe poziția h_{stanga} , tehnica de implementare a unei astfel de deplasări apare în secvența [3.2.5.b] în variantă **pseudocod**.

*/*Deplasarea unei chei de sus în jos într-un ansamblu h. Cheia nouă este introdusă pe poziția h[stanga]- varianta pseudocod*/*

```

procedure Deplasare(TipIndice stanga,dreapta) [3.2.5.b]
  /*stânga și dreapta sunt limitele ansamblului*/
  i=stanga; /*indică elementul curent*/
  j= 2*i; /*indică fiul stâng al elementului curent*/

```

```

temp= h[i] /*elementul care se deplasează*/
cât timp există niveluri în ansamblu (j≤dreapta) și locul
    de plasare nu a fost găsit execută
    *selectează pe cel mai mic dintre fii elementului
        indicat de i (pe h[j] sau pe h[j+1]);
    dacă temp>fiul_selectat atunci
        *deplasează fiul selectat în locul tatălui
            său (h[i]=h[j]);
        *avansează pe nivelul următor al ansamblului
            (i=j; j=2*i);
    altfel
        *locul a fost găsit;
    □
□
*plasează pe temp la locul său în ansamblu
    (h[i]=temp);

```

- Funcția C care implementează algoritmul de deplasare apare în secvența [3.2.5.c].

/* Deplasarea unei chei de sus în jos într-un ansamblu - Varianta C */

```

                                                                    /*[3.2.5.c]*/
void deplasare(tip_indice stanga, tip_indice dreapta)
{ /*stânga și dreapta sunt limitele ansamblului*/
  /*elementul de deplasat se află în poziția h[stanga]*/

  tip_indice i,j;          /*indici de parcurgere*/
  tip_element temp;        /*variabilă tampon*/
  boolean ret;             /*indicator boolean terminare proces*/

  i=stanga; /*indică elementul curent (vârful ansamblului)*/
  j=2*i;     /*indică fiul stâng al elementului curent*/
  temp=h[i]; /*elementul care se deplasează*/
  ret=false; /*inițializare indicator terminare*/
  /*deplasare în ansamblu*/
  while((j<=dreapta)&&(!ret))
  {
    if(j<dreapta)
      /*selectează pe cel mai mic fiu*/
      if (h[j].cheie>h[j+1].cheie)  j=j+1;
      if(temp.cheie>h[j].cheie)
      {
        /*deplasează fiul selectat în locul tatălui sau*/
        h[i]=h[j];
        /*avansează pe nivelul următor al ansamblului*/
        i=j; j=2*i;
      }
    else
      ret=true; /*locul a fost găsit*/
  } /*while*/
  /*plasează elementul la locul său în ansamblu*/
  h[i]= temp;
} /*deplasare*/

```

- Se observă că de fapt s-a definit un nou **tip de date abstract** numit **ansamblu** ("heap").

- **TDA Ansamblu** constă din **modelul matematic** descris de un **arbore binar parțial ordonat** peste care s-a definit **operatorul specific *deplasare (stanga, dreapta)***.
 - Acest subiect va fi reluat în cadrul capitolului 6.
- **R.W. Floyd** a conceput o metodă de a **construi un ansamblu in situ**, utilizând **TDA ansamblu** și operatorul ***deplasare*** prezentat mai sus:
 - Se consideră un tablou h_1, \dots, h_n care conține cele n elemente din care se va construi ansamblul.
 - În mod evident, elementele $h_{n/2}, \dots, h_n$ formează deja un **ansamblu** deoarece **nu** există nici o pereche de indici i și j care să satisfacă relația $j=2*i$ (sau $j=2*i+1$).
 - Aceste elemente formează cea ce poate fi considerat drept **șirul de bază al ansamblului** asociat.
 - În continuare, ansamblul $h_{n/2}, \dots, h_n$ este **extins spre stânga**, la fiecare pas cu câte un element, introdus în vârful ansamblului și deplasat până la locul său.
 - Prin urmare, considerând că tabloul inițial este memorat în h , procesul de generare "*in situ*" al unui **ansamblu** poate fi descris prin secvența [3.2.5.d].

/*Faza creare ansamblu*/

```

stanga=(n DIV 2)+1;                                [3.2.5.d]
cât timp stanga>1
|   stanga=stanga-1; /*extindere spre stânga ansamblu*/
|   deplasare(stanga,n);
|
□

```

- Același proces de generare "*in situ*" al unui **ansamblu** este reprezentat grafic în fig.3.2.5.f.

Construcție ansamblu

ansamblu h:

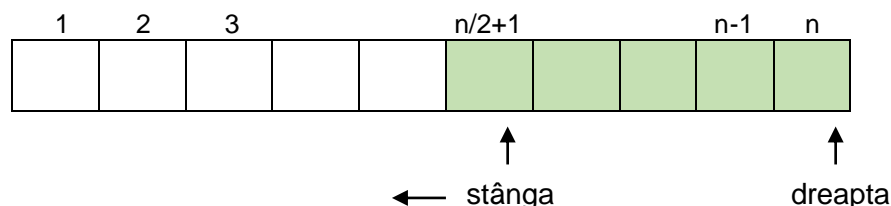


Fig. 3.2.5.f. Construcția unui ansamblu

- Odată finalizată construcția ansamblului se pune **problema sortării elementelor** componente în baza metodei lui **Williams** respectând constrângerea "*in situ*". În acest scop se utilizează tot operatorul ***deplasare***.

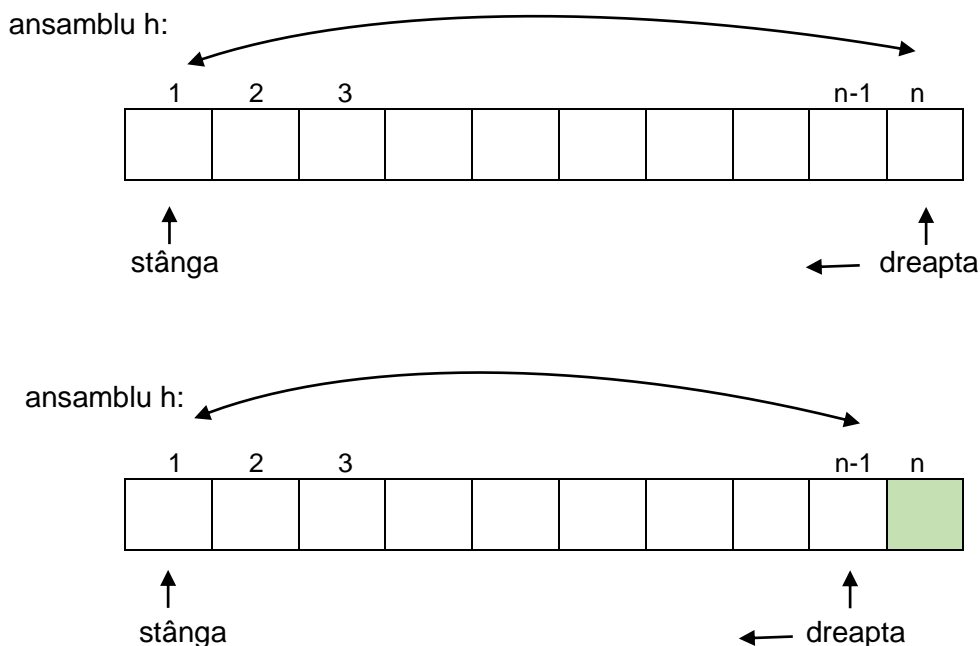
- În vederea **sortării elementelor**, se execută n pași de **deplasare**, după fiecare pas selectându-se vârful ansamblului.
- **Problema** care apare este aceea a **locului** în care se **memorează** vârfurile consecutive ale ansamblului, respectiv elementele sortate, respectând constrângerea "*in situ*".
- Această problemă poate fi rezolvată astfel:
 - În fiecare pas al procesului de sortare se **interschimbă** ultima componentă curentă a ansamblului cu componenta aflată în vârful acestuia ($h[1]$).
 - După fiecare astfel de interschimbare ansamblul se **restrânge la dreapta** cu o componentă.
 - În continuare se lasă componenta din vârf ($h[1]$) să se **deplaseze** spre locul său în ansamblu și se reia procesul.
 - În final rezultă tabloul sortat în ordine descrescătoare.
- În termenii operatorului **deplasare** această tehnică poate fi descrisă ca în secvența [3.2.5.e].

{Faza sortare ansamblu}

```
dreapta=n;
cât timp dreapta>1                                [3.2.5.e]
|  *interschimba h[1] cu h[dreapta];
|  dreapta=dreapta-1; /*restrângere la dreapta ansamblu*/
|  deplasare(1,dreapta);
|  □
```

- Reprezentarea grafică a procesului de sortare in situ a unui ansamblu apare în figura 3.2.5.g.

Sortare ansamblu



ansamblu h:

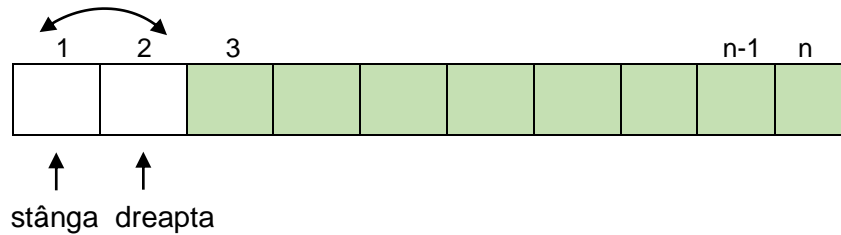


Fig. 3.2.5.g. Construcția unui ansamblu

- Cheile se obțin sortate în **ordine inversă**, lucru care poate fi ușor remediat modificând sensul relațiilor de comparație din cadrul procedurii **deplasare**.
- Rezultă următorul algoritm care ilustrează **tehnica de sortare heapsort** [3.2.5.f].

/*Sortare prin metoda ansamblelor - Heapsort - Varianta C*/

```

void heapsort();                               /*[3.2.5.f]*/
tip_element h[dimensiune_ansamblu] /*definire ansamblu*/

static void deplasare1(tip_indice* stanga, tip_element*
                        temp, tip_indice* dreapta)
/*stânga și dreapta sunt limitele ansamblului*/
/*temp este variabila utilizată la interschimbare*/
/*elementul de deplasat se află în poziția h[stanga]*/

{
    tip_indice i,j; /*indici parcurgere*/
    boolean ret;    /*indicator terminare deplasare*/

    i=*stanga;      /*indică elementul care se deplasează
                     (vârful ansamblului)*/
    j=2*i;           /*indică fiul stâng al elementului curent*/
    *temp=h[i];      /*elementul care se deplasează*/
    ret=false;      /*inițializare indicator terminare*/
    /*deplasare în ansamblu*/
    while ( (j<=*dreapta) && (!ret) )
    {
        if (j<*d)
            /*selectează pe cel mai mic fiu*/
            if (h[j].cheie<h[j+1].cheie)    j=j+1;
        if (temp->cheie<h[j])
        {
            /*deplasează fiul selectat în locul tatălui sau*/
            h[i]=h[j];
            /*avansează pe nivelul următor al ansamblului*/
            i=j; j=2*i;
        }
        else

```



```

        ret=true; /*locul a fost găsit*/
    } /*while*/
    /*plasează elementul la locul său în ansamblu*/
    h[i]= *temp;
} /*deplasare1*/

void heapsort()
{
    tip_indice stanga,dreapta; tip_element temp;

    /*faza construcție ansamblu*/
    stanga=(n/2)+1; dreapta=n;
    while(stanga>1)
    {
        stanga=stanga-1; /*extindere spre stânga ansamblu*/
        deplasare1(&stanga,&temp,&dreapta);
    } /*while*/
    /*faza sortare ansamblu*/
    while(dreapta>1)
    {
        temp=h[1]; h[1]=h[dreapta]; h[dreapta]=temp;
        dreapta=dreapta-1; /*restrângere la dreapta ansamblu*/
        deplasare1(&stanga,&temp,&dreapta);
    } /*while*/
} /*heapsort*/
-----

```

3.2.5.1. Analiza metodei heapsort

- La prima vedere nu rezultă în mod evident faptul că această metodă conduce la rezultate bune.
- **Analiza detaliată a performanței metodei heapsort** contrazice însă această părere.
 - (1) În **faza de construcție a ansamblului** sunt necesari $n/2$ pași de deplasare.
 - În **fiecare pas** se mută elemente de-a lungul a respectiv $\log_2(n/2), \log_2(n/2 + 1), \dots, \log_2(n-1)$ poziții, (în cel mai defavorabil caz), unde logaritmul se ia în baza 2 și se trunchiază la prima valoare întreagă.
 - (2) În continuare, **faza de sortare** necesită $n-1$ deplasări fiecare cu cel mult respectiv $\log_2(n-1), \log_2(n-2), \dots, 1$ mișcări.
 - (3) În plus mai sunt necesare $3 \cdot (n-1)$ mișcări pentru a **așeza** elementele sortate în ordine.
- Toate acestea dovedesc că în cel mai defavorabil caz, tehnica **heapsort** are nevoie de un număr de pași de ordinul $O(n \cdot \log_2 n)$ [3.2.5.g].

$$O(n/2 \cdot \log_2(n-1) + (n-1) \cdot \log_2(n-1) + 3 \cdot (n-1)) = O(n \cdot \log_2 n) \quad [3.2.5.g]$$

- Este greu de determinat cazul cel mai defavorabil și cazul cel mai favorabil pentru această metodă.
 - În general însă, **tehnica heapsort** este mai eficientă în cazurile în care elementele sunt într-o mai mare măsură sortate în ordine inversă.
- **Numărul mediu de mișcări** este aproximativ egal cu $1/2 \cdot n \cdot \log_2 n$, adică o mișcare la 2 pași de sortare, deviațiile de la această valoare fiind relativ mici.
- În manieră specifică metodelor de sortare avansate, valorile mici ale numărului de elemente n , **nu** sunt suficient de reprezentative, eficiența metodei crescând o dată cu creșterea lui n .
- În secvența [3.2.5.h] se prezintă varianta C a algoritmului de sortare prin metoda ansamblelor.

/*Sortare prin metoda ansamblelor -heapsort- varianta 1 C*/

/*[3.2.5.h]*/

int a[n]; **/*definire ansamblu*/**

deplasare(**int** stanga,**int** dreapta)

```
{
  int i=stanga,j=2*stanga,x=a[i-1],ret=0;
  while(j<=dreapta && !ret)
  {
    if(j<dreapta && a[j-1]<a[j])j++;
    (x<a[j-1])?(a[i-1]=a[j-1],i=j,j=2*i):(ret=1);
  }
  a[i-1]=x;
}
```

heapsort() {

```
/*construcție ansamblu*/
int stanga=n/2+1,dreapta=n;
while(stanga-1)deplasare(--stanga,n);
/*sortare ansamblu*/
while(dreapta-1)
{
  int x=a[0]; a[0]=a[dreapta-1]; a[dreapta-1]=x;
  deplasare(1,--dreapta);
}
}
```

3.2.6. Sortarea prin partiționare. Sortarea quicksort

- După cum s-a demonstrat, **metoda de sortare bubblesort** bazată pe principiul interschimbării este cea mai puțin performantă dintre metodele de sortare studiate.
- **C.A.R. Hoare** însă, pornind de la același principiu, a conceput o metodă de sortare cu performanțe spectaculare pe care a denumit-o **quicksort** (sortare rapidă).
- Metoda se bazează pe aceeași idee de a crește eficiența interschimbărilor prin mărirea distanței dintre elementele implicate.

- **Sortarea prin partiționare** se bazează pe următorul **algoritm**:
 - Fie x un **element oarecare** al tabloului de sortat a_1, \dots, a_n .
 - Se parcurge tabloul de la **stânga spre dreapta** până se găsește primul element $a_i > x$.
 - În continuare se parcurge tabloul de la **dreapta spre stânga** până se găsește primul element $a_j < x$.
 - Se **interschimbă** între ele elementele a_i și a_j .
 - Se continuă parcurgerea tabloului de la **stânga** respectiv de la **dreapta**, din punctele în care s-a ajuns anterior, până se găsesc alte două elemente care se interschimbă, ș.a.m.d.
 - Procesul se termină când cele două parcurgeri se "**întâlnesc**" undeva în interiorul tabloului.
 - Procedul de partiționare este ilustrat grafic în figura 3.2.6.a.

Partitionarea unui interval de tablou

tabloul a:

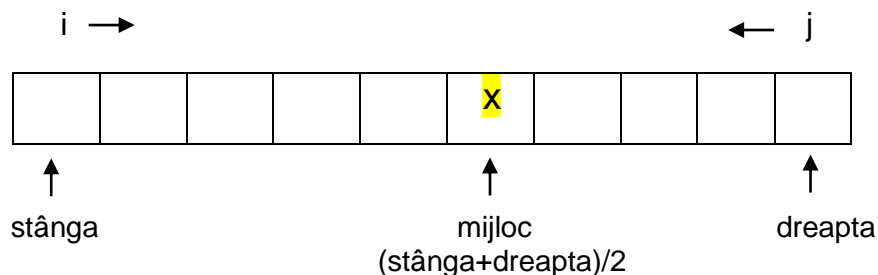


Fig. 3.2.6.a. Partiționarea unui interval de tablou

- **Efectul final** este acela că tabloul inițial este **partiționat** într-o **partiție stânga** cu chei mai mici decât x și o **partiție dreapta** cu chei mai mari decât x .
- Considerând elementele șirului memorate în tabloul a , **principiul partiționării** apare prezentat sintetic în [3.2.6.a].

/*Partiționarea unui tablou - varianta pseudocod - pas 1 de rafinare*/

```
procedure Partiționare /*[3.2.6.a]*/
/*partiționează tabloul a[stânga..dreapta]*/
```

```
  *selectează elementul x (de regulă de la mijlocul
                        intervalului de partiționat);
```

```
  repetă
```

```
    *caută primul element a[i]>x, parcurgând
      intervalul de la stânga la dreapta;
```

```
    *caută primul element a[j]<x, parcurgând
      intervalul de la dreapta la stânga;
```

```

    dacă i<=j atunci
        *interschimbă pe a[i] cu a[j];
    □
până când parcurgerile se întâlnesc (i>j);
□

```

- Înainte de a trece la sortarea propriu-zisă, se dă o formulare mai precisă **partiționării** în forma unei proceduri [3.2.6.b].
 - Se precizează că relațiile $>$ respectiv $<$ au fost înlocuite cu \geq respectiv \leq ale căror negate utilizate în instrucțiunile **WHILE** sunt $<$ respectiv $>$.
 - În acest caz x joacă rol de **fanion** pentru ambele parcurgeri.
-

/*Procedura Partiționare - varianta pseudocod - pas 2 de rafinare*/

```

PROCEDURE Partiționare(stanga, dreapta);    /*[3.2.6.b]*/
    tip_element a[n], x;

[1]  i=stanga; j=dreapta;                    /*inițializare indici*/
[2]  x=a[stanga+dreapta DIV 2]; /*setare pivot*/
[3]  repeta
[4]      cât timp (a[i].cheie<x.cheie) i=i+1;
[5]      cât timp (a[j].cheie>x.cheie) j=j-1;
[6]      dacă (i<=j)
[7]          *interschimba a[i] cu a[j];
[8]          i=i+1; j=j-1;
        □
[9]  până când (i>j)
        □
    } /*Partiționare*/

```

- În continuare, cu ajutorul partiționării, **sortarea** se realizează simplu:
 - După o primă partiționare a secvenței de elemente se aplică același procedeu celor două partiții rezultate.
 - Apoi celor patru partiții ale acestora, ș.a.m.d.
 - Procesul se termină când fiecare partiție se reduce la un singur element.
 - **Tehnica sortării bazată pe partiționare** este ilustrată în secvența [3.2.6.c].
-

***Sortarea prin partiționare -quicksort - varianta pseudocod**

```

procedure QuickSort(stanga,dreapta)                [3.2.6.c]
    *partiționează intervalul stanga,dreapta față de Mijloc;
    dacă există partiție stânga atunci
        QuickSort(stanga,Mijloc-1);
    dacă există partiție dreapta atunci
        QuickSort(Mijloc+1,dreapta);

```

- În secvența [3.2.6.e] apare o **implementare a sortării Quicksort** în limbajul C.

```
-----
/*Sortarea prin partiționare - quicksort*/
/*[3.2.6.e]*/

int a[n]; /*definire tablou de sortat*/
quicksort(int stanga,int dreapta) {
    int i=stanga; /*inițializare indice interval stânga*/
    int j=dreapta; /*inițializare indice interval dreapta*/
    int x=a[(stanga+dreapta)/2]; /*setare pivot*/

    /*partiționarea interval curent stânga, dreapta*/
    do {
        /*caută primul element a[i]>x, parcurgând intervalul de
        la stânga la dreapta*/
        while(a[i]<x) i++;
        /*caută primul element a[j]<x, parcurgând intervalul de
        la dreapta la stânga*/
        while(a[j]>x) j--;
        if(i<=j)
        { /*interschimbare a[i] cu a[j]*/
            int temp=a[i]; a[i]=a[j]; a[j]=temp;
            /*avans indici parcurgere*/
            i++;j--;
        }
    } while(i<=j);
    /*sortare quicksort*/
    if(s<j) quicksort(stanga,j); /*procesare partiție stânga*/
    if(d>i) quicksort(i,dreapta); /*procesare partiție dreapta*/
} /*quicksort*/
-----
```

- În continuare se prezintă o manieră de implementare a aceluiași algoritm utilizând o procedură **nerecursivă**.
- Elementul cheie al **soluției iterative** rezidă în menținerea unei **liste a cererilor de partiționare**.
 - La fiecare trecere apar două noi partiții.
 - Una dintre ele se prelucrează imediat, cealaltă se amână, prin memorarea ei ca cerere într-o listă.
 - În mod evident, lista de cereri trebuie rezolvată în sens invers, adică prima solicitare se va rezolva ultima și invers.
 - Cu alte cuvinte lista se tratează ca o **stivă**, de fapt ea chiar este o stivă.
 - În secvența [3.2.6.f] apare o schiță de principiu a acestei implementări în variantă pseudocod.

```
-----
/*Sortare QuickSort. Implementare nerecursivă - Varianta
pseudocod*/
```

```
procedure QuickSortNerecursiv; [3.2.6.f]
    *se introduc în stivă limitele intervalului inițial
    de sortare (amorsarea procesului);
    repetă
        *se extrage intervalul din vârful stivei care
```

```

        devine IntervalCurent;
        *se reduce stiva cu o poziție;
    repetă
    |   repetă
    |   |   *se partiționează IntervalCurent;
    |   |   până când terminare partiționare;
    |   |
    |   |   dacă există interval drept atunci
    |   |   |   *se introduc limitele sale în stivă;
    |   |   |
    |   |   *se face intervalul stâng IntervalCurent;
    |   |   până când intervalul ajunge de lățime 1 sau 0;
    |   |
    |   până când stiva se golește;
    □
□

```

- În procedura care implementează algoritmul din secvența anterioară [3.2.6.f]:
 - (1) O **cerere de partiționare** este reprezentată printr-o **pereche de indici** care delimitează **zona de tablou** ce urmează a fi partiționată.
 - (2) **Stiva** este modelată cu ajutorul unui **tablou** cu dimensiunea variabilă numit stiva și de un index *is* care precizează vârful acesteia [3.2.6.g].
 - (3) Dimensiunea maximă *m* a stivei va fi discutată pe parcursul analizei metodei.
-

```

/*Sortare QuickSort.Implementare nerecursivă*/
/*[3.2.6.g]*/
int a[n];          /*tablou de sortat*/
void qsort_nrec()
{
    enum {m=31};    /*dimensiune maximă stivă*/
    int i,j;        /*indici partiționare*/
    int stanga,dreapta; /*limite interval curent*/
    tip_element temp; /*variabilă interschimbare*/
    tip_element x;    /*variabilă pivot*/
    unsigned char is; /*indicator stivă*/
    /*definire stivă*/
    struct {
        int stanga,dreapta; /*limite interval*/
    } stiva[m];

    /*amorsare proces, intervalul inițial în stivă*/
    is=0; stiva[0].stanga=0; stiva[0].dreapta=n-1;

    /*se procesează cererea din vârful stivei*/
    do {
        stanga=stiva[is].stanga; /*limita stânga interval*/
        dreapta=stiva[is].dreapta; /*limite dreapta interval*/
        is=is-1; /*eliminare cerere din stivă*/
        /*partiționare intervalul curent a[stanga], a[dreapta]*/
        do
        {
            i=stanga; j=dreapta; /*setare indici partiționare*/
            x=a[(stanga+dreapta)/2]; /*setare pivot*/

```

```

do
{
    /*caută primul element a[i]>x, parcurgând intervalul
    de la stânga la dreapta*/
    while(a[i].cheie<x.cheie)    i=i+1;
    /*caută primul element a[j]<x, parcurgând intervalul
    de la dreapta la stânga*/
    while(x.cheie<a[j].cheie)    j=j-1;
    if(i<=j)
    {
        /*interschimbare a[i] cu a[j]*/
        temp=a[i]; a[i]=a[j]; a[j]=temp;
        /*avans indici parcurgere*/
        i=i+1; j=j-1;
    }
} while(!(i>j));
if(i<dreapta)
{
    /*partiția dreapta se introduce în stivă*/
    is=is+1; /*avans indicator stivă*/
    stiva[is].stanga=i;          /*limita stânga*/
    stiva[is].dreapta=dreapta; /*limita dreapta*/
}
dreapta=j; /*partiția stânga devine curentă*/
} while(!(stanga>=dreapta)); /*sfârșit partiționare
                                interval curent*/
} while(!(is==--1)); /*stiva este vida*/
} /*quicksort_nerecursiv*/
-----

```

3.2.6.1. Analiza metodei quicksort

- Pentru a analiza performanța acestei metode, se analizează mai întâi **partiționarea**.
 - Se presupun pentru simplificare următoarele **precondiții**:
 - (1) Setul ce urmează a fi partiționat constă din n chei **distincte** și **unice** cu valorile $\{1, 2, 3, \dots, n\}$.
 - (2) Dintre cele n chei a fost selectată cheia cu valoarea x în vederea partiționării.
 - În consecință această cheie ocupă a **x -a poziție** în mulțimea ordonată a cheilor și ea poartă denumirea de **pivot**.
- Se ridică următoarele **întrebări**:
- (1) Care este **probabilitatea** ca după ce a fost selectată cheia cu **valoarea** x ca **pivot**, o cheie oarecare a partiției să fie **interschimbată** ?
 - Pentru ca o cheie să fie interschimbată ea trebuie să fie mai mare sau egală ca x .
 - Sunt $n-x+1$ chei mai mari sau egale ca x .

- Rezultă că **probabilitatea ca o cheie oarecare să fie interschimbată** este $(n-x+1)/n$ (raportul dintre numărul de cazuri **favorabile** și numărul de cazuri **posibile**).
- (2) Care este **numărul de interschimbări** necesar pentru o partiționare a unei secvențe de n chei, dacă s-a selectat ca **pivot** cheia situată pe **poziția** x ?
 - La dreapta pivotului există $n-x$ poziții care vor fi procesate în procesul de partiționare.
 - **Numărul posibil de interschimbări** în acest context este deci egal cu **produsul** dintre numărul de chei care vor fi selectate $(n-x)$ și probabilitatea determinată anterior ca o cheie selectată să fie interschimbată [3.2.6.h].

$$NrInt = (n-x) \cdot \frac{(n-x+1)}{n} \quad [3.2.6.h]$$

- (3) Care este **numărul mediu de interschimbări** pentru partiționarea unei **secvențe de n chei**?
 - La o partiționare poate fi selectată ca și pivot **oricare** din cele n chei, deci x poate lua orice valoare cuprinsă între 1 și n .
 - **Numărul mediu M de interschimbări** pentru **partiționarea unei secvențe de n chei** se obține astfel:
 - (1) Pentru fiecare valoare a lui x selectat ca pivot, cuprinsă între 1 și n se determină numărul de interschimbări $NrInt_x$.
 - (2) Se însumează toate numerele de interschimbări pentru toate valorile lui x anterior determinate.
 - (3) Se împarte suma obținută la numărul total de chei n [3.2.6.i].

$$M = \frac{1}{n} \cdot \sum_{x=1}^n NrInt = \frac{1}{n} \cdot \sum_{x=1}^n (n-x) \cdot \frac{(n-x+1)}{n} = \frac{n}{6} - \frac{1}{6 \cdot n} \approx \frac{n}{6} \quad [3.2.6.i]$$

- Concluzia este că pentru **partiționarea unei secvențe de n chei** sunt necesare în medie **$n/6$ interchimbări**.
- Presupunând în mod exagerat că întotdeauna va fi selectată **mediana** partiției (mijlocul său valoric), fiecare partiționare va divide tabloul în două jumătăți egale.
 - Se face precizarea că **mediana** este elementul situat ca și valoare în mijlocul partiției, dacă aceasta este ordonată.

- În aceste condiții, pentru a realiza sortarea, sunt necesare un **număr de $\log_2 n$ treceri prin toate elementele tabloului** de dimensiune n (fig.3.2.6.a).
- După cum rezultă din figura 3.2.6.a, pentru un tablou de 15 elemente sunt necesare $\lceil \log_2 15 \rceil = 4$ treceri prin toate elementele tabloului sau 4 pași de partiționare integrală a tabloului.

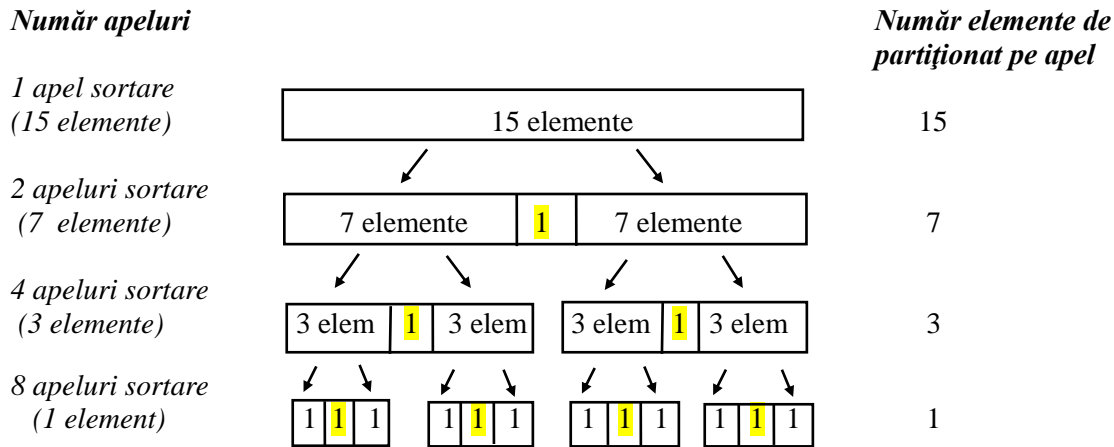


Fig. 3.2.6.a. Funcționarea principală a sortării prin partiționare

- Din păcate **numărul de apeluri recursive** ale procedurii este egal cu 15, adică exact cu numărul de elemente.
- Rezultă că **numărul total de comparații** este $n \cdot \log_2 n$ deoarece la o trecere sunt comparate **toate** cheile [3.2.6.j].
- **Numărul de mișcări** este $n/6 \cdot \log_2 n$ deoarece conform formulei [3.2.6.i] la partiționarea a n chei sunt necesare în medie $n/6$ mișcări [3.2.6.j].

$C = n \cdot \log_2 n$	$M = \frac{1}{6} \cdot n \cdot \log_2 n$	[3.2.6.j]
------------------------	--	-----------

- Aceste rezultate sunt **excepțional** de bune, dar se referă numai la **cazul optim** în care s-a presupus că la fiecare trecere se selectează **mediana**, eveniment care de altfel are probabilitatea doar $1/n$.
- Marele succes al **algoritmului quicksort** se datorește însă faptului surprinzător că **performanța sa medie**, la care **alegerea pivotului se face la întâmplare**, este inferioară performanței optime doar cu un factor egal cu $2 \cdot \ln(2) = 1.4$ deci cu aproximativ 40 % [Wi76].
- Tehnica prezentată are însă și **dezavantaje**.

- (1) În primul rând ca și la toate metodele de sortare avansate, performanțele ei sunt **moderate** pentru valori mici ale lui n .
 - Acest dezavantaj poate fi contracarat prin încorporarea unor **metode de sortare directe** pentru partițiile mici, lucru realizabil relativ simplu la această metodă în raport cu alte metode avansate.
- (2) Un al doilea dezavantaj, se referă la **cazul cel mai defavorabil** în care performanța metodei scade catastrofal.
 - Acest caz apare când la fiecare partiționare este selectată cea mai mare valoare ca și pivot.
 - În acest caz, fiecare pas va partaja secvența formată din n elemente, într-o partiție stânga cu $n-1$ elemente și o partiție dreapta cu un singur element.
 - Vor fi necesare astfel n partiționări în loc de $\log(n)$, iar performanța obține valori de ordinul $O(n^2)$.
- În mod aparent, elementul esențial al acestei metode îl reprezintă **selecția pivotului** x [GG78].
- În exemplul prezentat, pivotul a fost ales la **mijlocul partiției**.
 - El poate fi însă ales la extremitatea stângă sau dreaptă a acesteia, situație în care, cazul cel mai defavorabil îl reprezintă partiția deja sortată.
- **Tehnica quicksort** se comportă straniu:
 - Are **performanțe slabe** în cazul sortărilor banale.
 - Are **performanțe deosebite** în cazul tablourilor dezordonate.
- De asemenea, dacă x se alege întotdeauna la mijloc (mediana), atunci **tabloul sortat invers** devine **cazul optim** al sortării quicksort.
 - De fapt performanța medie este cea mai bună în cazul alegerii **pivotului la mijlocul partiției**.
- **Hoare** sugerează ca alegerea să se facă:
 - (1) Fie la "întâmplare".
 - (2) Fie prin selecția medianei unui număr redus de chei (spre exemplu trei chei).
 - O astfel de alegere judicioasă a pivotului influențează serios în mod **negativ performanța medie** a algoritmului quicksort, dar **îmbunătățește** în mod considerabil **performanța cazului cel mai defavorabil**.
- Pentru programator, în multe situații **cazul cel mai defavorabil** are o influență deosebită.

- Spre exemplu în secvența [3.2.6.g] care implementează **sortarea quicksort** în manieră **iterativă**, în cazul cel mai defavorabil, la fiecare partiționare rezultă o partiție dreapta **cu un singur element**, a cărei cerere de sortare se introduce în stivă.
- Este evident că în acest caz, **dimensiunea maximă a stivei** trebuie să fie egală cu numărul de elemente n , situație care nu este acceptabilă.
- Acest lucru este și mai grav în cazul **recursivității** unde stiva gestionată în mod automat este mult mai substanțială, fiind necesar câte un nod pentru fiecare apel recursiv, nod care presupune spațiu de memorie pentru stocarea valorilor parametrilor locali ai apelului la care se adaugă de regulă și codul efectiv al procedurii.
- Această situație se poate rezolva în cazul **implementării iterative**, introducând întotdeauna în stivă **cererea de sortare a partiției mai mari** și continuând cu partiționarea partiției mai mici.
 - În acest caz dimensiunea m a stivei va fi limitată la $m = \log_2 n$.
- Pentru a implementa această tehnică, secvența [3.2.6.g] se modifică în porțiunile în care se procesează cererile de partiționare conform [3.2.6.k].

/*Reducerea dimensiunii stivei în implementarea iterativă a sortării Quicksort*/

```

if (j-stanga<dreapta-i)                                /*[3.2.6.k]*/
{
    if (i<dreapta) /*partiția dreapta este mai mare decât
                    stanga*/
    {    /*cerere sortare partiție dreapta în stivă*/
        is=is+1; stiva[is].stanga=i;
        stiva[is].dreapta=dreapta;
    }
    dreapta=j    /*se continuă sortarea partiției stânga*/
}
else
{
    if (stanga<j)    /*partiția stânga este mai mare*/
    {    /*cerere sortare partiție stânga în stivă*/
        is=is+1; stiva[is].stanga=stanga;
        stiva[is].dreapta=j;
    }
    stanga=i    /*se continuă sortarea partiției dreapta*/
}

```

3.2.7. Determinarea medianei

- **Mediana** a n elemente este definită ca fiind acel element care este mai mic (sau egal) decât **jumătate** din elemente și este mai mare (sau egal) decât cealaltă jumătate.
 - Spre exemplu mediana secvenței 16, 12, 99, 95, 18, 87, 10 este 18.

- Problema aflării medianei este corelată direct cu cea a **sortării** deoarece, o metodă sigură de a determina **mediana** este următoarea:
 - (1) Se **sortează** cele n elemente.
 - (2) Se **extrage** elementul din mijloc.
- **Tehnica partiționării** poate însă conduce la o metodă generală mai rapidă, cu ajutorul căreia se poate determina cel de-al **k -lea element** ca valoare dintre n elemente.
 - Găsirea **medianei** reprezintă cazul special $k=n/2$.
 - În același context, $k=1$ precizează aflarea **minimului**, iar $k=n$, aflarea **maximului**.
- **Algoritmul pentru determinarea celui de-al k -lea element** conceput de **C.A.R. Hoare** funcționează după cum urmează.
 - Se presupune că elementele avute în vedere sunt memorate în **tabloul** a cu dimensiunea n .
 - Pentru început se realizează o **partiționare** cu limitele $stanga=0$, $dreapta=n-1$ și cu $a[k]$ selectat pe post de pivot x .
 - În urma acestei **partiționări** rezultă valorile index i și j care satisfac relațiile [3.2.7.a].

-
- 1) $x=a[k]$
 - 2) $a[h] \leq x$ pentru toți $h < i$
 - 3) $a[h] \geq x$ pentru toți $h > j$
 - 4) $i > j$
-

- Sunt posibile **trei situații** evidențiate în fig.3.2.7.a.

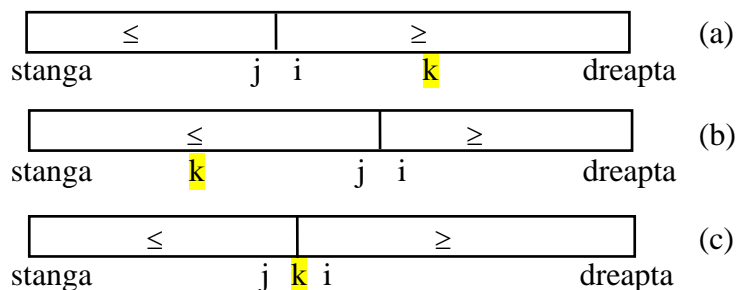


Fig. 3.2.7.a. Determinarea medianei

- (1) Valoarea pivotului x este prea **mică**, astfel încât limita dintre cele două partiții este sub valoarea dorită k .
 - Procesul de partiționare se reia pentru elementele **partiției dreapta** $a[i], \dots, a[dreapta]$ (fig.3.2.7.a (a)).

- (2) Valoarea pivotului x este prea **mare**.
 - Operația de partiționare se reia pentru elementele **partiției stânga** $a[\text{stanga}], \dots, a[j]$ (fig.3.2.7.a (b)).
- (3) $j < k < i$.
 - În acest caz elementul $a[k]$ separă tabloul în două partiții, el desemnând **mediana** (fig.3.2.7.a (c)).
- Procesul de partiționare se repetă până la realizarea cazului (3).
- Algoritmul aferent este prezentat în variantă pseudocod în secvența [3.2.7.b] respectiv o primă rafinare în secvența de program [3.2.7.c].

/*Aflarea medianei - Varianta pseudocod*/

```

procedure Mediana (stanga,dreapta,k)                                [3.2.7.b]
  cât timp există partiție
    *alege pivotul (elementul din poziția k);
    *partiționează intervalul curent față de valoarea
      Pivotului;
    dacă limită partiționare<k atunci *selectează
      partiția dreapta;
    dacă limită partiționare>k atunci *selectează
      partiția stânga;
  □

```

{Procedura Mediana - Pas rafinare 1}

```

  stanga=0; dreapta=n-1;                                            [3.2.7.c]
  cât timp (s<d)
    x=a[k];
    *se partiționează a[stanga]..a[dreapta] față de a[k]
    if (j<k) stanga=i;
    if (k<i) dreapta=j;
  □

```

-
- Programul aferent apare în secvența [3.2.7.d].

/*Procedura Mediana - Implementare C*/

```

                                                                    /*[3.2.7.d]*/
void mediana (int k)
  /*determină cea de-a k componentă ca valoare dintre
  elementele tabloului a*/
  {
    tip_element a[n];      /*elemente inițiale*/
    tip_element x;         /*pivot partiționare*/
    tip_element temp;      /*element interschimbare*/
    int stanga,dreapta;    /*limite partiție*/
    int i,j;               /*indici partiționare*/

    stanga=0; dreapta=n-1; /*setare limite partiție*/
    /*partiționare interval stânga dreapta*/
    while(stanga<dreapta)
    {
      /*inițializare indici partiționare*/
      i=stanga; j=dreapta;

```

```

x=a[k]; /*setare pivot partiționare*/
/*partiționarea în raport cu a[k]*/
do
{
    while(a[i]<x) i++;
    while(x<a[j]) j--;
    if (i<=j)
    {
        temp=a[i]; a[i]=a[j]; a[j]=temp;
        i++; j--;
    }
} while (!(i>j));
if(j<k) stanga=i; /*selectează partiția stânga*/
if(k<i) dreapta=j; /*selectează partiția dreapta*/
}
}
/*mediana*/

```

3.2.7.1. Analiza determinării medianei

- Dacă se presupune că în medie **fiecare partiționare înjumătățește partiția** în care se găsește elementul căutat, atunci **numărul necesar de comparații C** este de ordinul $O(n)$ [3.2.7.e].
-

$$C = n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2 \cdot n - 1 \quad [3.2.7.e]$$

- **Numărul de mișcări M** nu poate depăși numărul de comparații, el fiind de regulă mai mic, ca atare tot $O(n)$.
- Valorile indicatorilor C și M estimați pentru **determinarea medianei** subliniază superioritatea acestei metode.
- În același timp explică **performanța superioară** a metodei față de metodele bazate pe sortarea tabloului și extragerea celui de-al k -lea element, a căror performanță în cel mai bun caz este de ordinul $O(n \cdot \log_2 n)$.
- În cel mai **defavorabil** caz:
 - Fiecare partiționare reduce setul de candidați numai cu 1, rezultând un număr de comparații de ordinul $O(n^2)$.
 - Și în acest caz metoda **medianei** este indicată pentru valori mari ale lui n ($n > 10$).

3.2.8. Sortarea binsort. Determinarea distribuției cheilor

- În general algoritmi de sortare bazați pe metode avansate au nevoie de $O(n \cdot \log_2 n)$ pași pentru a sorta n elemente.
- Trebuie precizat însă faptul că acest lucru este valabil în situația în care:
 - Nu există nici o altă informație suplimentară referitoare la chei, decât faptul că pe mulțimea acestora este definită o **relație de ordonare**, prin intermediul căreia se poate preciza dacă valoarea unei chei este mai **mică** respectiv mai **mare** decât o alta.
- După cum se va vedea în continuare, sortarea se poate face și **mai rapid** decât în limitele performanței $O(n \cdot \log_2 n)$, **dacă**:
 - (1) Există și **alte informații** referitoare la **cheile** care urmează a fi sortate.
 - (2) Se **renunță** la constrângerea de sortare "*in situ*".
- Spre **exemplu**:
 - Se cere să se sorteze un set de n chei de tip întreg, ale căror valori sunt unice și aparțin intervalului de la 1 la n .
 - Dacă a și b sunt **tablouri** cu câte n elemente, a conținând cheile care urmează a fi sortate, atunci sortarea se poate realiza direct în tabloul b , într-o singură trecere, conform secvenței [3.2.8.a].

/*Exemplu de sortare liniară*/

```
for i=0 to n-1                                     [3.2.8.a]
  b[a[i].cheie]=a[i];                               /*O(n)*/
```

- Ideea metodei:
 - Se determină locul elementului $a[i]$ și se plasează elementul exact la locul potrivit în tabloul b .
 - Întregul ciclu necesită $O(n)$ pași.
 - Rezultatul este însă corect numai în cazul în care există **un singur element** cu cheia x , pentru fiecare valoare cuprinsă între $[0, n-1]$.
 - Un al doilea element cu aceeași cheie va fi introdus tot în $b[x]$ distrugând elementul anterior.
- Acest tip de sortare poate fi realizat și "*in situ*" (secvența [3.2.8.b])
 - Astfel, fiind dat tabloul a de dimensiune n , ale cărui elemente au respectiv cheile $0, \dots, n-1$,
 - Se baleează pe rând elementele sale (buclo **for** exterioară).

- Dacă elementul $a[i]$ are cheia j , atunci se realizează interschimbarea lui $a[i]$ cu $a[j]$.
- Fiecare interschimbare plasează elementul aflat în locația i exact la locul său în tabloul ordonat, fiind necesare în cel mai rău caz $3 \cdot n$ mișcări pentru întreg procesul de sortare.
- Secvența de program care ilustrează această tehnică apare în [3.2.8.b].

/*Sortare liniară in situ*/

```
for i=0 to n-1                                /*O(n)*/      [3.2.8.b]
  while (a[i].cheie<>i)
    *interschimba a[i] cu a[a[i].cheie];
```

- Secvențele [3.2.8.a, b] ilustrează **tehnica de sortare** numită **binsort**, în cadrul căreia se crează **bin**-uri, fiecare bin păstrând un element sortat cu o anumită cheie [AHU85].
- **Tehnica sortării** este simplă:
 - (1) Se examinează pe rând fiecare element de sortat.
 - (2) Se introduce elementul curent în **bin**-ul corespunzător valorii cheii.
 - În secvența [3.2.8.a] bin-urile sunt chiar elementele tabloului b , unde $b[i]$ este binul cheii având valoarea i .
 - În secvența [3.2.8.b] bin-urile sunt chiar elementele tabloului a după reșezare.
- Tehnica aceasta simplă și performantă se bazează pe următoarele **cerințe apriorice**:
 - (1) **Domeniul limitat** al cheilor $[0, n-1]$.
 - (2) **Unicitatea** fiecărei chei.
- Dacă cea de-a doua cerință **nu** este respectată, și de fapt acesta este cazul obișnuit, este necesar ca într-un **bin** să fie memorate **mai multe elemente** având aceeași cheie.
 - Acest lucru se realizează fie prin **însiruire**, fie prin **concatenare**, fiind utilizate în acest scop **structuri listă**.
 - Această situație **nu** deteriorează prea mult performanțele acestei tehnici, efortul de sortare ajungând egal cu $O(n+m)$, unde n este **numărul de elemente** iar m **numărul de chei**.
 - Din acest motiv, această metodă reprezintă punctul de plecare al mai multor tehnici de sortare a **structurilor listă** [AHU85].
- Spre exemplu, o metodă de rezolvare a unei astfel de situații este cea bazată pe **determinarea distribuției cheilor** ("distribution counting") [Se88].
- **Problema** se formulează astfel:

- Se cere să se sorteze un tablou cu n articole ale căror chei sunt cuprinse în intervalul $[0, m-1]$.
- Dacă m **nu** este prea mare pentru rezolvarea problemei poate fi utilizat algoritmul de "*determinare a distribuției cheilor*".
- **Ideea** algoritmului este următoarea:
 - (1) Se **contorizează** într-o primă trecere **numărul de chei** pentru fiecare valoare de cheie care apare în tabloul a .
 - (2) Se **ajustează** valorile **contoarelor**.
 - (3) Într-o a doua trecere, utilizând aceste contoare, se **mută** direct articolele în poziția lor ordonată în tabloul b .
- Formularea **algoritmului de sortare cu determinarea distribuțiilor cheilor** este cea din secvența [3.2.8.c].
 - Pentru simplificare se presupune că tabloul a conține doar chei.

```

-----
/*Sortare cu determinarea distribuțiilor cheilor - Varianta
C*/
                                                    /*[3.2.8.c]*/
enum {n=100, m=10}; /*inițializare număr elemente și număr
                     chei*/
typedef int tip_tablou[n]; /*definire tip tablou*/
int contor[m];           /*tablou contoare*/
tip_tablou a;            /*tablou de sortat*/
tip_tablou b;            /*tablou auxiliar*/
int i, j;                /*indici parcurgere*/

int main(int argc, const char* argv[])
{
    /*inițializare contoare*/
    for(j=0; j<m-1; j++) contor[j]=0;
    /*determinarea distribuției cheilor*/
    for(i=0; i<n-1; i++) contor[a[i]]=contor[a[i]]+1;
    /*ajustare contoare*/
    for(j=1; j<m-1; j++) contor[j]=contor[j-1]+contor[j];
    /*sortare cu determinarea distribuției cheilor în b*/
    for(i=n-1; i>0; i--)
    { /*tabloul a se parcurge de la dreapta la stânga*/
        b[contor[a[i]]-1]=a[i]; /*mutare element din a în b*/
        contor[a[i]]=contor[a[i]]-1; /*ajustare contor*/
    }
    /*mută tabloul b în a*/
    for(i=0; i<n-1; i++) a[i]=b[i];
    return 0;
} /*Sortare cu determinarea distribuției cheilor*/
-----

```

- Funcționarea algoritmului:
 - Contoarele asociate cheilor sunt memorate în tabloul `contor` de dimensiune m .

- Inițial locațiile tabloului `contor` sunt inițializate pe zero (prima buclă **for**).
- Se contorizează cheile în tabloul `contor` (a doua buclă **for**).
- În continuare sunt ajustate valorile contoarelor tabloului `contor` (a treia buclă **for**).
- Se parcurge tabloul `a` de la sfârșit spre început, iar cheile sunt introduse exact la locul lor în tabloul `b` cu ajutorul contoarelor memorate în tabloul `contor` (a patra buclă **for**).
- Concomitent cu introducerea cheilor are loc și **decrementarea** contoarelor specifice astfel încât în final, cheile identice apar în binul specific în **ordinea relativă** în care apar în secvența inițială.
- Ultima buclă **for** realizează **mutarea** integrală a elementelor tabloului `b` în tabloul `a`, (dacă acest lucru este necesar).
- Deși se realizează mai multe treceri prin elementele tabloului totuși în ansamblu, **performanța algoritmului de sortare baza pe determinarea distribuției cheilor** este $O(n)$.
- Aceasta metodă de sortare pe lângă faptul că este rapidă are avantajul de a fi **stabilă**, motiv pentru care ea stă la baza mai multor **metode de sortare de tip radix**.
- În continuare se prezintă un **exemplu** de funcționare a **algoritmului de sortare bazat pe determinarea distribuției cheilor**.

- **Exemplul 3.2.8.** Schematic, în vederea **sortării cu determinarea distribuției cheilor** se parcurg următorii pași.

1) Se consideră inițial că tabloul `a` are următorul conținut:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	b	a	c	a	d	a	b	b	a	d	d	a
0	1	1	0	2	0	3	0	1	1	0	3	3	0

2) Se inițializează tabloul `contor`.

0	1	2	3
0	0	0	0

3) Se contorizează valorile cheilor tabloului `a`.

0	1	2	3
6	4	1	3

4) Se ajustează valorile tabloului `contor`.

0	1	2	3
6	10	11	14

5) Se iau elementele tabloului `a` de la dreapta la stânga și se introduc pe rând în tabloul `b`, fiecare în poziția indicată de contorul propriu minus 1 din tabloul `contor`.

- După introducerea fiecărui element în tabloul `b`, contorul specific din tabloul `contor` este decrementat cu o unitate.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	a	a	a	a	a	b	b	b	b	c	d	d	d

3.2.8.1 Analiza performanței sortării cu determinarea distribuției cheilor

- Deși sunt necesare mai multe treceri pentru a sorta un tablou, performanța de ansamblu a sortării cu **determinarea distribuției cheilor** este liniară $O(n)$.
- Această metodă pe lângă faptul că este rapidă, are avantajul de a fi o **metodă de sortare stabilă**.
- Aceasta este și rațiunea pentru care această metodă stă la baza mai multor metode de sortare de tip **radix**.

3.2.9. Sortarea bazată pe baze de numerație. Radix sort

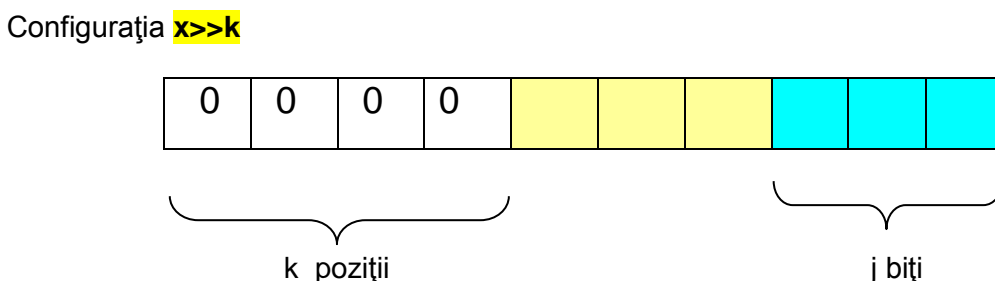
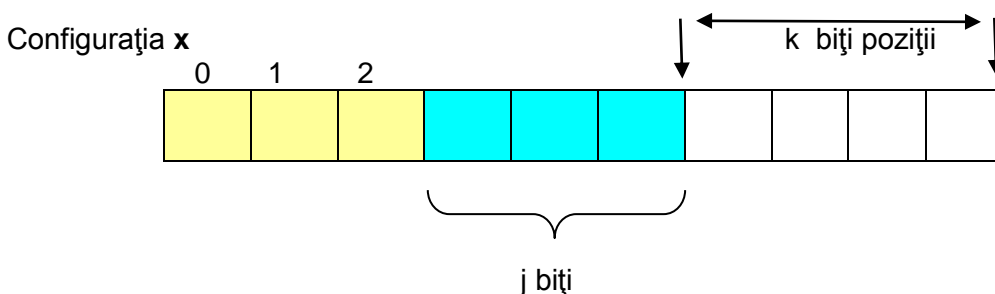
- Metodele de sortare prezentate până în prezent, concep cheile de sortat ca **entități** pe care le prelucrează integral prin comparare și interschimbare.
- În unele situații însă se poate profita de faptul că aceste chei sunt de fapt **numere** exprimate prin **cifre** aparținând unui **domeniu mărginit**.
- Metodele de sortare care iau în considerare **proprietățile digitale ale numerelor** sunt **metodele de sortare bazate pe baze de numerație** ("radix sort").
- Algoritmii de tip **bază de numerație**:
 - (1) Consideră cheile ca și numere reprezentate într-o **bază de numerație** m , unde m poate lua diferite valori ("radix").
 - (2) Procesează **cifrele individuale** ale numărului.
- Un **exemplu** sugestiv îl reprezintă sortarea unui teanc de cartele care au perforate pe ele **numere formate trei cifre**.
 - Se grupează cartelele în 10 grupe distincte, prima cuprinzând cheile mai mici decât 100, a doua cheile cuprinse între 100 și 199, etc., adică se realizează o sortare după **cifra sutelor**.
 - În continuare se sortează pe rând grupele formate aplicând aceeași metodă, după **cifra zecilor**.
 - Apoi fiecare grupă nou formată, se sortează după **cifra unităților**.
- Acesta este un exemplu simplu de **sortare radix** cu $m = 10$.
- Pentru sistemele de calcul, unde prelucrările se fac exclusiv în baza 2, se pretează cel mai bine metodele de **sortare radix** care operează cu **numere binare**.
- În general, în **sortarea radix** a unui set de numere, **operația fundamentală** este extragerea unui **set contiguu de biți** din numărul binar care reprezintă **cheia**.

- Spre **exemplu**:
 - Pentru a extrage primii 2 biți ai unui număr binar format din 10 cifre binare:
 - (1) Se realizează o **deplasare la dreapta** cu 8 poziții a reprezentării binare a numărului.
 - (2) Se operează configurația obținută, printr-o **operație "și"** cu masca 0000000011.
- Aceste operații:
 - (1) Pot fi **implementate direct** cu ajutorul facilităților de prelucrare a configurațiilor la nivel de biți puse la dispoziție de limbajele de programare.
 - (2) Pot fi **simulate** cu ajutorul operatorilor întregi **DIV** și **MOD**.
 - Spre **exemplu**, în situația anterioară, dacă x este numărul binar în cauză, primii doi biți se obțin prin expresia $(x \text{ DIV } 2^8) \text{ MOD } 2^2$.
- În continuare, pentru implementarea **sortărilor radix**, se consideră definit un **operator *biti***(int x, k, j) care combină cele două operații **returnând** valorile a j biți care apar la k poziții de la marginea dreaptă a lui x .
- O posibilă implementare în limbajul C a acestui **operator** apare în secvența [3.2.9.a].

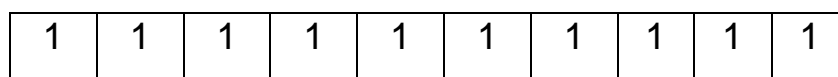
{operator care returnează j biți care apar la k poziții de marginea dreaptă a lui x }

```
unsigned biti(unsigned x, int k, int j){ /*[3.2.9.a]*/
    return (x>>k) & ~ (~0<<j); }
```

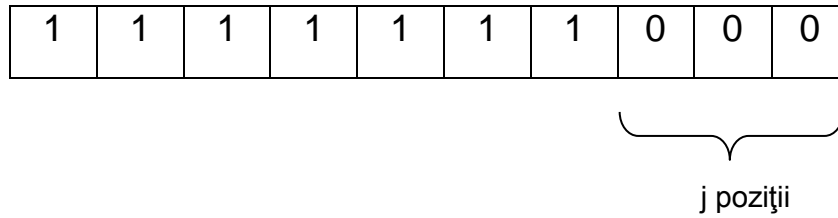
- Explicația funcționării operatorului ***biti***(x, k, j) este ilustrată în fig. 3.2.9.a.



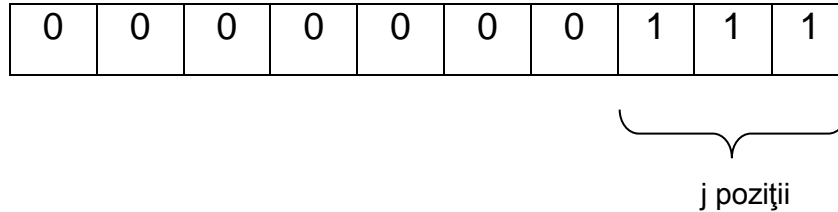
Masca ~ 0



Masca $\sim 0 \ll j$



Masca $\sim(\sim 0 \ll j)$



Configurația $(x \gg k) \& \sim(\sim 0 \ll j)$

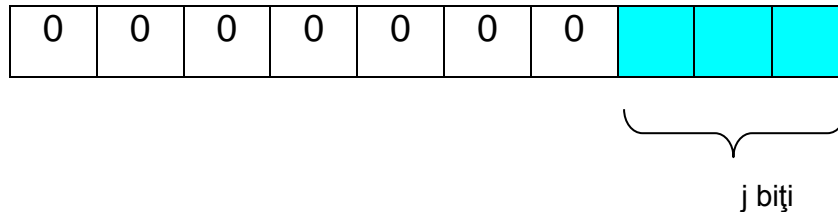


Fig. 3.2.9.a. Funcționarea operatorului *biti* (x, k, j)

- Există două metode de bază pentru implementarea sortării radix.
- (1) Prima metodă examinează biții cheilor de la stânga la dreapta și se numește sortare radix prin interschimbare ("radix exchange sort").
 - Se bazează pe observația că:
 - Rezultatul comparației a două chei este determinat de valoarea biților din prima poziție la care ele diferă în parcurgere de la stânga la dreapta.
 - Astfel, elementele ale căror chei au primul bit 0 sunt trecute în fața celor care au primul bit 1.
 - În continuare în fiecare grup astfel format se aplică aceeași metodă pentru bitul următor și așa mai departe.
 - Sortarea propriu-zisă se realizează prin schimbarea sistematică a elementelor în maniera precizată.
- (2) A doua metodă se numește sortare radix directă ("straight radix sort").
 - Ea examinează biții din cadrul cheilor de la dreapta la stânga.

- Se bazează pe principiul interesant care reduce sortarea cheilor de b -biți la b sortări ale unor chei de 1 bit.

3.2.9.1. Sortarea radix prin interschimbare ("radix exchange sort")

- **Ideea sortării radix prin interschimbare** este următoarea:
 - Se sortează elementelor tabloului a astfel încât toate elementele ale căror chei încep cu un bit zero să fie trecute în fața celor ale căror chei încep cu 1.
 - Acest proces va avea drept consecință formarea a două **partiții** ale tabloului inițial.
 - Cele două partiții la rândul lor se sortează independent, conform aceleași metode **după cel de-al doilea bit** al cheilor elementelor, rezultând 4 partiții.
 - Cele 4 partiții rezultate se sortează similar **după al 3-lea bit**, ș.a.m.d.
- Acest mod de lucru sugerează abordarea **recursivă** a implementării metodei de sortare.
- **Procesul de sortare radix prin interschimbare** se desfășoară exact ca și la **partiționare**:
 - Se balează tabloul de la **stânga spre dreapta** până se găsește un element a cărui cheie începe cu 1.
 - Se balează tabloul de la **dreapta spre stânga** până se găsește un element a cărui cheie începe cu 0.
 - Se **interschimbă** cele două elemente în tablou.
 - Procesul **continuă** până când indicatorii de parcurgere **se întâlnesc** formând două partiții.
 - Se reia aceeași procedură pentru **cel de-al doilea bit** al cheilor elementelor în cadrul **fiecăreia dintre cele două partiții rezultate** ș.a.m.d. [3.2.9.1.a].

```

/*Sortare radix prin interschimbare - Varianta C*/
/*[3.2.9.1.a]*/
tip_element a[n]; /*tabloul de sortat*/
void radix_interschimb (tip_indice stanga,tip_indice
    dreapta, int b)
/*stanga, dreapta - limitele curente ale tabloului de
    sortat*/
/*b - lungimea în biți a cheii de sortat*/
{
    tip_indice i,j; /*indici de lucru*/
    tip_element t; /*variabilă interschimbare*/

    if ((dreapta>stanga) && (b>=0) )
    {
        i=stanga; j=dreapta; /*inițializare indici de lucru*/
        b=b-1; /*setare distanță bit selectat față de marginea
            din dreapta a configurației de biți a cheii*/
        /*partiționare interval curent stânga-dreapta*/
        do
        {

```

```

    while((biti(a[i].cheie,b,1)==0)&&(i<j))    i=i+1;
    while((biti(a[j].cheie,b,1)==1)&&(i<j))    j=j-1;
    t=a[i]; a[i]=a[j]; a[j]=t; /*interschimbare*/
  } while(!(j==i));
  if(biti(a[dreapta].cheie,b,1)==0)    j=j+1; /*dacă
    ultimul bit testat este 0 se reface lungimea
    partiției*/
  /*procesează partiția stânga*/
  radix_interschimb(stanga,j-1,b-1);
  /*procesează partiția dreapta*/
  radix_interschimb(j,dreapta,b-1);
} /*if*/
}
/*radix_interschimb*/

```

- Spre exemplu, se presupune că tabloul `a[0..14]` conține chei întregi care au valoarea mai mică decât 2^5 (adică se reprezintă utilizând 5 cifre binare).
- Apelul procedurii **RadixInterschimb**(0,14,5) va realiza sortarea după cum se prezintă schematic în figura 3.2.9.1.

A	00001	A	0	0001	A	0	0	001	A	00	0	01	A	000	0	1	A	0000	1	A
S	10011	E	0	0101	E	0	0	101	A	00	0	01	A	000	0	1	A	0000	1	A
O	01111	O	0	1111	A	0	0	001	E	00	1	01	E	001	0	1	E	0010	1	E
R	10010	L	0	1100	E	0	0	101	E	00	1	01	E	001	0	1	E	0010	1	E
T	10100	M	0	1101	G	0	0	111	G	00	1	11	G	001	1	1	G	0011	1	G
I	01001	I	0	1001	I	0	1	001	I	01	0	01	I	010	0	1	I	0100	1	I
N	01110	N	0	1110	N	0	1	110	N	01	1	10	L	011	0	0	L	0110	0	L
G	00111	G	0	0111	M	0	1	101	M	01	1	01	M	011	0	1	M	0110	1	M
E	00101	E	0	0101	L	0	1	100	L	01	1	00	N	011	1	0	N	0111	0	N
X	11000	A	0	0001	O	0	1	111	O	01	1	11	O	011	1	1	O	0111	1	O
A	00001	X	1	1000	S	1	0	011	S	10	0	11	P	100	0	0	P	1000	0	P
M	01101	T	1	0100	T	1	0	100	R	10	0	10	R	100	1	0	R	1001	0	R
P	10000	P	1	0000	P	1	0	000	P	10	0	00	S	100	1	1	S	1001	1	S
L	01100	R	1	0010	R	1	0	010	T	10	1	00	T	101	0	0	T	1010	0	T
E	00101	S	1	0011	X	1	1	000	X	11	0	00	X	110	0	0	X	1100	0	X

Fig. 3.2.9.1. Sortare radix prin interschimbare

- O problemă serioasă care afectează această metodă se referă la **degenerarea partițiilor**.
 - **Degenerarea partițiilor** apare de obicei în situația în care cheile sunt reprezentate prin numere mici (care încep cu multe zerouri).
 - Această situație apare frecvent în cazul caracterelor interpretate pe 8 biți.
- Din punctul de vedere al **performanței**, **metoda de sortare radix prin interschimbare** sortează n chei de b biți utilizând un număr de comparații de biți egal cu $n \cdot b$.
 - Cu alte cuvinte, **sortarea radix prin interschimbare** este **liniară** cu **numărul de biți ai unei chei**.
 - Pentru o **distribuție normală** a biților cheilor, **sortarea radix prin interschimbare** este ceva mai rapidă decât metoda **quicksort** [Se88].

3.2.9.2. Sortarea radix directă ("straight radix sort")

- O altă variantă de implementare a **sortării radix** este aceea de a examina biții cheilor elementelor de la **dreapta la stânga**.
 - Este metoda utilizată de vechile mașini de sortat cartele.
 - Teancul de cartele trece de 80 de ori prin mașină, câte odată pentru fiecare coloană începând de la dreapta spre stânga, fiecare trecere realizând sortarea după o coloană.
- Această metodă se numește **sortare radix directă**.
- **Ideea** metodei de **sortare radix directă**:
 - Se sortează cheile după un bit examinând biții lor **de la dreapta spre stânga**.
 - Sortarea după bitul i constă în extragerea tuturor elementelor ale căror chei au zero pe poziția i și plasarea lor în fața celor care au 1 pe aceeași poziție.
 - Când se ajunge la bitul i venind dinspre dreapta, cheile sunt gata sortate pe ultimii $i-1$ biți ai lor.
- **Nu** este ușor de demonstrat că metoda este corectă: de fapt ea este corectă **numai** în situația în care **sortarea după 1 bit** este o **sortare stabilă**.
 - Datorită acestei cerințe, **interschimbarea normală nu** poate fi utilizată deoarece **nu** este o **metodă de sortare stabilă**.
- În ultimă instanță trebuie **sortat stabil** un tablou cu numai două valori 0 și 1.
- Metoda bazată pe **determinarea distribuțiilor cheilor (distribution counting)** poate fi utilizată cu succes în acest scop.
 - Se consideră **algoritmul de sortare** bazat pe **determinarea distribuției cheilor** în care:
 - (1) Se ia $m = 2$.
 - (2) Se înlocuiește $a[i].cheie$ cu ***biti***($a[i].cheie, k, 1$) - adică se extrage din cheie 1 bit situat la distanța k față de sfârșitul cheii.
 - (3) Se iterează sortarea bazată pe determinarea distribuțiilor pentru fiecare bit al cheii, de la **dreapta la stânga**, respectiv pentru toate valorile lui $k = 0, 1, 2, \dots, b-1$.
 - Se obține astfel, o metodă de **sortare stabilă** a tabloului a , după bitul k , de la dreapta la stânga, rezultatul fiind memorat într-o **tablou temporar** t .
- Rezultă că pentru **sortarea integrală a tabloului** sunt necesare b **treceri** unde b este lungimea cheii.

- Pentru a crește performanța procesului de sortare, **nu** este indicat să se lucreze cu $m=2$, ci este convenabil ca m să fie cât mai mare.
 - În acest fel se reduce numărul de treceri.
- Dacă se prelucrează m biți **odată**:
 - Timpul de sortare **scade** prin reducerea numărului de treceri.
 - Tabela de distribuții însă **crește** ca dimensiune ea trebuind să conțină $m1=2^m$ locații, deoarece cu m biți se pot alcătui 2^m configurații binare.
- În acest mod, **sortarea radix-directă** devine o **generalizare a sortării bazate pe determinarea distribuțiilor**.
- Algoritmul din secvența [3.2.9.2.a] sortează după această metodă tabloul $a[1..n]$.
 - Cheile sunt de b biți lungime.
 - Cheile sunt parcurse de la **dreapta la stânga**, procesând câte m biți odată. În consecință, pentru sortarea tabloului vor fi necesare b/m treceri.
 - Pentru sortare se utilizează tabloul suplimentar $t[1..n]$.
 - Procedura funcționează **numai** dacă b este **multiplu** de m deoarece algoritmul de sortare divizează biții cheilor într-un număr întreg de părți de dimensiuni egale care se procesează deodată.
- Dacă se ia $m=b$ se obține **sortarea bazată pe determinarea distribuțiilor**;
- Dacă se ia $m=1$ rezultă **sortarea radix directă**.
- Implementarea propusă sortează tabloul a în tabloul t și după fiecare pas de sortare recopiază tabloul t în a (ultima buclă **for**).
- Acest lucru poate fi evitat, **concatenând** în aceeași procedură **două copii** ale algoritmului de sortare: una care sortează din a în t , cealaltă din t în a .

/*Sortare radix directă - Varianta C*/

```

                                                    /*[3.2.9.2.a]*/
tip_element a[n];    /*tablou de sortat*/
tip_element t[n];    /*tablou suplimentar*/

void radix_direct()
{
    int i,j;          /*indici de lucru*/
    int trecere;      /*contor treceri*/
    int contor[m1];   /*tablou contoare m1=2m*/

    /*procesul de sortare necesită b/m treceri*/
    for(trecere=0;trecere<=(b/m)-1;trecere++)
    {
        /*inițializare contoare pentru trecerea curentă*/
        for(j=0;j<=m1-1;j++) contor[j]=0;
        /*determinarea distribuției cheilor pentru trecerea
           curentă*/
    }
}
```

```

    for (i=1; i<=n; i++)
        contor[biti(a[i].cheie, trecere*m, m)] =
            contor[biti(a[i].cheie, trecere*m, m)] + 1;
/*ajustare contoare pentru trecerea curentă*/
for (j=1; j<=m1-1; j++)
    contor[j] = contor[j-1] + contor[j];
/*sortare cu determinarea distribuției cheilor
pentru trecerea curentă în tabloul t*/
for (i=n; i>=1; i--)
{
    t[contor[biti(a[i].cheie, trecere*m, m)]] =
        a[i];
    contor[biti(a[i].cheie, trecere*m, m)] =
        contor[biti(a[i].cheie, trecere*m, m)] - 1;
}
/*mută tabloul t în a*/
for (i=1; i<=n; i++) a[i] = t[i];
}
}
/*radix_direct*/

```

- **Performanța** sortării radix directe:
 - Sortează n elemente cu chei de b biți în b/m treceri.
- **Dezavantajele** metodei de sortare radix directă:
 - (1) Utilizează un **spațiu suplimentar de memorie** pentru 2^m contoare.
 - (2) Utilizează un **buffer** pentru rearanjarea tabloului cu dimensiunea egală cu cea a tabloului original.

3.2.9.3. Performanța sortărilor radix

- **Timpul de execuție** al celor două metode de sortare radix fundamentale, pentru n elemente având chei de b biți este în esență proporțional cu $n \cdot b$.
- Pe de altă parte, timpul de execuție poate fi aproximat ca fiind $n \cdot \log_2(n)$, deoarece dacă toate cheile sunt diferite, b trebuie să fie cel puțin $\log_2(n)$.
- În realitate, nici una din metode **nu** atinge de fapt limita precizată $n \cdot b$, necesitând de fapt un efort mai redus :
 - Metoda de la **stânga la dreapta** se oprește când apare o diferență între chei.
 - Metoda de la **dreapta la stânga** poate prelucra mai mulți biți deodată.
- Sortările radix se bucură de următoarele **proprietăți** [Se88].
 - **Proprietatea 1.** Sortarea **radix prin interschimbare** examinează în medie $n \cdot \log_2(n)$ biți.

- **Proprietatea 2.** La sortarea a n chei de câte b biți, **ambele sortări radix** examinează de regulă mai puțini decât $n \cdot b$ biți.
- **Proprietatea 3.** Sortarea **radix directă** poate sorta n elemente cu chei de b biți, în b/m treceri utilizând un spațiu suplimentar pentru 2^m contoare și un **buffer** pentru rearanjarea tabloului.

3.2.9.4. Sortarea liniară

- **Sortarea radix directă**, anterior prezentată realizează b/m **treceri** prin tabloul de sortat.
- Dacă se alege pentru m o valoare **suficient de mare** se obține o metodă de sortare foarte eficientă, cu condiția să existe un spațiu suplimentar de memorie cu 2^m locații.
- O alegere rezonabilă a valorii lui m este $b/4$ (un sfert din dimensiunea cheii).
 - În acest caz, sortarea radix se reduce la 4 sortări bazate pe determinarea distribuțiilor, care sunt practic **liniare**.
- De obicei valorile $m=4$ sau $m=8$ sunt propice pentru actuala organizare a sistemelor de calcul.
 - Ele conduc la dimensiuni rezonabile ale tabloului de contoare (16 respectiv 256 de locații).
 - În consecință fiecare pas este liniar și deoarece sunt necesari numai 8 (4) pași pentru chei de 32 de biți, procesul de sortare este **practic liniar**.
 - Rezultă astfel una din **cele mai performante metode de sortare**, care concurează metoda quicksort, departajarea între ele fiind o problemă dificilă.
- **Dezavantajele** majore ale metodei sunt:
 - (1) Necesitatea **distribuției uniforme** a cheilor.
 - (2) Necesitatea unor **spații suplimentare** de memorie pentru **tabloul de contoare** și pentru **zona de sortare**.

3.2.10. Sortarea tablourilor cu elemente de mari dimensiuni. Sortarea indirectă

- În situația în care tablourile de sortat au **elemente de mari dimensiuni**, regia mutării acestor elemente în procesul sortării este mare.
- De aceea este mult mai convenabil ca:
 - (1) **Algoritmul de sortare** să opereze **indirect** asupra tabloului original prin intermediul unui **tablou de indici**.
 - (2) **Tabloul original** să fie **sortat ulterior** într-o singură trecere.

- **Ideea metodei de sortare a tablourilor cu elemente de mari dimensiuni:**
 - Se consideră un tablou $a[0..n-1]$ cu elemente de mari dimensiuni.
 - Se asociază lui a un **tablou de indici** (indicatori) $p[0..n-1]$.
 - Inițial tabloul de indici se completează cu $p[i]=i$ pentru $i=0, n-1$.
 - Algoritmul utilizat în sortare se modifică astfel încât să se acceseze elementele tabloului a prin construcția sintactică $a[p[i]]$ în loc de $a[i]$.
 - Accesul la $a[i]$ prin $p[i]$ se va realiza numai pentru **comparații**, mutările impuse de procesul de sortare efectuându-se doar în tabloul $p[i]$.
 - Cu alte cuvinte algoritmul **va sorta tabloul de indici** astfel încât $p[0]$ va conține indicele celui mai mic element al tabloului a , $p[1]$ indicele elementului următor, etc.
- În acest mod se **evită regia mutării** unor elemente de mari dimensiuni.
 - Se realizează de fapt o **sortare indirectă** a tabloului a .
- Principal o astfel de sortare este prezentată în figura 3.2.10.

Tabloul a **înainte** de sortare:

0	1	2	3	4	5	6	7	8	9
32	22	0	1	5	16	99	4	3	50

Tabloul de indici p **înainte** de sortare:

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Tabloul a **după** sortare:

0	1	2	3	4	5	6	7	8	9
32	22	0	1	5	16	99	4	3	50

Tabloul de indici p **după** sortare:

0	1	2	3	4	5	6	7	8	9
2	3	8	7	4	5	1	0	9	6

Fig. 3.2.10. Exemplu de sortare indirectă

- Această **idee** poate fi aplicată practic **oricărui** algoritm de sortare.
- Pentru exemplificare în secvența [3.2.10.a] se prezintă un algoritm care realizează **sortarea indirectă** bazată pe **metoda inserției** a unui tablou a .

- Poziția 0 a tabloului a[0] este utilizată drept fanion.

```
/*Sortare indirectă bazată pe metoda inserției a unui tablou  
a de mari dimensiuni - Varianta C*/
```

```
/*[3.2.10.a]*/
```

```
tip_element a[n+1];  /*tablou de sortat*/  
tip_indice p[n+1];   /*tablou de indici*/
```

```
void insertie_indirecta()
```

```
{  
    tip_indice i,j,v;  /*indici de lucru*/  
  
    /*inițializarea tabloului de indici*/  
    for(i=1;i<=n;i++) p[i]=i;  
    /*sortare prin inserție a tabloului de indici p*/  
    for(i=2;i<=n;i++)  
    {  
        v=p[i]; a[0]=a[i];  /*a[0] este poziția fanion*/  
        j=i-1;  
        while(a[p[j]].cheie>a[v].cheie)  
        {  
            p[j+1]=p[j];  
            j=j-1;  
        } /*while*/  
        p[j+1]=v;  
    } /*for*/  
} /*insertie_indirecta*/
```

- După cum se observă, cu excepția atribuirii fanionului, accesele la tabloul a se realizează **numai** pentru comparații.
- În multe aplicații este suficientă numai obținerea tabloului p nemaifiind necesară și permutarea elementelor tabloului.
 - Spre exemplu, în procesul tipăririi, elementele pot fi listate în ordine, referirea la ele realizându-se simplu, în mod indirect prin tabloul de indici.
- Dacă este absolut necesară mutarea, cel mai simplu acest lucru se poate realiza într-un alt tablou b într-o singură trecere.
- Dacă acest lucru nu se acceptă, se poate utiliza procedura de reșezare **"in situ"** din secvența [3.2.10.b].

```
/*Procedură de reșezare "in situ" a unui tablou de mari  
dimensiuni - varianta C*/
```

```
/*[3.2.10.b]*/
```

```
void reasezare_in_situ()
```

```
{  
    tip_indice i,j,k;  
    tip_element t_  
  
    for(i=1;i<=n;i++)  
        if(p[i]!=i)  
        {  
            t_=a[i]; k=i;
```

```

do {
    j=k; a[j]=a[p[j]];
    k=p[j]; p[j]=j;
} while (!(k==i));
a[j]=t_;
}
} /*reășezare_in_situ*/

```

- În cazul unor aplicații particulare, viabilitatea acestei tehnici depinde de lungimea relativă a cheilor și articolelor.
 - Metoda descrisă **nu** se justifică pentru **articole de mici dimensiuni** deoarece necesită o zonă de memorie suplimentară pentru tabloul p și timp suplimentar pentru comparațiile indirecte.
 - Pentru **articole de mari dimensiuni** se indică de regulă sortarea indirectă, fără a se mai realiza permutarea efectivă a elementelor.
 - Pentru **articolele de foarte mari dimensiuni** metoda se indică a se utiliza integral, inclusiv permutarea ulterioară a elementelor [Se88].

3.2.11. Concluzii referitoare la sortarea structurilor tablou

- În încheiere se impun câteva aprecieri referitoare la tehnicile de sortare a tablourilor.
- Conform celor discutate până în prezent se notează cu n numărul de elemente al tabloului de sortat și cu C și M numărul de comparații respectiv de mișcări necesare în procesul de sortare.
- În figura 3.2.11 apar relațiile analitice ale valorilor minime, medii și maxime ale indicatorilor C și M pentru trei dintre **metodele directe de sortare**, relații valabile pentru cele $n!$ permutări ale celor n elemente.
- Pentru metodele avansate de sortare, formule sunt mult mai complicate.
 - Este însă esențial de reținut că efortul de calcul în vederea sortării este $c_1 \cdot n^{1.2}$ în cazul tehnicii **shellsort** și $c_1 \cdot n \cdot \log(n)$ în cazul tehnicilor **heapsort** și **quicksort**.
- Aceste formule care aproximează în mod grosier performanțele funcție de numărul de elemente n , permit clasificarea tehnicilor de sortare a tablourilor în:
 - (1) **Tehnici (metode) primitive sau directe** la care efortul de sortare este proporțional cu n^2 .
 - (2) **Tehnici avansate sau logaritmice** la care efortul este proporțional cu $n \cdot \log(n)$.

Metodă		Min	Med	Max
Insertie	C	$n-1$	$\frac{n^2+n-2}{4}$	$\frac{(n-1) \cdot n}{2}$

	M	$3 \cdot (n-1)$	$\frac{n^2 + 11 \cdot n - 12}{4}$	$\frac{n^2 + 5 \cdot n - 6}{2}$
Selecție	C	$\frac{n^2 - 3 \cdot n + 2}{2}$	$\frac{n^2 - 3 \cdot n + 2}{2}$	$\frac{n^2 - 3 \cdot n + 2}{2}$
	M	$3 \cdot (n-1)$	$n \cdot (\ln n + 0,57)$	$\frac{n^2}{4} + 3 \cdot (n-1)$
Interschimb (bubblesort)	C	$\frac{n^2 - 3 \cdot n + 2}{2}$	$\frac{n^2 - 3 \cdot n + 2}{2}$	$\frac{n^2 - 3 \cdot n + 2}{2}$
	M	0	$\frac{3 \cdot (n^2 - 3 \cdot n + 2)}{4}$	$\frac{3 \cdot (n^2 - 3 \cdot n + 2)}{2}$

Fig. 3.2.11. Relații referitoare la performanțele metodelor de sortare directe

- Analizele și măsurătorile efectuate asupra metodelor de sortare prezentate în acest capitol, au permis evidențierea următoarelor **concluzii** [Wi76]:
 - Beneficiile pe care le aduce **inserția binară** față de **inserția simplă** sunt nesemnificative și chiar negative în cazul secvențelor deja sortate.
 - **Tehnica bubblesort** este **cea mai puțin performantă** tehnică de sortare. Chiar versiunea ei îmbunătățită **shakersort** este inferioară inserției sau selecției directe (mai puțin în cazul tablourilor gata sortate).
 - Tehnica **quicksort** este superioară celei **heapsort** cu un factor de 2 la 3. Ea sortează un tablou ordonat invers cu o viteză practic egală cu cea corespunzătoare unuia gata ordonat.
- Se constată că **mărimea dimensiunii unui element**, respectiv a informației utile conținute în element, în raport cu câmpul **cheie**, nu influențează semnificativ performanțele relative ale tehnicilor prezentate.
- Cu toate acestea, pentru această situație se evidențiază următoarele concluzii suplimentare:
 - Performanțele **selecției directe** se îmbunătățesc, situând această tehnică pe primul loc în rândul metodelor directe.
 - **Tehnica bubblesort** este și în acest caz cea mai puțin performantă, chiar pierde teren, iar varianta ei **shakersort** obține performanțe ceva mai bune în cazul tablourilor ordonate invers.
 - **Tehnica quicksort** își întărește pozițiile ca fiind cea mai rapidă metodă și apare de departe ca cea mai bună metodă de sortare a tablourilor.
- Se face precizarea că în aceste aprecieri au fost luate în considerare numai tehnicile generale, care nu necesită **nici un fel de informații suplimentare** referitoare la cheile de sortat.
- În condițiile în care se dispune de astfel de informații, respectiv cheile de sortat îndeplinesc anumite condiții apriorice și/sau se utilizează zone de memorie

suplimentare renunțându-se la restricția **in situ**, performanțele sortării pot fi îmbunătățite.

- În aceasta categorie pot fi incluse **tehnica binsort** care limitează domeniul cheilor și varianta sa extinsă **sortarea bazată pe determinarea distribuției cheilor**, ambele tehnici apropiindu-se de performanța liniară $O(n)$.
- În aceeași categorie pot fi incluse și tehnicile de **sortare radix** cu variantele prin **interschimbare** și **directă** ambele concurând puternic tehnica quicksort.
 - Utilizând tablouri suplimentare de dimensiuni corespunzătoare pentru memorarea distribuției cheilor, aceste metode se pot apropia ca și performanțe de **sortarea liniară** $O(n)$.
- Pentru tablourile ale căror elemente sunt de mari dimensiuni, există tehnici de **sortare indirectă**, aplicabile practic oricărei metode de sortare, care îmbunătățesc substanțial performanțele.
- Ca și o remarcă finală se reamintește faptul că această secțiune a abordat doar **sortarea structurilor de date tablou**, deși unele dintre tehnicile prezentate pot fi aplicate și altor tipuri de structuri.

3.3. Sortarea secvențelor. Sortarea externă

- Metodele de sortare prezentate în paragraful anterior **nu** pot fi aplicate unor date care **nu** încap în **memoria centrală** a sistemului, dar care pot fi spre exemplu memorate pe **dispozitive periferice secvențiale** cum ar fi **benzile magnetice** sau **discurile magnetice** în alocare secvențială.
- În acest caz datele pot fi modelate cu ajutorul unei **structuri secvență** având drept **caracteristică esențială** faptul că **accesul la componente** se realizează în manieră **strict secvențială**.
 - Aceasta este o **restricție** foarte severă comparativ cu **accesul direct** oferit de structura tablou, motiv pentru care tehnicile de sortare sunt de cu totul altă natură.
- Una dintre cele mai importante **tehnici de sortare a secvențelor** este **sortarea prin interclasare** ("merging").

3.3.1. Sortarea prin interclasare

- **Interclasarea** presupune **combinarea** a două sau mai multe secvențe ordonate într-o singură secvență ordonată, prin selecții repetate ale componentelor curent accesibile.
 - Interclasarea este o operație simplă, care este utilizată ca **auxiliar** în procesul mult mai complex al **sortării secvențiale**.
- O **metodă de sortare** bazată pe **interclasare** a unei secvențe a este următoarea:
 - (1) Se **împarte** secvența de sortat a în două jumătăți b și c .

- (2) Se **interclasează** b cu c, combinând câte un element din fiecare, în **perechi ordonate** obținându-se o nouă secvență a.
- (3) Se **repetă** cu secvența interclasată a, pașii (1) și (2) de această dată combinând perechile ordonate în **qvadruple ordonate**.
- (4) Se **repetă** pașii inițiali, interclasând qvadruplele în **8-uple**, ș.a.m.d, de fiecare dată **dublând** lungimea subsecvențelor de interclasare până la sortarea întregii secvențe.

- Spre exemplu fie secvența:

a: 34 65 12 22 83 18 04 67

- Execuția pasului 1 conduce la două jumătăți de secvență:

b: 34 65 12 22

c: 83 18 04 67

- Interclasarea componentelor unice în **perechi ordonate** conduce la secvența:

a: 34 83 | 18 65 | 04 12 | 22 67

- Înjumătățind din nou:

b: 34 83 | 18 65

c: 04 12 | 22 67

- Și interclasând perechile în **qvadruple** se obține:

a: 04 12 34 83 | 18 22 65 67

- Cea de-a treia înjumătățire:

b: 04 12 34 83

c: 18 22 65 67

- Interclasând cele două qvadruple într-un **8-uplu** se ajunge la secvența gata sortată:

a: 04 12 18 22 34 65 67 83

- Se observă faptul extrem de semnificativ că, **accesul** la componentele succesive ale secvențelor se realizează în manieră **strict secvențială**.
 - Fiecare operație care tratează întregul set de date se numește **fază**.
 - Procesul prin repetarea căruia se realizează sortarea se numește **trecere**.
- Procesul de sortare anterior descris constă din 3 treceri fiecare cuprinzând 2 faze: o fază de **înjumătățire** și una de **interclasare**.

- Pentru a realiza sortarea sunt necesare trei secvențe motiv pentru care sortarea se numește **interclasare cu trei secvențe**.
- Aceasta este de fapt o **interclasare neechilibrată cu 3 secvențe**.

3.3.1.1. Interclasarea neechilibrată cu trei secvențe

- **Interclasarea neechilibrată cu trei secvențe** reprezintă implementarea procedurii de sortare precizat anterior.
- **Schema de principiu** a acestui procedeu apare în figura 3.3.1.1.

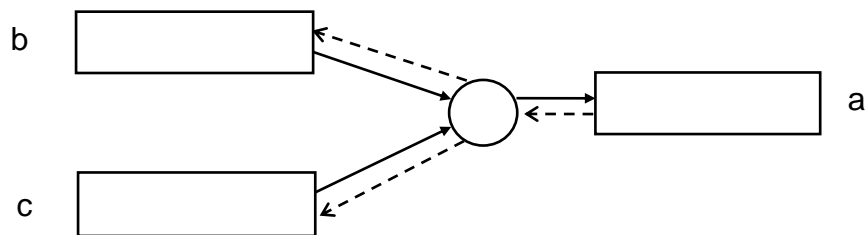


Fig. 3.3.1.1. Interclasare neechilibrată cu trei secvențe

- Într-o primă etapă în secvențele [3.3.1.1.a, b] se prezintă structurile de date și schița de principiu a algoritmului.

```
/*Interclasarea neechilibrată cu trei secvențe - Structuri de date*/
```

```
typedef struct tip_element /*[3.3.1.1.a]*/
{
    tip_cheie cheie;
    /*alte câmpuri*/
} tip_element
```

```
tip_secventa = FILE OF tip_element;
```

```
tip_secventa a,b,c;
```

```
/*Interclasare neechilibrată cu 3 secvențe - Pas rafinare 0*/
```

```
PROCEDURE Interclasare3Secvente; /*[3.3.1.1.b]*/
```

```
    tip_secventa a;    /*secvența de sortat*/
    tip_secventa b,c;  /*secvențe de lucru*/
```

```
    int p; /*p este dimensiunea n-uplu*/
    int k; /*k este numărul n-upluri interclasate*/
```

```

p=1;      /*inițializare dimensiune n-uplu*/
repetă
    *înjumătățire; /*distribuie n-uplele de pe a pe b și c*/
    *interclasare; /*interclasează n-uplele de pe b și c
                    pe a*/
    p=2*p      /*dublează dimensiunea n-uplului*/
până când k=1;
□ /*k este contorul de n-uple interclasate*/

```

- După cum se observă, fiecare trecere care constă dintr-o reluare a buclei **repeat** conține două faze:
 - (1) O **fază de înjumătățire** adică de **distribuție** a n -uplelor secvenței a pe cele două secvențe b și c.
 - (2) O **fază de interclasare** în care n -uplele de pe secvențele b și c se interclasează în n -uple de dimensiune dublă pe secvența a.
 - Variabila p inițializată pe 1, precizează **dimensiunea** n -uplelor curente, dimensiune care după fiecare trecere se dublează.
 - În consecință **numărul total de treceri** va fi $\lceil \log_2 n \rceil$.
 - Variabila k contorizează **numărul** de n -uple create în procesul de interclasare.
 - Procesul de sortare se încheie când în final rămâne un singur n -uplu ($k=1$).
- În continuare algoritmul de sortare se dezvoltă utilizând tehnologia de dezvoltare "stepwise refinement" [Wi76].
 - Procesul de dezvoltare constă de fapt în **rafinarea succesivă în manieră iterativă și incrementală** a celor două faze.
- În secvența [3.3.1.1.c] apare primul pas de rafinare al fazei de **Injumătățire**, iar în secvența următoare rafinarea enunțului "scrie un n -uplu de dimensiune p în secvența d".

/*Procedura Înjumătățire - Pas rafinare 1*/

```

PROCEDURE Injumatatire(int p);      /*[3.3.1.1.c]*/
/*distribuie n-uplele de pe a pe b și c*/
/*p - dimensiune n-uplu*/

```

```

    tip_secventa a;      /*secvența de sortat*/
    tip_secventa b,c;    /*secvențe de lucru*/

```

```

    RESET(a); REWRITE(b); REWRITE(c);

```

```

    cât timp (NOT Eof(a))
        *scrie un n-uplu pe b;
        *scrie un n-uplu pe c;

```

```

    □

```

```

PROCEDURE ScrieNuplu(TipSecventa d);      /*[3.3.1.1.d]*/
/*scrie un n-uplu de dimensiune p pe secvența destinație d

```

citirea elementelor se face de pe secvența a*/

```
tip_secventa a; /*secvența sursă*/
tip_secventa d; /*secvența destinație*/
tip_element x; /*element transferat*/
int i;          /*contor elemente pentru n-uplul curent*/
int p;          /*dimensiune n-uplu curent*/

i=0; /*inițializare contor elemente n-uplu curent*/
cât timp ((i<p) AND NOT Eof(a)) /*p este lungimea n-uplu*/
|   *citeste(a,x);
|   *scrie(d,x);
|   i=i+1;
| □
```

- Variabila i reprezintă **contorul de elemente** al n-uplului care poate lua valori între 0 și p-1.
 - Scrierea se **termină** la atingerea numărului p de elemente sau la terminarea secvenței sursă.
- Rafinarea **fazei de interclasare** apare în secvența [3.3.1.1.e].

/*Procedura Intercalsare - Pas rafinare 1*/

```
PROCEDURE Interclasare(int p, int k)      /*[3.3.1.1.e]*/
/*interclasează n-uplurile de pe b și c pe a */
/*p - dimensiune n-uplu, k - contor n-uple interclasate*/

tip_secventa a; /*secvența destinație*/
tip_secventa b,c; /*secvențe sursă*/
tip_element x,y; /*elemente de interclasat*/
int k;          /* contorul de n-uple interclasate pe a*/

Rewrite(a); Reset(b); Reset(c);
k= 0; /*inițializare contor de n-uple interclasate pe a*/
*inițializare interclasare;
*citeste în x primul element din b; /*tehnica lookahead*/
*citeste în y primul element din c; /*tehnica lookahead*/
repetă
|   *interclaseaza câte un n-uplu de pe b și c pe a;
|   *incrementează pe k cu 1;
până când (EndPrelucr_b AND EndPrelucr_c)
| □
Close(a); Close(b); Close(c);
```

-
- Variabila de intrare p reprezintă **dimensiunea** n-uplelor care se interclasează, iar k este **contorul** de n-uple.
 - Practic interclasarea propriu-zisă (bucla **repetă**) se încheie la terminarea prelucrării secvențelor b și c.
 - Datorită particularităților de implementare a fișierelor sunt necesare câteva **precizări**:
 - (1) Variabila Eof(f) se poziționează pe **true** la citirea ultimului element al fișierului f.
 - (2) Citirea dintr-un fișier cu Eof poziționat pe **true** conduce la **eroare**.

- (3) Din punctul de vedere al algoritmului de interclasare, terminarea prelucrării unui fișier **nu** coincide cu poziționarea lui `Eof` pe **true**, deoarece mai trebuie prelucrat ultimul element citit.
- Pentru rezolvarea acestor constrângeri se utilizează **tehnica scrutării** ("lookahead").
 - **Tehnica scrutării** constă în introducerea unei **întârzieri** între momentul citirii și momentul prelucrării unui element.
 - Astfel în fiecare moment se prelucrează elementul citit în **pasul anterior** și se citește un **nou element**.
 - În acest scop pentru fiecare fișier implicat în prelucrare se utilizează:
 - (1) O variabilă specială de tip `tip_element` care **memorează** elementul curent.
 - (2) O variabilă booleană `EndPrelucr` a cărei valoare **true** semnifică **terminarea prelucrării** ultimului element al fișierului.
- Rafinarea enunțului "interclasează câte un n -uplu de pe b și c pe a și incrementează pe k " apare în secvența [3.3.1.1.f] care aplică tehnica anterior precizată.
- Variabilele specifice asociate secvențelor b și c sunt x și y respectiv `EndPrelucr_b` și `EndPrelucr_c` [3.3.1.1.g].

/*interclaseaza câte un n-uplu de pe b si c pe a și incrementează pe k*/

```

                                                    /*[3.3.1.1.f]*/
tip_secventa a;    /*secvența destinație*/
tip_secventa b,c;  /*secvențe sursă*/
tip_element x;     /*x - elementul curent al secvenței b*/
tip_element y;     /*y - elementul curent al secvenței c*/
int i;             /*contor n-uplu curent pe b*/
int j;             /*contor n-uplu curent pe c*/
int k;             /*contor de n-uple interclasate pe a*/
int p;             /*dimensiune n-uplu de interclasat*/

i=0;    /*inițializare contor n-uplu curent pe b*/
j=0;    /*inițializare contor n-uplu curent pe c*/
cât timp ((i<p)AND(j<p) AND NOT EndPrelucr_b AND
            NOT EndPrelucr_c)
  dacă (x.cheie<y.cheie)
    *scrie(a,x); i=i+1;
    dacă Eof(b) EndPrelucr_b=true; /*x este ultimul
      altfel                               element din b*/
      *citeste(b,x); /*citește elementul următor din
    □                               b*/
    altfel
      *scrie(a,y); j=j+1;
      dacă Eof(c) EndPrelucr_c=true; /*y este ultimul elem
        altfel                               din c*/
        *citeste(c,y); /*citește următorul element din
      □ {altfel}                               c*/
    □ {dacă}
  □ {cât timp}
  *copiază restul n-uplului de pe b pe a (dacă există);

```

```

    *copiază restul n-uplului de pe c pe a (dacă există);
    k=k+1;    /*incrementează contorul de n-uple interclasate
              pe secvența a*/

```

- O variantă de implementare integrală a procedurii de **sortare neechilibrată cu 3 benzi** incluzând structurile de date aferente și funcțiile implicate apare în secvența [3.3.1.1.g] iar programul integral în [3.3.1.1.h].
-

//Interclasare neechilibrată cu 3 secvențe

```

#include <stdio.h>                                /*[3.3.1.1.g]*/
#include <stdlib.h>
#include <string.h>

typedef int boolean;
typedef int tip_element;

#define true (1)
#define false (0)
#define LUNGIME_MAXIMA_NUME_FISIER 256

// definirea tipului fisier
typedef struct
{
    char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
    FILE* file;
} tip_fisier;

// definirea tipului secvență
typedef struct tip_secventa
{
    tip_fisier fisier;
    tip_element elemCurent;
    boolean termPrelucr;
} tip_secventa;

tip_secventa a,b,c; // secvențele de lucru
tip_element x; // x este elementul curent al secvenței b
tip_element y; // y este elementul curent al secvenței c
int p; // p este dimensiunea n-uplu
int k; // k este numărul n-upluri interclasate

// definirea funcțiilor referitoare la interclasarea neechilibrată

void ScribeNuplu(tip_secventa d)
// scrie un n-uplu de dimensiune p pe secvența d, citirea elementelor
se face de pe secvența a

{
    int i;
    i=0; // contor elemente n-uplu curent
    while((i<p) && !Eof(&a.fisier)) // p este lungimea n-uplu
    {
        Read(&a.fisier,&x);
        Write(&d.fisier,x);
        i=i+1;
    }
}

```

```

    }
}

void Injumatatire(int p)
// distribuie n-uplele de pe a pe b și c
// p - dimensiune n-uplu

{
    Reset(&a);
    Rewrite(&b);
    Rewrite(&c);

    while(!Eof(&a.fisier))
    {
        ScrieNuplu(b);
        ScrieNuplu(c);
    }
}

void Interclasare(int p,int* k)
//interclasează pe rând câte un n-uplu de pe b și c pe a și
//incrementează contorul de n-uple k, pâna la epuizarea acestora
//p - dimensiune n-uplu, k - contor n-uple

{
    int i,j;
    boolean EndPrelucr_b;
    boolean EndPrelucr_c;

    Reset(&b);
    Reset(&c);
    Rewrite(&a);

    *k=0; // k este contorul de n-uple interclasate pe a

    EndPrelucr_b=Eof(&b.fisier);
    EndPrelucr_c=Eof(&c.fisier);

    if(!EndPrelucr_b) Read(&b.fisier,&x); // lookahead
    if(!EndPrelucr_c) Read(&c.fisier,&y); // lookahead

    do
    {
        i=0; // contor n-uplu curent pe b
        j=0; // contor n-uplu curent pe c

        // interclasarea unui n-uplu
        while(i<p && j<p && !EndPrelucr_b && !EndPrelucr_c)
        {
            if(x<y)
            {
                Write(&a.fisier,x);
                i=i+1;
                if(Eof(&b.fisier))
                    EndPrelucr_b=true;
                else
                    Read(&b.fisier,&x); // citește următorul element din b
            }
            else
            {

```

```

        Write(&a.fisier,y);
        j=j+1;
        if(Eof(&c.fisier))
            EndPrelucr_c=true;
        else
            Read(&c.fisier,&y); // citește următorul element din c
    }
} // while

// copiază restului n-uplului de pe b pe a
while (i<p && !EndPrelucr_b)
{
    Write(&a.fisier, x);
    i=i+1;
    if(Eof(&b.fisier))
        EndPrelucr_b=true;
    else
        Read(&b.fisier,&x); // citește următorul element din b
}
// copiază restului n-uplului de pe c pe a
while(j<p && !EndPrelucr_c)
{
    Write(&a.fisier,y);
    j=j+1;
    if(Eof(&c.fisier))
        EndPrelucr_c=true;
    else
        Read(&c.fisier,&y); // citește următorul element din c
}

*k=*k+1; // incrementeaza contorul de n-uple interclasate pe
secvența a

}while(!EndPrelucr_b || !EndPrelucr_c);
}

// operatorul interclasare cu 3 secvențe
void Interclasare3Secvente()
{
    p=1;
    do
    {
        Inumatatire(p); // distribuie n-uplele de pe a pe b și c
        Interclasare(p,&k); // interclasează n-uplele de pe b și c pe a
        p=p*2; // dublează dimensiunea n-uplului

    }while(k != 1); // k este contorul de n-uple interclasate

    Close(a.fisier);
    Close(b.fisier);
    Close(c.fisier);
}

```

- O variantă de implementare integrală a procedurii de **sortare neechilibrată cu 3 benzi** apare în secvența [3.3.1.1.h]. Această variantă include pe lângă structurile de date aferente și funcțiile implicate în sortare, operatorii referitori la procesarea fișierelor, funcții de generare a numerelor aleatoare, funcții de inițializare și generare de secvențe utilizate în vederea creării inițiale a secvențelor de sortat precum și programul principal.

//Program Interclasare neechilibrată cu 3 secvențe -
variantă integrală

```
#include <stdio.h>                                /*[3.3.1.1.h]*/
#include <stdlib.h>
#include <string.h>

typedef int boolean;
typedef int tip_element;

#define true (1)
#define false (0)
#define LUNGIME_MAXIMA_NUME_FISIER 256

// definirea tipului fisier
typedef struct
{
    char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
    FILE* file;
} tip_fisier;

// definirea tipului secvență
typedef struct tip_secventa
{
    tip_fisier fisier;
    tip_element elemCurent;
    boolean termPrelucr;
} tip_secventa;

tip_secventa a,b,c; // secvențele de lucru
tip_element x;      // x este elementul curent al secvenței b
tip_element y;      // y este elementul curent al secvenței c
int p;              // p este dimensiunea n-uplu
int k;              // k este numărul n-upluri interclasate

// definirea operatorilor referitori la prelucrarea fișierelor
void Write(tip_fisier* fisier, tip_element buf)
{
    fprintf(fisier->file, "%d ", buf);
}

boolean Eof(tip_fisier* fisier)
{
    int current_position = ftell(fisier->file);
    fseek(fisier->file, 0, SEEK_END);
    int size = ftell(fisier->file);
    fseek(fisier->file, current_position, SEEK_SET);
    return feof(fisier->file) || size == current_position;
}

void Reset(tip_secventa* secventa)
{
    fseek(secventa->fisier.file, 0, SEEK_SET);
    secventa->termPrelucr = false;
}

void Read(tip_fisier* fisier, tip_element* elem_curent)
{

```

```

        fscanf(fisier->file, "%d ", elem_curent);
    }

    void Rewrite(tip_secventa* secventa)
    {
        fclose(secventa->fisier.file);
        secventa->fisier.file = fopen(secventa->fisier.num_fisier,
"w+");
    }

    void Close(tip_fisier fisier)
    {
        fclose(fisier.file);
    }

    // definirea funcțiilor referitoare la interclasarea neechilibrată

    void ScrieNuplu(tip_secventa d)
    // scrie un n-uplu de dimensiune p pe secvența d, citirea elementelor
    se face de pe secvența a

    {
        int i;
        i=0; // contor elemente n-uplu curent
        while((i<p) && !Eof(&a.fisier)) // p este lungimea n-uplu
        {
            Read(&a.fisier,&x);
            Write(&d.fisier,x);
            i=i+1;
        }
    }

    void Injumatatire(int p)
    // distribuie n-uplele de pe a pe b și c
    // p - dimensiune n-uplu

    {
        Reset(&a);
        Rewrite(&b);
        Rewrite(&c);

        while(!Eof(&a.fisier))
        {
            ScrieNuplu(b);
            ScrieNuplu(c);
        }
    }

    void Interclasare(int p,int* k)
    //interclasează pe rând câte un n-uplu de pe b și c pe a și
    incrementează contorul de n-uple k, până la epuizarea acestora
    //p - dimensiune n-uplu, k - contor n-uple

    {
        int i,j;
        boolean EndPrelucr_b;
        boolean EndPrelucr_c;

        Reset(&b);
        Reset(&c);
    }

```

```

Rewrite(&a);

*k=0; // k este contorul de n-uple interclasate pe a

EndPrelucr_b=Eof(&b.fisier);
EndPrelucr_c=Eof(&c.fisier);

if(!EndPrelucr_b) Read(&b.fisier,&x); // lookahead
if(!EndPrelucr_c) Read(&c.fisier,&y); // lookahead

do
{
    i=0; // contor n-uplu curent pe b
    j=0; // contor n-uplu curent pe c

    // interclasarea unui n-uplu
    while(i<p && j<p && !EndPrelucr_b && !EndPrelucr_c)
    {
        if(x<y)
        {
            Write(&a.fisier,x);
            i=i+1;
            if(Eof(&b.fisier))
                EndPrelucr_b=true;
            else
                Read(&b.fisier,&x); // citește următorul element din b
        }
        else
        {
            Write(&a.fisier,y);
            j=j+1;
            if(Eof(&c.fisier))
                EndPrelucr_c=true;
            else
                Read(&c.fisier,&y); // citește următorul element din c
        }
    } // while

    // copiază restului n-uplului de pe b pe a
    while (i<p && !EndPrelucr_b)
    {
        Write(&a.fisier, x);
        i=i+1;
        if(Eof(&b.fisier))
            EndPrelucr_b=true;
        else
            Read(&b.fisier,&x); // citește următorul element din b
    }
    // copiază restului n-uplului de pe c pe a
    while(j<p && !EndPrelucr_c)
    {
        Write(&a.fisier,y);
        j=j+1;
        if(Eof(&c.fisier))
            EndPrelucr_c=true;
        else
            Read(&c.fisier,&y); // citește următorul element din c
    }

    *k=*k+1; // incrementează contorul de n-uple interclasate pe
    secvența a
}

```

```

    }while(!EndPrelucr_b || !EndPrelucr_c);

}

// operatorul interclasare cu 3 secvențe
void Interclasare3Secvente()
{
    p=1;
    do
    {
        Injumatatire(p);    // distribuie n-uplele de pe a pe b și c
        Interclasare(p,&k); // interclasează n-uplele de pe b și c pe a
        p=p*2;              // dublează dimensiunea n-uplului

    }while(k != 1);        // k este contorul de n-uple interclasate

    Close(a.fisier);
    Close(b.fisier);
    Close(c.fisier);
}

// funcții generare numere aleatoare
void generare_numere_aleatoare(const char* nume_fisier, int dimensiune)
// generează numere aleatoare autonom
{
    FILE* destination = fopen(nume_fisier, "w");
    if (!destination)
    {
        printf("\nEroare la deschidere\n");
        exit(1);
    }

    int aleat = 7789;
    do
    {
        aleat = (131071 * aleat) % 2147483647;
        int temp = aleat / 214784;

        fprintf(destination, "%d ", temp);
        dimensiune--;

    }while (dimensiune > 0);

    fclose(destination);
}

void generare_numere_aleatoare_poz(const char* nume_fisier, int
dimensiune)
// generează numere aleatoare cu ajutorul funcției rand()
{
    FILE* destination = fopen(nume_fisier, "w");
    if (!destination)
    {
        printf("\nEroare la deschidere\n");
        exit(1);
    }

    do
    {
        int temp = rand() % 30;

```

```

        fprintf(destination, "%d ", temp);
        dimensiune--;
    }
    while (dimensiune > 0);

    fclose(destination);
}

// functii initializare și generare secvențe
void InitializareUnFisier(tip_secventa *secventa, char* nume_fisier)
{
    secventa->termPrelucr = false;
    strcpy(secventa->fisier.nume_fisier, nume_fisier);
    secventa->fisier.file = fopen(nume_fisier, "w+");
}

void InitializareFisiere()
{
    InitializareUnFisier(&a, "a.txt");
    InitializareUnFisier(&b, "b.txt");
    InitializareUnFisier(&c, "c.txt");

    tip_fisier f;
    f.file = fopen("initial.txt", "r");
    int x;
    while(!Eof(&f))
    {
        Read(&f, &x);
        Write(&a.fisier, x);
    }
    fclose(f.file);
}

// programul principal
int main(void)
{
    generare_numere_aleatoare_poz("initial.txt", 30);
    InitializareFisiere();
    Interclasare3Secvente();
    return 0;
}
-----

```

3.3.1.2. Interclasarea echilibrată cu 4 secvențe (două căi)

- **Faza de înjumătățire** care de fapt **nu** contribuie direct la sortare (în sensul că ea **nu** permută nici un element), consumă jumătate din operațiile de copiere.
- Acest neajuns poate fi remediat prin **combinarea fazei de înjumătățire** cu cea de **interclasare**.
 - Astfel **simultan** cu **interclasarea** se realizează și **redistribuirea** n-uplelor interclasate pe două secvențe care vor constitui sursa trecerii următoare.
- Acest proces se numește **interclasare cu o singură fază** sau **interclasare echilibrată cu 4 secvențe** (2 căi).

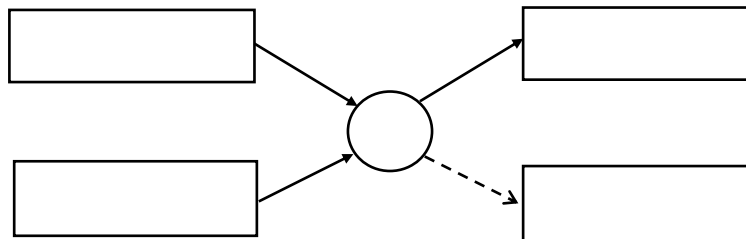


Fig. 3.3.1.2.a. Interclasare echilibrată cu patru secvențe

- Într-o primă etapă, se va **modela** interclasarea echilibrată cu 4 secvențe utilizând drept suport de modelare o **structură tablou**, care va fi parcurs în manieră strict secvențială.
- Într-o etapă ulterioară, interclasarea va fi aplicată unor **structuri secvență**, permițând compararea celor două abordări și în același timp demonstrând puternica dependență a formei **algoritmului** în raport cu **structurile de date** pe care le utilizează.
- Un **tablou** poate **modela** două fișiere, dacă este privit ca o secvență cu două capete.
 - Astfel, în procesul de interclasare a celor două fișiere sursă, elementele se vor lua de la cele două capete ale tabloului.
- Pentru **modelarea sortării echilibrate** se vor utiliza **două astfel de tablouri** numite SURSA respectiv DESTINAȚIE
 - Faza combinată înjumătățire-interclasare apare reprezentată schematic în figura 3.3.1.2.b.

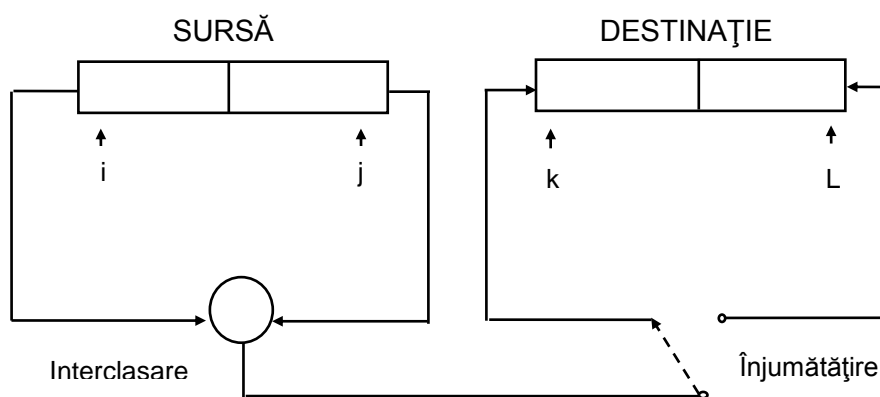


Fig. 3.3.1.2.b. Model pentru interclasarea echilibrată

- Destinația articolelor interclasate este **comutată** după fiecare pereche ordonată la prima trecere, după fiecare quadruplu la a doua trecere, ș.a.m.d, astfel încât cele două secvențe destinație, sunt de fapt cele două capete ale unui singur tablou.
- După fiecare trecere cele două tablouri se **interschimbă** sursa devenind noua destinație și reciproc.

- În continuare lucrurile se pot simplifica **reunind** cele două tablouri conceptuale într-unul singur de lungime dublă:

```
tip_element a[2*n];
```

- În acest tablou, indicii i și j precizează două elemente **sursă**, iar k și L două **destinații**.
 - **Secvența inițială** va fi conținută de prima parte a tabloului $a[0], \dots, a[n-1]$.
- Se introduce o **variabilă booleană** `sus` care precizează sensul mișcării elementelor:
 - De la **sursa** $a[0], \dots, a[n-1]$ spre **destinația** $a[n], \dots, a[2*n-1]$ când `sus` este **adevărat**.
 - De la **sursa** $a[n], \dots, a[2*n-1]$ spre **destinația** $a[0], \dots, a[n-1]$ când `sus` este **fals**.
 - În mod evident valoarea lui `sus` alternează de la o trecere la alta.
- Se mai introduce variabila `p` care precizează **lungimea subsecvențelor** ce urmează să fie interclasate.
 - `p` ia valoarea inițială 1 și se dublează după fiecare trecere.
 - Pentru a simplifica lucrurile se presupune momentan că n este o putere a lui 2.
- În aceste condiții, în secvența [3.3.1.2.a] apare **varianta pseudocod** a procedurii de interclasare iar în secvența [3.3.1.2.b] primul pas de rafinare.

{Procedura Interclasare echilibrată cu 4 secvențe - varianta pseudocod}

```
PROCEDURE Interclasare                                {[3.3.1.2.a]}
  boolean sus; {indicator boolean pentru sensul mișcării
               elementelor}
  int p;      {p este lungimea n-uplului curent}
  int n;      {n este numărul inițial de elemente de sortat}

  sus=true;  p=1; {p este lungimea n-uplului curent}
  repeta
    |  daca sus atunci
    |    stânga este sursa; dreapta este destinație;
    |    altfel
    |      dreapta este sursa; stânga este destinație;
    |  □{daca}
    |    *se interclasează n-uple de lungime p de la
    |      două capete ale sursei, alternativ în cele
    |      două capete ale destinației;
    |    sus=not sus;
    |    p=2*p;      {dublare lungime n-uplului curent}
  până când (p=n) {n este numărul inițial de elemente de
                  sortat}
  □{repeta}
```

{Procedura Interclasare echilibrată cu 4 secvențe - Pasul 1 de rafinare}

PROCEDURE Interclasare

{[3.3.1.2.b]}

```
int i,j; {indici delimitatori pentru sursă}
int k,L; {indici delimitatori pentru destinație}
boolean sus; {indicator boolean pentru sensul mișcării
              elementelor}
int p;      {p este lungimea n-uplului curent}
int n;      {n este numărul inițial de elemente de sortat}

sus= true; p= 1;
repetă {inițializare indici}
    dacă sus
        i=0; j=n-1;      {stanga este sursă}
        k=n; L=2*n-1;    {dreapta este destinație}
        □
        altfel
            k=0; L=n-1;    {stanga este sursă}
            i=n; j=2*n-1;  {dreapta este destinație}
            □ {altfel}
        □{daca}
        *interclasează p-tuplele secvențelor i și j, alternativ
          în secvența k respectiv L;
        sus=NOT sus;
        p=2*p; {dublare lungime n-uplului curent}
    până când (p=n) {n- numărul inițial de elemente de sortat}
    □{repetă}
{Interclasare}
```

- Interclasarea este de fapt o buclă **repetă** pentru p mergând de la 1 la n.
 - La fiecare trecere p se dublează iar sus comută.
- În cadrul unei **tregeri**:
 - Funcție de variabila sus se asignează indicii sursă-destinație.
 - Se interclasează p -tuplele secvențelor **sursă** în p -tuple de dimensiune dublă și se depun în secvența **destinație**.
- În pasul următor al detalierilor succesive se rafinează, enunțul "interclasează p -tuplele secvențelor i și j, alternativ, în secvența k respectiv L".
- Este clar că interclasarea a n elemente este la rândul ei o succesiune de interclasări parțiale ale unor subsecvențe de lungime precizată p, în particular p -tuple, în subsecvențe de lungime 2p.
- După fiecare astfel de **interclasare** parțială, destinația este **comutată** de la un capăt al tabloului destinație la celălalt, asigurând astfel **distribuția egală** a subsecvențelor .

- Dacă destinația elementului interclasat este capătul inferior al tabloului destinație, atunci k este indicele destinație și valoarea sa este **incrementată** cu 1 după fiecare mutare.
- Dacă mutările se execută la capătul superior, atunci L este indicele destinație și valoarea sa va fi **decrementată** cu 1 după fiecare mutare.
- Pentru simplificare, **destinația** va fi precizată **tot timpul** de indicele k , intercomutând valorile lui k și L după interclasarea fiecărui p -tuplu și asignând pentru incrementul permanent h valoarea 1 respectiv -1.
- Rezultă următoarea rafinare [3.3.1.2.c]:

{Rafinare enunț: interclasează p-uplele secvențelor i și j , alternativ în secvența k respectiv L }

```

-----
{[3.3.1.2.c]}
int n; {n este număr inițial de elemente de interclasat }
int m; {m este numărul rămas de elemente de interclasat}
int h; {h este incrementul curent al secvenței destinație}
int q; {q este lungime curentă n-uplu de interclasat de la
      sursa i}
int r; {r este lungime curentă n-uplu de interclasat de la
      sursa j}
int p; {p este lungime curentă n-uple de interclasat}

h=1; {inițializare increment destinație}
m=n; {inițializare număr rămas de elemente de interclasat
      cu numărul inițial de elemente de interclasat}
repetă
  q=p; {q - lungime curentă n-uplu de interclasat de la
        sursa i}
  r=p; {r - lungime curentă n-uplu de interclasat de la
        sursa j}
  m=m-2*p; {m este numărul rămas de elemente de interclasat}
  *interclasează q elemente de la sursa-i cu r elemente de
    la sursa-j; indicele destinație este k cu rația h;
  h=-h; {comutare valoare increment destinație}
  *interschimbă indicii destinație (k și L) pentru
    alternanță;
până când (m=0)
  □ {repetă}
-----

```

- Referitor la secvența [3.3.1.2.c] se precizează următoarele:
 - S-au notat cu r respectiv q **lungimile secvențelor** care se interclasează.
 - De regulă la începutul intercalsării $r=q=p$, ele modificându-se eventul în zona finală dacă n nu este o putere a lui 2.
 - S-a notat cu m **numărul de elemente** care mai sunt de interclasat în cadrul unei treceri.
 - Inițial $m=n$ și el scade după fiecare interclasare a două subsecvențe cu $2 \cdot p$.

- Procesul de interclasare pentru o trecere se încheie când $m=0$.
- Pasul următor de detaliere rezolvă **interclasarea**.
 - Se precizează faptul că în situația în care procesul de interclasare parțială **nu** epuizează o subsecvență, restul rămas neprelucrat trebuie adăugat secvenței destinație printr-o operație de copiere [3.3.1.2.d].

{Rafinare enunț: interclasează q elemente de la sursa i cu r elemente de la sursa j ; indicele destinație este k }

{ [3.3.1.2.d] }

```
tip_element a[2n]; {tablou secvențe}
int q;           {număr elemente secvența i}
int r;           {număr elemente secvența j}
int i,j;         {indici surse}
int k;           {indice destinație}
```

```
cât timp ((q<>0) AND (r<>0))
  {selectează cel mai mic element dintre a[i] și a[j]}
  dacă (a[i].cheie<a[j].cheie)
    *mută un element de la i la k, avansează i și k;
    q=q-1;
    □
  altfel
    *mută un element de la j la k, avansează j și k
    r=r-1;
    □{altfel}
  □{dacă}
□{cât timp}
*copiază restul secvenței i;
*copiază restul secvenței j;
```

- Înainte de a se trece la redactarea propriu-zisă a programului, se va elimina **restricția** ca n să fie o putere a lui 2.

- În acest caz, se continuă interclasarea p -tuplurilor până când secvențele sursă care rămân au lungimea mai mică decât p .
- Se observă ușor că singura parte a algoritmului care este influențată de această situație este aceea în care se determină valorile lui q și r , care sunt lungimile secvențelor de interclasat din [3.3.1.2.c].
- În consecință cele trei instrucții:

$$q=p; \quad r=p; \quad m=m-2*p;$$

- Vor fi înlocuite cu:

dacă $m \geq p$ **atunci** $q=p$; **altfel** { $q=m$; $m=m-q$; }

dacă $m \geq p$ **atunci** $r=p$ **altfel** { $r=m$; $m=m-r$; }

unde m este numărul de elemente care au mai rămas de interclasat.

- În plus, condiția care controlează **terminarea algoritmului** $p=n$, trebuie modificată în $p \geq n$.
- Varianta finală de rafinare a procedurii **Interclasare** împreună cu structurile de date implicate apare în [3.3.1.2.e].

 // Implementare interclasare echilibrată cu 4 secvențe implementate cu ajutorul structurii tablou liniar

```
typedef int boolean;
```

```
#define true (1)
```

```
#define false (0)
```

```
int a[2*n+1]; // tabloul suport pentru cele 4 secvențe
```

```
// interclasare echilibrată cu 4 secvențe, structura suport este tabloul a
```

```
void Interclasare()
```

```
{
    int i,j;      // indicii i,j sunt delimitatori pentru sursă
    int k,L;      // indicii k,L sunt delimitatori pentru destinație
    int t;        // t este o variabilă auxiliară
    int h;        // h este incrementul curent al secvenței destinație
    int m;        // m este numărul rămas de elemente de interclasat
    int n;        // n este numărul inițial de elemente de sortat}
    int p;        // p este lungime curentă a n-uplelor de interclasat
    int q;        // q este lungime curentă n-uplu de interclasat de la sursa i
    int r;        // r este lungime curentă n-uplu de interclasat de la sursa j
    boolean sus;  // indicator boolean pentru sensul mișcării elementelor
```

```
    sus=true;
```

```
    p=1; // p este lungimea n-uplului curent
```

```
    do
```

```
    {
```

```
        // inițializare indici
```

```
        h=1; // h este incrementul curent al secvenței destinație
```

```
        m=n; // m = numărul de elemente de interclasat, iar n = numărul inițial
              de elemente de sortat
```

```
        if(sus)
```

```
        {
```

```
            i=1; j=n;          // sursa
```

```
            k=n+1; L=2*n;      //destinație
```

```
        }
```

```
        else {
```

```
            k=1; L=n;          // sursa
```

```
            i=n+1; j=2*n;      //destinație
```

```
        }
```

```
    do
```

```
    {
```

```
        // interclasează de la i și j la k
```

```
        // q=lungimea de la i; r=lungimea de la j
```

```
        if (m>=p)
```

```
            q=p; // q - lungime curentă n-uplu de interclasat de la sursa i
```

```
        else
```

```
            q=m;
```

```
        m=m-q;
```

```
        if (m>=p)
```

```
            r=p; // r - lungime curentă n-uplu de interclasat de la sursa j
```

```
        else
```

```
            r=m;
```

```
        m=m-r;
```

```
        while(q!=0 && r!=0)
```

```
        {
```

```
            // selectează cel mai mic element dintre a[i] și a[j]
```

```
            if(a[i]<a[j])
```

```
            {
```

```
                a[k]=a[i]; k=k+h; i=i+1; q=q-1;
```

```

    }
    else {
        a[k]=a[j]; k=k+h; j=j-1; r=r-1;
    }
}
//copiază restul secvenței j
while(r!=0)
{
    a[k]=a[j]; k=k+h; j=j-1; r=r-1;
}
// copiază restul secvenței i
while(q!=0)
{
    a[k]=a[i]; k=k+h; i=i+1; q=q-1;
}

// interschimbă indicii destinație (k și l)
h=-h; t=k; k=L; L=t;

}while(m!=0);

sus=!sus; // comutare sursă-destinație
p=p*2;    // dublarea lungimii n-uplului curent

}while(p<n); // n este numărul inițial de elemente de sortat

// mutare elemente sortate în secvența inițială (dacă este cazul)
if(!sus)
    for(i=1;i<=n;i++)
        a[i]=a[i+n];
}
-----

```

3.3.1.3. Analiza interclasării echilibrate cu 4 secvențe (2 căi)

- Deoarece la fiecare trecere p se dublează, îndeplinirea condiției $p > n$ presupune $\lceil \log_2 n \rceil$ **trecuri**, deci o **eficiență** de ordinul $O(\log_2 n)$.

- Fiecare pas, prin definiție, copiază întregul set de n elemente exact odată, prin urmare **numărul de mișcări** M este cunoscut exact:

$$M = n \cdot \lceil \log_2 n \rceil$$

- **Numărul de comparații** C este chiar mai mic decât M deoarece operația de copiere a resturilor **nu** presupune nici o comparație.
- Cu toate astea, întrucât tehnica interclasării presupune utilizarea unor dispozitive periferice, efortul de calcul necesar **operațiilor de mutare** poate domina efortul necesar **operațiilor de comparare** cu mai multe ordine de mărime, motiv pentru care **analiza numărului de comparații nu prezintă interes**.
- În aparență **tehnica interclasării echilibrate** aplicată unor **structuri tablou**, obține performanțe la nivelul celor mai performante tehnici de sortare.
- În realitate însă:
 - **Regia manipulării indicilor** este relativ ridicată.
 - **Necesarul dublu de zonă de memorie** este un dezavantaj decisiv.
- Din aceste motive această tehnică este rar utilizată în sortarea tablourilor.

- **Măsurătorile reale** efectuate situează **tehnica interclasării** aplicată **tablourilor** la nivelul performanțelor metodei **heapsort**, deci sub performanțele **quicksort**-ului.

3.3.2. Sortarea prin interclasare naturală

- **Tehnica de sortare prin interclasare nu** ia în considerare faptul că **datele inițiale** pot fi **parțial sortate**, subsecvențele interclasate având o lungime fixă predeterminată adică 2^k în trecerea k .
 - De fapt, oricare două subsecvențe ordonate de lungimi m și n pot fi interclasate într-o singură subsecvență ordonată de lungime $m+n$.
- **Tehnica de interclasare** care în fiecare moment combină **cele mai lungi secvențe ordonate posibile** se numește **sortare prin interclasare naturală**.
 - În cadrul acestei tehnici un rol central îl joacă noțiunea de **monotonie**, care va fi clarificată pe baza următorului exemplu.
- Se consideră următoarea secvență de chei:

| 1 13 | 2 4 7 | 6 18 | 9 10 14 | 11 | 3 75 |

- Se pun linii verticale la extremitățile secvenței precum și între elementele a_j și a_{j+1} , ori de câte ori $a_j > a_{j+1}$.
 - În felul acesta secvența a fost defalcată în **secvențe parțiale monotone**.
 - Secvențele obținute sunt de **lungime maximă**, adică **nu** pot fi prelungite fără a-și pierde proprietatea de a fi **monotone**.
- În general fie a_1, a_2, \dots, a_n o **secvență** oarecare de numere întregi.
- **Formal**, se înțelege prin **monotonie** orice secvență parțială a_i, \dots, a_j care satisface următoarele condiții [3.3.2.a]:

-
- 1) $1 \leq i \leq j \leq n$; { [3.3.2.a] }
 - 2) $a_k \leq a_{k+1}$ pentru orice $i \leq k < j$;
 - 3) $a_{i-1} > a_i$ sau $i=1$;
 - 4) $a_j > a_{j+1}$ sau $j=n$;
-

- Această definiție include și monotoniile cu un singur element, deoarece în acest caz $i=j$ și proprietatea 2) este îndeplinită, neexistând nici un k cuprins între i și $j-1$.
- **Sortarea naturală** este cea sortare care **interclasează monotonii**.
- **Sortarea naturală** se bazează pe următoarea **proprietate**:
 - Dacă se intercalează două secvențe a câte n monotonii fiecare, rezultă o secvență cu exact n monotonii.
 - Ca atare, la fiecare trecere numărul monotoniilor se înjumătățește și în consecință **numărul maxim de treceri** pentru sortare va fi $\lceil \log_2 n \rceil$.

- În consecință, pentru procesul integral de sortare:
 - **Numărul de mișcări** este $n \cdot \lceil \log_2 n \rceil$, în medie mai redus.
 - **Numărul de comparații** este însă mult mai mare deoarece pe lângă comparațiile necesare interclasării elementelor sunt necesare comparații între elementele consecutive ale fiecărei secvențe pentru a determina sfârșitul fiecărei monotonii.
- În continuare în dezvoltarea programului aferent acestei tehnici va fi utilizată aceeași **metodă a detalierilor succesive** (*"Stepwise refinement"*)
- Se utilizează o structură de date **fișier secvențial** asupra căreia se aplică **modelul de sortare prin interclasare neechilibrată în două faze**, utilizând **trei secvențe**.
- Algoritmul lucrează cu secvențele a, b și c .
 - Secvența c este cea care trebuie procesată și care în final devine secvența sortată.
 - În practică, din motive de securitate, c este de fapt o copie a secvenței inițiale.
- Se utilizează următoarele structuri de date [3.3.2.b].

{Sortarea prin interclasare naturală - structuri de date}

```
tip_secventa = FILE OF tip_element;           {[3.3.2.b]}
tip_secventa c;    {secvența de sortat}
tip_secventa a,b; {secvențe auxiliare}
```

- Secvențele a și b sunt auxiliare și ele servesc la defalcarea provizorie a lui c pe monotonii.
- **Fiecare trecere** constă din **două faze alternative** care se numesc **distribuire** respectiv **interclasare**.
 - În faza de **distribuire** monotoniile secvenței c se distribuie alternativ pe secvențele a și b .
 - În faza de **interclasare** se recombina în c , monotoniile de pe secvențele a și b (fig. 3.3.2).

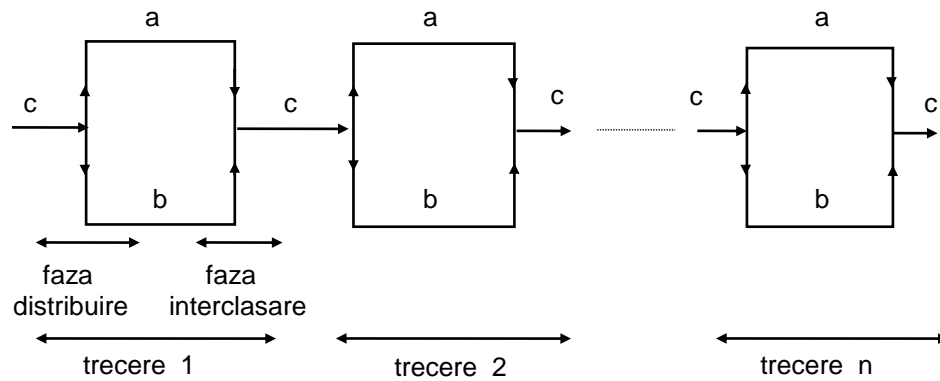


Fig. 3.3.2. Modelul sortării prin interclasare naturală. Trecri și faze

- Sortarea se termină în momentul în care numărul monotoniilor secvenței c devine egal cu 1.
- Pentru numărarea monotoniilor se utilizează variabila L .
- Prima formă a algoritmului de sortare naturală apare în secvența [3.3.2.c].
 - Cele două faze apar ca două **enunțuri**, care urmează să fie **rafinare** în continuare .

{Sortarea prin interclasare naturală - Pas de rafinare 0}

PROCEDURE InterclasareNaturala

{[3.3.2.c]}

```

int L;           {număr de monotonii interclasate}
tip_secventa c;  {secvența de sortat}
tip_secventa a,b; {secvențe auxiliare}

```

repetă

```

  Rewrite(a); Rewrite(b); Reset(c);

```

Distribuire;

```

  Reset(a); Reset(b); Rewrite(c);

```

```

  L=0;

```

Interclasare;

până când (L=1)

□ {repetă}

{InterclasareNaturala}

- **Procesul de rafinare** poate fi realizat:
 - (1) Prin **substituția directă** a enunțurilor cu secvențele care le corespund - proces cunoscut sub denumirea de „**rafinare prin inserție**”.
 - (2) Prin **interpretarea** enunțurilor ca **proceduri sau funcții** și procedând în consecință la dezvoltarea lor proces denumit „**rafinare prin selecție**”.
- În continuare, se prezintă procesul de dezvoltare al **algoritmului de sortare prin interclasare naturală**, utilizând **rafinarea prin selecție**.

- În secvențele [3.3.2.d] respectiv [3.3.2.e] apar primii pași de rafinare pentru **Distribuire** respectiv **Interclasare**.

{Sortarea prin interclasare naturală - rafinarea procedurii Distribuire}

```

PROCEDURE Distribuire                                     {[3.3.2.d]}
{distribuire monotoniile de pe c alternativ pe a și b}

tip_secventa c;    {secvența sursă}
tip_secventa a,b;  {secvențe destinație}

  repetă
    CopiazaMonotonia(c,a);
    dacă (NOT Eof(c)) CopiazaMonotonia(c,b)
  până când (Eof(c))
  □ {repetă}
{Distribuire}

```

- Această metodă de defalcare distribuie:
 - (1) Un număr egal de monotonii pe secvențele a respectiv b, dacă numărul de monotonii de pe secvența c este par.
 - (2) Cu o monotonie mai mult pe secvența a, dacă numărul de monotonii de pe secvența c este impar.
 - (3) După interclasarea monotoniiilor perechi, monotonia nepereche (dacă există) trebuie recopiată pe c.

{Sortarea prin interclasare naturală - rafinarea procedurii Interclasare}

```

PROCEDURE Interclasare                                   {[3.3.2.e]}
{interclasează monotoniile de pe a și b pe c}

tip_secventa c;    {secvența destinație}
tip_secventa a,b;  {secvențe sursă}
int L;             {număr de monotonii interclasate}

  repetă
    InterclasareMonotonie; L=L+1;
  până când (Eof(b))
  □ {repetă}
  dacă (NOT Eof(a))
    {monotonia nepereche}
    CopiazaMonotonia(a,c);
    L=L+1;
  □ {dacă}
{Interclasare}

```

- Procedurile **Distribuire** și **Interclasare** sunt redactate la rândul lor în termenii unor proceduri subordonate (**InterclasareMonotonie**, **CopiazaMonotonia**)

care se referă la o **singură monotonie** și care vor fi rafinate în continuare în [3.3.2.f] respectiv [3.3.2.g].

- Se introduce variabila booleană **sm** (sfârșit monotonie) care specifică dacă s-a ajuns sau **nu** la sfârșitul unei monotonii.
 - La epuizarea uneia dintre monotonii restul celeilalte este copiat în secvența destinație.

{Sortarea prin interclasare naturală - rafinarea procedurii CopiazaMonotonia}

```
PROCEDURE CopiazaMonotonia (tip_secventa x,y) {[3.3.2.f]}
    {copiază o monotonie de pe secvența x pe secvența y}

    boolean sm;      {indicator sfârșit monotonie}

    repeat
        CopiazaElement(x,y);
    until sm {sfârșit monotonie}
    □
{CopiazaMonotonia}
```

{Sortarea prin interclasare naturală - rafinarea procedurii InterclasareMonotonie}

```
PROCEDURE InterclasareMonotonie                                {[3.3.2.g]}
{intreclasează câte o monotonie de pe a și b într-o
monotonie pe c}

    tip_secventa c;      {secvența destinație}
    tip_secventa a,b;    {secvențe sursă}
    boolean sm;          {indicator sfârșit monotonie}

    repetă
        dacă (a.elemCurent.cheie<b.elemCurent.cheie)
            CopiazaElement(a,c);
            dacă sm CopiazaMonotonia(b,c);
            □
        altfel
            CopiazaElement(b,c);
            IF sm THEN CopiazaMonotonia(a,c);
            □ {altfel}
        □ {dacă}
    până când sm
    □ {repetă}
{InterclasareMonotonie}
```

- Pentru redactarea procedurilor de mai sus se utilizează o procedură subordonată **CopiazaElement**(tip_secventa x,y), care transferă elementul curent al secvenței sursă x în secvența destinație y, poziționând variabila sm funcție de atingerea sau **nu** a sfârșitului monotoniei.
- În acest scop se utilizează **tehnica "lookahead"** (tehnica scrutării) care presupune:
 - (1) Prelucrarea în pasul curent a elementului citit în pasul anterior.

| □ {dacă}
{CopiazăElement}

- După cum se observă:
 - La momentul **curent** se scrie pe secvența destinație y elementul `x.elemCurent` citit în pasul anterior.
 - Dacă `x.elemCurent` a fost ultimul element al secvenței, atunci prelucrarea s-a terminat.
 - Dacă `x.elemCurent` nu a fost ultimul element al secvenței, el se salvează în variabila aux de tip `tip_element` și se citește elementul următor `x.elemCurent` în vederea determinării sfârșitului de monotonie sm.
- Desigur unii dintre pașii de rafinare precizați pot suferi anumite modificări, funcție de natura secvențelor reale care se utilizează și de setul de operații disponibile asupra acestora.
- Din păcate, programul dezvoltat cu ajutorul acestei metode **nu** este corect în toate cazurile.

- Spre exemplu, defalcarea secvenței c cu 10 monotonii:

| 13 57 | 17 19 | 11 59 | 23 29 | 7 61 | 31 37 | 5 67 | 41 43 | 2 3 | 47 71 |

are drept consecință, datorită distribuției cheilor, formarea a 5 monotonii pe secvența a și a unei singure monotonii pe secvența b, în loc de 5 cum era de așteptat.

a: | 13 57 | 11 59 | 7 61 | 5 67 | 2 3 |

b: | 17 19 23 29 31 37 41 43 47 71 |

- Faza de interclasare conduce la o secvență cu două monotonii (în loc de 5)

c: | 13 17 19 23 29 31 37 41 43 47 57 71 | 11 59 |

deoarece în procesul de interclasare s-a ajuns la sfârșitul secvenței b, conform procedurii **Interclasare** [3.3.2.e] se mai copiază o **singură** monotonie din a.

- După trecerea următoare sortarea se încheie, dar rezultatul este **incorect**:

c: | 11 13 17 19 23 29 31 37 41 43 47 57 59 71 |

- Acest lucru se întâmplă deoarece **nu** a fost luat în considerare faptul că deși procesul de distribuire repartizează în mod egal monotonii pe secvențele a respectiv b, **numărul real de monotonii** pe cele două secvențe poate **diferi** foarte mult de **numărul preconizat**, datorită distribuției cheilor.
- Pentru a **remedia** această situație, este necesar ca procedura **Interclasare** să fie **modificată** astfel încât, în momentul în care se ajunge la sfârșitul unei secvențe, să copieze în c **tot** restul celeilalte secvențe.

- Versiunea finală revizuită a **algoritmului de sortare prin interclasare naturală** apare în [3.3.2.j].

```
-----
/*Sortarea prin interclasare naturală - varianta finală*/

#include <stdio.h>                                     /*[3.3.2.j]*/
#include <stdlib.h>
#include <string.h>

#define true (1)
#define false (0)
#define LUNGIME_MAXIMA_NUME_FISIER (256)

typedef int boolean;
typedef int tip_element;

//definire tip fisier
typedef struct
{
    char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
    FILE* file;
} tip_fisier;

//definire tip secvență
typedef struct tip_secventa
{
    tip_fisier fisier;
    tip_element elemCurent;
    boolean termPrelucr;
} tip_secventa;

const char nume_fisier_initial[] = "initial.txt";

int L;          //număr monotonii interclasate
boolean sm;     //indicator sfârșit monotonie
tip_secventa a,b,c; //cele trei secvențe

//copiază un element de pe secvența x pe secvența y
void CopiazaElement(tip_secventa* x, tip_secventa* y)
{
    tip_element aux;
    Write(&y->fisier, x->elemCurent); //scrie elementul curent al lui x pe y
    if (Eof(&x->fisier))               //x este ultimul element
    {
        sm=true;
        x->termPrelucr=true;
    }
    else
    {
        aux=x->elemCurent;              //salvare element curent
        Read(&x->fisier, &x->elemCurent); //citire element următor
        sm=aux > x->elemCurent;          //setare sfârșit monotonie
    }
}

//copiază o monotonie de pe secvența x pe secvența y
void CopiazaMonotonia(tip_secventa* x, tip_secventa* y)
{
    do
    {
        //x - secvența sursă în care se delimitează monotonia
        //y - secvența destinație în care se copiază monotonia
        CopiazaElement(x, y);
    }
    while (!sm);
}

//distribuie monotoniile de pe c alternativ pe a și b
void Distribuie()
```

```

{
    Rewrite(&a);
    Rewrite(&b);
    Reset(&c);

    c.termPrelucr=Eof(&c.fisier);
    Read(&c.fisier, &c.elemCurent);

    do
    {
        CopiazaMonotonia(&c, &a);
        if (!c.termPrelucr)
            CopiazaMonotonia(&c, &b);
    }
    while (!c.termPrelucr);
}

//interclasează o monotonie de pe a și b pe c
void InterclasareMonotonie()
{
    do
    {
        if (a.elemCurent < b.elemCurent)
        {
            CopiazaElement(&a, &c);
            if (sm)
                CopiazaMonotonia(&b, &c);
        }
        else
        {
            CopiazaElement(&b, &c);
            if (sm)
                CopiazaMonotonia(&a, &c);
        }
    }
    while (!sm);
}

//interclasează monotoniile de pe a și b pe c
void Interclasare()
{
    Reset(&a);
    Reset(&b);
    Rewrite(&c);

    a.termPrelucr = Eof(&a.fisier);
    b.termPrelucr = Eof(&b.fisier);

    //tehnica lookahead
    if (!a.termPrelucr)
        Read(&a.fisier, &a.elemCurent); //primul element al secvenței a
    if (!b.termPrelucr)
        Read(&b.fisier, &b.elemCurent); //primul element al secvenței b

    //interclasează monotoniile de pe a și b pe c
    while (!a.termPrelucr && !b.termPrelucr)
    {
        InterclasareMonotonie();
        L=L+1;
    }

    //copiază monotoniile rămase de pe b pe c (dacă există)
    while (!b.termPrelucr)
    {
        CopiazaMonotonia(&b, &c);
        L=L+1;
    }

    //copiază monotoniile rămase de pe a pe c (dacă există)
    while (!a.termPrelucr)

```

```

    {
        CopiazaMonotonia(&a, &c);
        L=L+1;
    }
}

//Interclasare naturală - programul principal
void InterclasareaNaturala()
{
    do
    {
        Distribuire();
        L=0;
        Interclasare();
    }
    while (L != 1);
    Close(a.fisier);
    Close(b.fisier);
    Close(c.fisier);
}

```

3.3.2.1. Analiza sortării prin interclasare naturală

- După cum s-a mai precizat, la analiza unei **metode de sortare externă**, **numărul comparațiilor de chei nu** are importanță practică, deoarece durata prelucrărilor în unitatea centrală a sistemului de calcul este neglijabilă față de durata acceselor la memoriile externe.
 - Din acest motiv **numărul mutărilor M** va fi considerat drept **unic indicator de performanță**.
- În cazul **sortării prin interclasare naturală**:
 - La o **trecere**, în fiecare din cele două faze (defalcare și interclasare) se mută toate elementele, deci **numărul de mișcări** $M = 2 \cdot n$.
 - După fiecare trecere **numărul monotoniiilor** se micșorează de două ori, uneori chiar mai substanțial, motiv pentru care a fost necesară și modificarea anterioară a procedurii **Interclasare**.
 - Știind că numărul de monotonii inițiale este n , **numărul maxim de treceri** este $\lceil \log_2 n \rceil$, astfel încât în cel mai defavorabil caz **numărul de mișcări** $M = 2 \cdot n \cdot \lceil \log_2 n \rceil$, în medie simțitor mai redus.

3.3.3. Sortarea prin interclasare multiplă echilibrată

- Întrucât efortul de sortare este proporțional cu **numărul de treceri**, o cale de **reducere**, respectiv **de creștere a eficienței** a acestuia este aceea de a **distribui monotoniiile** pe mai mult decât două secvențe.
- În consecință:
 - Primul pas al interclasării a r **monotonii** care sunt distribuite pe N **secvențe** conduce la un r/N monotonii pe fiecare secvență.

- Al doilea pas conduce la r/N^2 monotonii pe fiecare secvență.
- Al treilea la r/N^3 monotonii ș.a.m.d.
- Această metodă de sortare se numește **interclasare multiplă-N**.
- **Numărul total de treceri** k , în cazul sortării a n secvențe prin **interclasare multiplă-N** este $k = \lceil \log_N n \rceil$, iar **numărul total de mișcări** $M = n \cdot \lceil \log_N n \rceil$.
 - O modalitate de implementare a acestei situații o reprezintă **interclasarea multiplă echilibrată** care se realizează într-o **singură fază**.
- **Interclasarea multiplă echilibrată** presupune că la fiecare trecere există un număr egal de secvențe de intrare și de ieșire, monotoniiile de pe primele fiind interclasate și imediat redistribuite pe celelalte.
 - Dacă se utilizează N secvențe (N par), avem de fapt de-a face cu o **interclasare multiplă echilibrată cu $N/2$ căi**.
- **Schema de principiu** a acestei metode apare în figura 3.3.3.a.

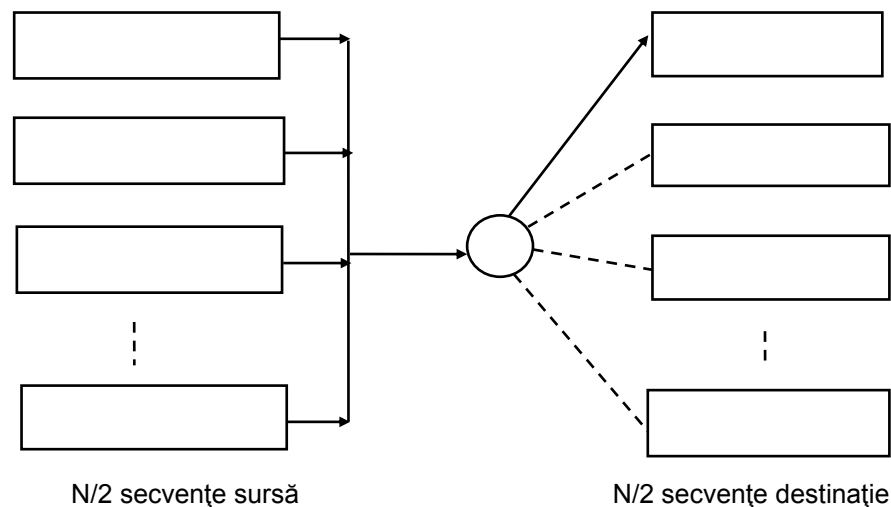


Fig. 3.3.3.a. Modelul interclasării multiple echilibrate cu $N/2$ căi

- **Algoritmul** care va fi dezvoltat în continuare se bazează pe o structură specifică de date și anume, **tabloul de secvențe**.
- Față de **procedeul interclasării multiple echilibrate cu 2 căi** utilizat anterior, trecerea la mai multe căi presupune **modificări esențiale**:
 - (1) Procesul de interclasare trebuie să gestioneze o **listă a secvențelor active**, din care va elimina pe rând secvențele ale căror monotonii s-au epuizat.
 - (2) Trebuie implementată **comutarea** grupelor de **secvențe de intrare și ieșire** după fiecare trecere.
- Pentru aceasta se definesc structurile de date din secvența [3.3.3.a].

```

-----
/*Structuri de date pentru sortarea prin interclasare
multiplă echilibrată*/

/*[3.3.3.a]*/

/*definire structură fișier secvențial*/
typedef struct
{
    char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
    FILE* file;
} tip_fisier;

/*definire structură tip secvență*/
typedef struct tip_secventa
{
    tip_fisier fisier;          /*fișier suport secvență*/
    tip_element elemCurent;    /*tampon secvență*/
    boolean termPrelucr;       /*indicator terminare procesare
                                secvență*/
} tip_secventa;

int Nr_Secvențe=n; /*număr de secvențe utilizate în sortare*/
tip_secventa f0;   /*secvența inițială*/
tip_secventa F[Nr_Secvențe]; /*tablou de secvențe*/
int t[Nr_Secvențe],td[Nr_Secvențe]; /*tablouri de indici*/
-----

```

- Se introduce o nouă structură de date și anume **tabloul de secvențe** F ale cărui elemente aparțin lui tip_secventa.
- Variabila Nr_Secvențe memorează numărul de **secvențe** utilizate în sortare.
- Se presupune de asemenea că secvența de elemente care urmează să fie sortată este furnizată ca o variabilă f0 și că pentru procesul de sortare se utilizează Nr_Secvențe secvențe (număr par).
- **Problema comutării secvențelor** se poate rezolva introducând **tabloul de indici** t care are drept componente indici care identifică secvențele.
 - Astfel în loc de a adresa o secvență din tabloul F direct prin indicele i, aceasta va fi adresată **via** tabloul t, respectiv F[t[i]] în loc de F[i].
 - Inițial t[i]=i pentru toate valorile lui i.
 - **Comutarea secvențelor** constă de fapt în **interschimbarea** perechilor de componente ale tabloului t după cum urmează, unde $NH = Nr_Secvențe/2$.

$$\begin{array}{lcl}
 t[1] & \leftrightarrow & t[NH+1] \\
 t[2] & \leftrightarrow & t[NH+2] \\
 & \dots & \\
 t[NH] & \leftrightarrow & t[N]
 \end{array}$$
- În continuare secvențele F[t[1]], ..., F[t[NH]] vor fi considerate **întotdeauna secvențe de intrare**, iar secvențele F[t[NH+1]], ..., F[t[N]] drept **secvențe de ieșire** (fig.3.3.3.b).


```

    repetă
        L=L+1;
        *interclasează câte o monotonie de pe fiecare
            intrare activă pe F[t[j]];
        dacă j<Nr_Secvențe j=j+1; altfel j=NH+1;
    până când *toate intrările active au fost epuizate;
    □ /*repetă*/
    *comută secvențele;
    până când L=1;
    □ /*repetă*/
    /*secvența sortată este t[1]*/
    /*SortareMultiplaEchilibrata*/

```

- După cum se observă, procesul de **sortare multiplă echilibrată** constă din **trei pași**.
- (1) Primul pas realizează **distribuția monotoniiilor inițiale** și este materializat de prima buclă **repetă**. În acest pas:
 - Monotoniiile de pe secvența inițială f_0 sunt distribuite succesiv pe secvențele sursă care sunt indicate de j .
 - După copierea fiecărei monotonii, indicele j care parcurge ciclic domeniul $[1..NH]$ este incrementat.
 - Distribuția se termină când se epuizează f_0 .
 - Numărul de monotonii distribuite se contorizează în L .
- (2) Pasul al doilea realizează **inițializarea** tabloului t (bucla **pentru**).
- (3) Pasul al treilea realizează **interclasarea** secvențelor de intrare $F[t[1]] \dots F[t[NH]]$ în secvențele de ieșire $F[t[NH+1]] \dots F[t[N]]$.
- **Principiul** interclasării este următorul:
 - Se iau toate intrările active și se interclasează câte o monotonie de pe fiecare într-o singură monotonie care se depune pe secvența $F[t[j]]$.
 - Se avansează la secvența de ieșire următoare, j parcurgând ciclic domeniul $[NH+1..N]$.
 - Se reia procedeul până la epuizarea tuturor intrărilor (bucla **repetă** interioară).
 - În acest moment se comută secvențele, intrările devin ieșiri și invers, și se reia interclasarea.
 - Procesul continuă până când numărul de monotonii de interclasat ajunge egal cu 1 (bucla **repetă** exterioră).
- În continuare se prezintă **rafinarea** enunțurilor implicate în algoritm.
- În secvența [3.3.3.c] apare rafinarea enunțului *copiază o monotonie de pe f_0 în secvența $F[j]$, utilizat în distribuția inițială a monotoniiilor.

```

/*rafinare enunț: copiază o monotonie de pe secvența f0 pe
secvența F[j]*/

```

```

/*[3.3.3.c]*/

```

```

/*f0 nu este vidă. Primul element din f0 este deja citit*/
tip_element buf;

repetă
    buf=f0.curent; /*buf memorează elementul curent*/
    dacă Eof(f0.fisier)
        f0.termPrelucr=true; /*tehnica lookahead*/
    altfel
        *citește(f0.fisier,f0.curent); /*citește elementul
                                         următor*/
□ /*dacă*/
/*scrie elementul curent pe secvența F[j]*/
*scrie(F[j].fisier,buf);
până când ((buf.cheie>f0.curent.cheie) OR f0.termPrelucr);
□ /*repetă*/

```

- Urmează rafinarea enunțului **inițializare secvențe de intrare*.
 - Pentru început trebuie identificate secvențele **curente** de intrare deoarece numărul **secvențelor active** poate fi mai mic decât numărul total al secvențelor sursă NH.
 - Prin **secvență activă** se înțelege o secvență pe care mai există monotonii de interclasat.
- De fapt, **numărul secvențelor** se reduce pe măsură ce se reduce **numărul monotoniiilor**.
 - Practic **nu** pot exista mai multe secvențe decât monotonii astfel încât sortarea se termină când mai rămâne **o singură secvență**.
- Se introduce variabila k1 pentru a preciza **numărul actual de secvențe** de intrare **active**, adică acelea care mai au monotonii.
- Cu aceste precizări, inițializarea **secvențelor de intrare** va fi [3.3.3.d]:

```

/*rafinare enunț: inițializare secvențe intrare*/

pentru i=1 la k1  Reset(F[t[i]]); /*[3.3.3.d]*/

```

- Datorită procesului repetat de interclasare, tendința lui k1 este de a **descrește** odată cu epuizarea monotoniiilor, deci condiția **toate intrările epuizate se poate exprima prin relația k1=0*.
- Enunțul **interclasează câte o monotonie de pe fiecare intrare pe F[t[j]] este mai complicat*.
 - El constă în selecția repetată a elementului cu cea mai mică cheie dintre **sursele disponibile** și trecerea lui pe **secvența de ieșire curentă**.
 - De la fiecare **secvență sursă** se parcurge câte **o monotonie** al cărei sfârșit poate fi atins fie drept consecință a uneia din următoarele două **condiții**:
 - (1) **Epuizarea** secvenței curente (eliminarea secvenței).

- (2) **Citirea unei chei mai mici** decât cea curentă (închidere monotonie).
- În cazul (1) **secvența** este **eliminată** și k_1 decrementat.
- În cazul (2) secvența este **exclusă temporar** de la selecția cheilor, până când se termină crearea monotoniei curente pe secvența de ieșire, operație numită **”închidere monotonie”**.
 - În acest scop se utilizează un al doilea indice k_2 care precizează **numărul de secvențe sursă disponibile curent** pentru selecția cheii următoare.
 - Valoarea sa inițială egală cu k_1 , se decrementează ori de câte ori se epuizează o monotonie din cauza condiției (2).
 - Când k_2 devine 0, s-a **terminat** interclasarea câte unei monotonii de la fiecare **intrare activă pe secvența de ieșire curentă**.
- Primul pas de rafinare al acestui enunț apare în [3.3.3.e].

/*rafinare enunț: interclasează câte o monotonie de pe fiecare intrare activă pe F[t[j]]*/

```

                                                                    /*[3.3.3.e]*/
int k1; /*numărul actual de secvențe de intrare active*/
int k2; /*numărul de secvențe sursă care mai au monotonii*/

pentru i=1 la k1  Reset(F[t[i]].fisier);
k2=k1; /*inițial k2=k1*/
repetă
  *selectează cheia minimă. Fie td[mx] indicele
    secvenței care o conține;
  buf=F[td[mx]].curent; /*tehnica look-ahead*/
  dacă (Eof(F[t[mx]].fisier))
    F[t[mx]].termPrelucr=true;
    altfel /*citește elementul următor*/
      *citeste(F[t[mx].fisier,F[t[mx].curent);
□ /*dacă*/
  *scrie(F[T[j]].fisier,buf); /*scrie elementul curent*/
  dacă (Eof(F[t[mx]].fisier))
    *elimină secvența; /*detectare sfârșit secvență*/
    altfel
      dacă (buf.cheie>F[t[mx]].curent.cheie)
        *închide monotonia; /*detectare sfârșit monotonie*/
□ /*dacă*/
până când k2=0;
□ /*repetă*/

```

- Din păcate, introducerea lui k_2 **nu** este suficientă, deoarece pe lângă numărul secvențelor disponibile trebuie **să se știe exact** care sunt acestea.
- O soluție în acest sens poate să o constituie utilizarea unui **tablou cu componente booleene** care indică disponibilitatea secvențelor.
- O soluție mai eficientă însă este utilizarea unui al doilea **tablou de indici** td . Indicii respectivi se referă numai la secvențele de intrare care se procesează.

- Tabloul td apare reprezentat schematic în figura 3.3.3.c.

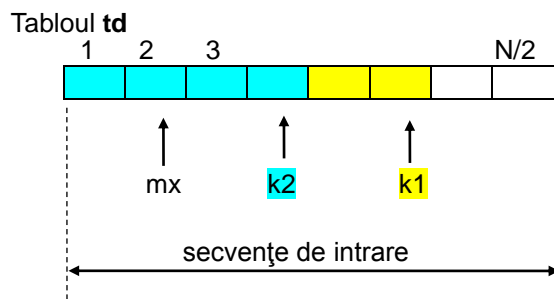


Fig. 3.3.3.c. Tabloul td utilizat ca auxiliar în procesul de interclasare

- Acest tablou este utilizat **în locul tabloului t** pentru **secvențele de intrare** astfel încât $td[1], \dots, td[k2]$ sunt indicii **secvențelor disponibile**.
- Tabloul td se inițializează la începutul fiecărei interclasări prin copierea indicilor secvențelor de intrare din tabloul t ($t[1], \dots, t[k1]$).
- Indicele $k1$ se inițializează cu:
 - Valoarea $Nr_Secvente/2$ dacă numărul de monotonii L este mai mare ca $Nr_Secvente/2$.
 - Valoarea curentă a lui L , dacă numărul de monotonii L este mai mic ca $Nr_Secvente/2$.
 - Se precizează faptul că L reprezintă numărul de monotonii interclasate în faza anterioară.
 - Indicele $k1$ precizează numărul de secvențe **active**.
 - Restul secvențelor, (de la $k1+1$ până la $Nr_Secvente/2$) **nu** mai au monotonii fiind **terminate fizic**, motiv pentru care acestea nici nu se mai consemnează.
- Indicele $k2$ care se inițializează cu valoarea lui $k1$, precizează **numărul de secvențe active** care mai **au monotonii** în trecerea curentă.
 - Secvențele cuprinse între indicii $k2$ și $k1$ și-au epuizat monotoniiile în trecerea curentă **fără** a fi însă epuizate fizic.
- În aceste condiții, **rafinarea enunțului $*închide$** monotonia corespunzător condiției (2), se implementează după cum urmează.
 - Fie mx indicele în tabloul td al secvenței pentru care s-a terminat monotonia curentă:
 - (1) Se **interschimbă** în tabloul td pozițiile $td[mx]$ și $td[k2]$.
 - (2) Se decrementează $k2$.
- În ceea ce privește enunțul **$*elimina$ secvența**, corespunzător condiției (1), considerând ca tot mx este indicele în tabloul td al secvenței care s-a epuizat,

rafinarea presupune următoarele. Se menționează că epuizarea fizică a secvenței presupune și închiderea ultimei monotonii.

- (1) Se mută în tabloul t_d secvența precizată de k_2 în locul secvenței precizate de m_x (închidere monotonie) adică $t_d[m_x] = t_d[k_2]$.
 - (2) Se mută secvența precizată de k_1 în locul secvenței indicate de k_2 (eliminarea secvenței) adică $t_d[k_2] = t_d[k_1]$
 - (3) Se decrementează k_1 .
 - (4) Se decrementează k_2 .
 - (5) Toate operațiile se reiau similar și pentru tabloul t .
- Forma finală a **sortării multiple echilibrate** apare în secvența [3.3.3.f].

/*Sortarea prin interclasare multiplă echilibrată - varianta finală*/

/*3.3.3.f*/

```
#define true (1)
#define false (0)
#define Nr_Secvente (4)
#define LUNGIME_MAXIMA_NUME_FISIER (256)

typedef int boolean;
typedef int tip_element;

// definire structura fisier
typedef struct
{
    char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
    FILE* file;
} tip_fisier;

// definire structura secvență
typedef struct tip_secventa
{
    tip_fisier fisier;
    tip_element elemCurent;
    boolean termPrelucr;
} tip_secventa;

// definire operatori referitori la prelucrarea secvențelor
const char nume_fisier_initial[] = "initial.txt";

void Rewrite(tip_secventa* secventa)
{
    fclose(secventa->fisier.file);
    secventa->fisier.file = fopen(secventa->fisier.nume_fisier, "w+");
}

void Read(tip_fisier* fisier, tip_element* elem_curent)
{
    fscanf(fisier->file, "%d ", elem_curent);
}

void Write(tip_fisier* fisier, tip_element buf)
{
    fprintf(fisier->file, "%d ", buf);
}

void Reset(tip_secventa* secventa)
{
    fseek(secventa->fisier.file, 0, SEEK_SET);
}
```

```

        secventa->termPrelucr = false;
    }

    void InitSecventa(tip_secventa* secventa, const char* nume_fisier, const char*
mode)
    {
        secventa->fisier.file = fopen(nume_fisier, mode);
        strcpy(secventa->fisier.nume_fisier, nume_fisier);
        secventa->termPrelucr = false;
    }

    void InitSecvente(tip_secventa* f0, tip_secventa secvente[Nr_Secvente])
    {
        InitSecventa(f0, nume_fisier_initial, "r");

        for (int i = 1; i <= Nr_Secvente; ++i)
        {
            char nume_fisier[101] = "aux_";
            sprintf(nume_fisier, "aux_%d.txt", i);

            InitSecventa(&secvente[i], nume_fisier, "w+");
        }
    }

    boolean Eof(tip_fisier* fisier)
    {
        int current_position = ftell(fisier->file);
        fseek(fisier->file, 0, SEEK_END);
        int size = ftell(fisier->file);
        fseek(fisier->file, current_position, SEEK_SET);
        return feof(fisier->file) || size == current_position;
    }

    // interclasarea multiplă echilibrată
    void InterclasareMultiplaEchilibrata()
    {
        int i, j, mx, tx, NH;
        int k1, k2, L;
        int x, min;

        // td = tablou secvențe de intrare
        // t = tablou secvențe pentru comutare
        int t[Nr_Secvente + 1]; // tablou de indici comutare secvențe
        int td[Nr_Secvente/2 + 1]; // tablou de indici secvențe de intrare
        tip_secventa F[Nr_Secvente + 1]; // tablou de secvențe
        tip_secventa f0; // secvența inițială
        tip_element buf;

        InitSecvente(&f0, F);

        NH = Nr_Secvente / 2;

        for (i = 1; i <= NH; i++) // inițializare secvențe
            Rewrite(&F[i]);

        j = NH; // indice parcurgere secvențe
        L = 0; // contor de monotonii distribuite

        Read(&f0.fisier, &f0.elemCurent); // primul element din f0 - lookahead

        do // distribuirea monotoniei inițiale pe t[1], ..., t[NH]
        {
            if (j < NH) j = j + 1; // parcurgere secvențe circular
            else j = 1;

            do // copiază o monotonie de pe secvența f0 pe secvența j
            {
                buf = f0.elemCurent;
                if (Eof(&f0.fisier))
                    f0.termPrelucr = true; // tehnica lookahead
            }
        }
    }

```

```

        else
            Read(&f0.fisier, &f0.elemCurent); // citește elementul
                                                următor
            Write(&F[j].fisier, buf); // scrie elementul curent pe secvența
                                      F[j]
        }
        while (buf <= f0.elemCurent && !f0.termPrelucr);
        L = L + 1;
    }
    while (!f0.termPrelucr);

    // inițializare tablou de indici t
    for (i = 1; i <= Nr_Secvente; i++) {
        t[i] = i;
    }

    do // interclasează de pe t[1],...,t[NH] pe t[NH + 1], ..., t[N]
    {
        if (L < NH) k1 = L;
        else k1 = NH;

        for (i = 1; i <= k1; i++) // inițializare secvențe intrare
            // k1 este numărul secvențelor de intrare active
        {
            Reset(&F[t[i]]);
            td[i] = t[i]; // inițializare tablou de indici de secvențe de
                          intrare
            Read(&F[td[i]].fisier, &F[td[i]].elemCurent);
        }

        L = 0; // numărul de monotonii interclasate
        j = NH + 1; // j este indexul secvenței de ieșire

        //interclasează câte o monotonie de pe fiecare intrare activă pe F[t[j]]

        do // interclasează câte o monotonie de pe td[1], ...td[k2] pe td[j]
        {
            L = L + 1;
            k2 = k1; // k2=nr de secvențe active

            do // selectează elementul cu cheia minimă
            {
                i = 1;
                mx = 1;
                min = F[td[i]].elemCurent;

                while (i < k2)
                {
                    i = i + 1;
                    x = F[td[i]].elemCurent;
                    if (x < min)
                    {
                        min = x;
                        mx = i;
                    }
                }

                // td[mx] conține elementul minim. Se trece pe td[j]
                buf = F[td[mx]].elemCurent;
                if (Eof(&F[td[mx]].fisier)) F[td[mx]].termPrelucr=true;
                else
                    Read(&F[td[mx]].fisier, &F[td[mx]].elemCurent);
                Write(&F[t[j]].fisier, buf);

                if (F[td[mx]].termPrelucr)
                {
                    // elimină secvența
                    Rewrite(&F[td[mx]]);
                    td[mx] = td[k2];
                    td[k2] = td[k1];
                    // actualizare tablou indici t

```



```

        t[mx] = t[k2];
        t[k2] = t[k1];
        k1 = k1 - 1;
        k2 = k2 - 1;
    }
    else if (buf > F[td[mx]].elemCurent)
    {
        // închide monotonia
        tx = td[mx];
        td[mx] = td[k2];
        td[k2] = tx;
        k2 = k2 - 1;
    }
}
while (k2 != 0); // terminare interclasare câte o monotonie de
                // la intrari pe t[j]

if (j < Nr_Secvente) j = j + 1; // selectează următoarea
                                // secvență destinație
else j = NH + 1;
}
while (k1 != 0); // toate intrările au fost epuizate

for (i = 1; i <= NH; i++) // comută secvențele
{
    tx = t[i];
    t[i] = t[i + NH];
    t[i + NH] = tx;
}
}
while (L != 1); // fișierul sortat se găsește pe t[1]

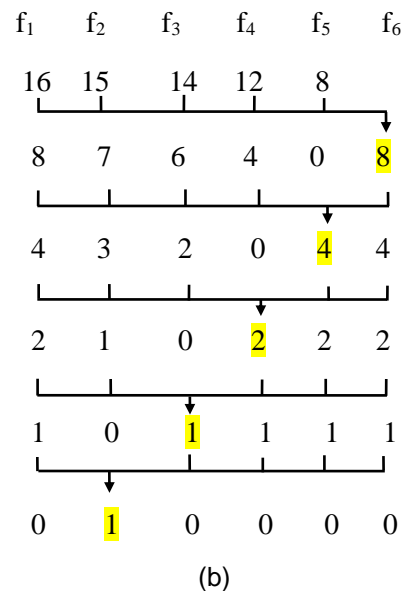
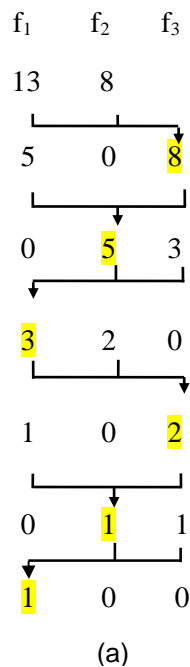
fclose(f0.fisier.file);
for (i = 1; i <= Nr_Secvente; ++i)
{
    fclose(F[i].fisier.file);
}
} // interclasare multiplă echilibrată
-----

```

3.3.4. Sortarea polifazică

- **Metoda interclasării echilibrate** contopește operațiile de distribuire și interclasare într-o aceeași fază, utilizând mai multe secvențe de intrare și mai multe secvențe de ieșire, care **nu** sunt folosite în totalitate.
- R.L. Gilstad, a propus **metoda sortării polifazice** care înlătură acest neajuns.
 - În plus în cadrul acestei metode însăși noțiunea de trecere devine difuză.
- Pentru început se consideră un **exemplu** cu **trei secvențe**.
 - În fiecare moment, sunt interclasate monotoniile de pe două secvențe pe cea de-a treia.
 - Ori de câte ori una din secvențele sursă se epuizează, ea devine imediat destinația operației de interclasare a celorlalte două secvențe (secvența neterminată și fosta destinație).
 - Procesul se termină când rămâne o singură monotonie.
- După cum se cunoaște, din procesul de interclasare a n monotonii de pe fiecare dintre secvențele de intrare, rezultă n monotonii pe secvența de ieșire.

- În cadrul acestei metode, pentru simplificare, se va vizualiza numai **numărul de monotonii** de pe fiecare secvență în locul cheilor propriu-zise.
- Astfel, în fig. 3.3.4.a (a) se presupune că secvențele de intrare f_1 și f_2 conțin 13 și respectiv 8 monotonii.
 - La prima "trecere", sunt interclasate de pe f_1 și f_2 pe f_3 8 monotonii,
 - La a doua "trecere" sunt interclasate de pe f_1 și f_3 pe f_2 cele 5 monotonii rămase, etc.
 - În final f_1 este secvența sortată.



- În aceeași figură (b) se prezintă un exemplu de sortare polifazică a 65 de monotonii utilizând 6 secvențe.
- Această tehnică este **mai eficientă** decât interclasarea echilibrată deoarece, fiind date N secvențe, ea lucrează la interclasare cu $N-1$ secvențe în loc de $N/2$.
 - Astfel **numărul de treceri** este aproximativ $\log_N n$, unde n este numărul de elemente de sortat iar N **gradul operației de interclasare** (numărul de secvențe sursă).
- În exemplele prezentate însă, **distribuția inițială a monotonilor** a fost aleasă cu mare grijă.
- Pentru a determina care dintre distribuțiile inițiale ale monotonilor asigură o funcționare corespunzătoare a algoritmului, se alcătuiesc tabelele din figurile 3.3.4.b și 3.3.4.c.
- Completarea celor două tabele se realizează pornind de la cele două exemple din fig. 3.3.4.a:
 - Fiecărei treceri îi corespunde un rând în tabelă.

- Trecherile se parcurg de jos în sus în figura 3.3.4.a, iar tabela se completează în ordine inversă, adică de sus în jos.
- Fiecare rând din tabel, cu excepția ultimului, conține pe prima poziție (a_1), **numărul de monotonii ale secvenței destinație** din trecerea corespunzătoare.
- În continuare se trec succesiv în tabel numerele de monotonii corespunzătoare din cadrul trecerii.
- Fiecare trecere în figura 3.3.4 se parcurge de la stânga la dreapta (începând cu secvența destinație) și se consideră circulară.
- Ultimul rând din tabel conține **situația inițială** a distribuției monotoniiilor adică cea dinaintea demarării procesului de sortare.

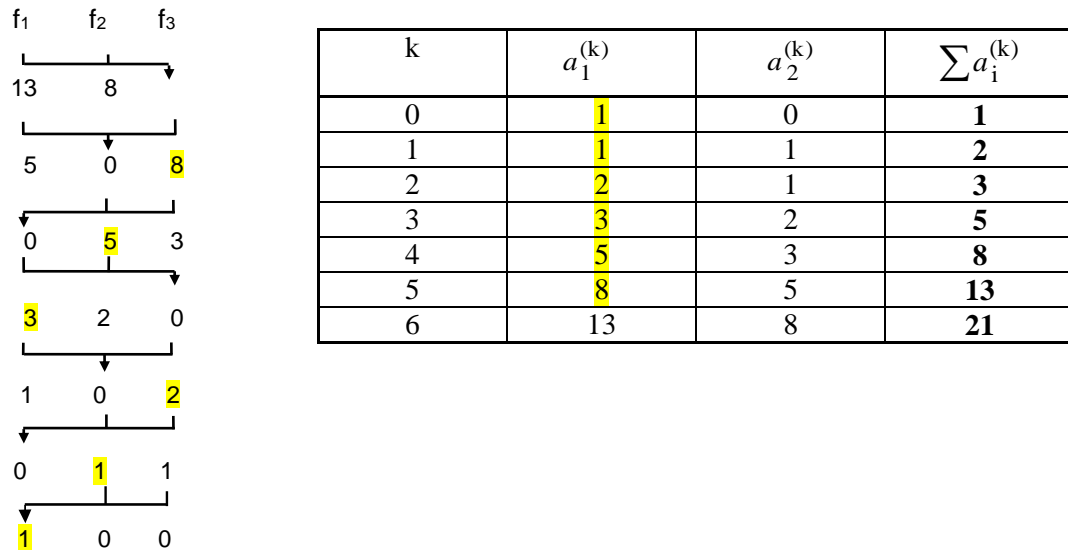


Fig. 3.3.4.b. Distribuția perfectă a monotoniiilor pe trei secvențe

- Din tabloul din figura 3.3.4.b se pot deduce următoarele relații [3.3.4.a]:

$$\begin{aligned}
 a_2^{(k+1)} &= a_1^{(k)} \\
 a_1^{(k+1)} &= a_1^{(k)} + a_2^{(k)} = a_1^{(k)} + a_1^{(k-1)} \quad \text{pentru } k > 0 \\
 \text{unde } a_1^{(0)} &= 1 \quad \text{și} \quad a_2^{(0)} = 0
 \end{aligned}
 \tag{3.3.4.a}$$

- Dacă facem $a_i^{(i)} = f_i^{(1)}$ prin înlocuire rezultă [3.3.4.b]:

$$\begin{aligned}
 f_{i+1}^{(1)} &= f_i^{(1)} + f_{i-1}^{(1)} \quad \text{pentru } i \geq 1 \\
 f_1^{(1)} &= 1 \\
 f_0^{(1)} &= 0
 \end{aligned}
 \tag{3.3.4.b}$$

- Aceasta este însă **regula recursivă** care definește numerele lui **Fibonacci de ordinul 1**:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Adică, fiecare număr din acest șir este **suma celor doi predecesori** ai săi.
- În consecință, pentru cazul **sortării cu trei secvențe**:
 - **Numerele inițiale de monotonii** pe cele două secvențe trebuie să fie **două numere Fibonacci de ordinul 1 consecutive**.
 - **Numărul total inițial de monotonii** este suma celor două numere Fibonacci de ordinul 1 consecutive, care în acest caz este tot un număr Fibonacci de ordinul 1.
- Pentru cel de-al doilea exemplu se obțin următoarele formule [3.3.4.c]:

k	$a_1^{(k)}$	$a_2^{(k)}$	$a_3^{(k)}$	$a_4^{(k)}$	$a_5^{(k)}$	$\sum a_i^{(k)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Fig. 3.3.4.c. Distribuția perfectă a monotoniiilor pe 6 secvențe

$$\begin{aligned}
 a_5^{(k+1)} &= a_1^{(k)} \\
 a_4^{(k+1)} &= a_1^{(k)} + a_5^{(k)} = a_1^{(k)} + a_1^{(k-1)} \\
 a_3^{(k+1)} &= a_1^{(k)} + a_4^{(k)} = a_1^{(k)} + a_1^{(k-1)} + a_1^{(k-2)} \\
 a_2^{(k+1)} &= a_1^{(k)} + a_3^{(k)} = a_1^{(k)} + a_1^{(k-1)} + a_1^{(k-2)} + a_1^{(k-3)} \\
 a_1^{(k+1)} &= a_1^{(k)} + a_2^{(k)} = a_1^{(k)} + a_1^{(k-1)} + a_1^{(k-2)} + a_1^{(k-3)} + a_1^{(k-4)}
 \end{aligned}$$

[3.3.4.c]

- Dacă facem $a_i^{(i)} = f_i^{(4)}$ rezultă [3.3.4.d] adică numerele **Fibonacci de ordinul 4** caracterizate prin proprietatea că fiecare număr din șir este suma celor 5 numere Fibonacci care îl preced.

$$\begin{aligned}
 f_{i+1}^{(4)} &= f_i^{(4)} + f_{i-1}^{(4)} + f_{i-2}^{(4)} + f_{i-3}^{(4)} + f_{i-4}^{(4)} \quad \text{pentru } i \geq 4 \\
 \text{unde } f_4^{(4)} &= 1, \quad f_i^{(4)} = 0 \quad \text{pentru } i < 4
 \end{aligned}$$

[3.3.4.d]

În general, **numerele Fibonacci de ordinul p** sunt definite după cum urmează [3.3.4.e]:

$$\begin{aligned}
 f_{i+1}^{(p)} &= f_i^{(p)} + f_{i-1}^{(p)} + \dots + f_{i-p}^{(p)} \quad \text{pentru } i \geq p \\
 \text{unde } f_p^{(p)} &= 1, \quad f_i^{(p)} = 0 \quad \text{pentru } 0 \leq i < p
 \end{aligned}$$

[3.3.4.e]

- **Numerele Fibonacci** de ordinul 1 sunt cele obișnuite.

- În cazul **sortării polifazice** cu n secvențe, **numerele inițiale de monotonii** de pe cele $n-1$ secvențe sursă sunt:
 - O **sumă** de $n-1$ numere Fibonacci consecutive de ordinul $n-2$ pe prima secvență.
 - O **sumă** de $n-2$ numere Fibonacci consecutive de ordinul $n-2$ pe a doua secvență.
 - O **sumă** de $n-3$ numere Fibonacci consecutive de ordinul $n-2$ pe a treia secvență.
 - Ș.a.m.d.
 - Pe ultima secvență (cea numerotată cu $n-1$) trebuie să existe inițial un număr de monotonii egal cu 1 număr Fibonacci de ordinul $n-2$.
- Aceasta implică faptul că **metoda sortării polifazice** este aplicabilă numai în cazul sortării unei secvenței al cărei **număr inițial de monotonii** este **suma** a $n-1$ de astfel de **sume Fibonacci**.
 - O astfel de distribuție a monotoniiilor inițiale se numește **distribuție perfectă**.
- În cazul sortării polifazice cu 6 secvențe ($n=6$), sunt necesare numere Fibonacci de ordinul $n-2=4$:

0, 0, 0, 0, 1, 1, 2, 4, **8**, 16, 31, 61, 120, ...

- **Distribuția inițială** a monotoniiilor, prezentată în fig. 3.3.4.a, se stabilește după cum urmează, pornind de la numărul Fibonacci **8** (cel de-al 9-lea din șir):
 - Secvența f_1 va conține un număr de monotonii egal cu suma a $n-1=5$ numere Fibonacci consecutive: $1 + 1 + 2 + 4 + 8 = 16$.
 - Secvența f_2 va conține un număr de monotonii egal cu suma a $n-2=4$ numere Fibonacci consecutive: $1 + 2 + 4 + 8 = 15$.
 - Secvența f_3 , o sumă de $n-3=3$ numere: $2 + 4 + 8 = 14$.
 - Secvența f_4 , o sumă de $n-4=2$ numere: $4 + 8 = 12$.
 - Secvența f_5 , o sumă de $n-5=1$ numere Fibonacci de ordinul 4, adică 8 monotonii.
- În total **secvența inițială** care urmează să fie sortată trebuie să conțină :

$$16 + 15 + 14 + 12 + 8 = 65 \text{ monotonii}$$

- Aceste monotonii se distribuie inițial pe cele 5 secvențe sursă în concordanță cu numerele determinate anterior realizându-se astfel o **distribuție perfectă**.
- Se observă simplu că în fig. 3.3.4.c, **distribuția monotoniiilor** pentru **fiecare nivel** k se obține aplicând același algoritm, alegând ca bază, numere consecutive din șirul Fibonacci 1, 1, 2, 4, 8, ... pentru respectiv $k = 1, 2, 3, 4, 5, \dots$.
- Dacă numărul inițial de monotonii **nu** satisface condiția de a fi o astfel de sumă de $n-1$ sume Fibonacci, se **simulează** un număr de **monotonii ipotetice vide**, astfel încât **suma** să devină **perfectă**.

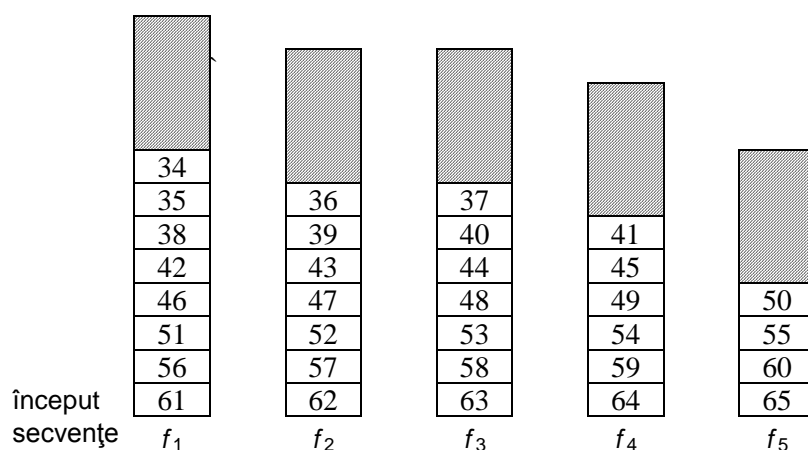
- Aceste monotonii se numesc "**fictive**".
- Problema care se pune se referă la maniera în care aceste monotonii sunt recunoscute respectiv procesate.
- Pentru început se va investiga problema **distribuției inițiale a monotonilor**, iar apoi cea a **distribuției monotonilor fictive**.
- Este evident faptul că selecția unei **monotonii fictive** de pe secvența i înseamnă că secvența respectivă este ignorată în timpul interclasării curente, rezultând o interclasare de pe mai puțin de $n-1$ secvențe sursă.
 - Interclasarea unei **monotonii fictive** de pe toate cele $n-1$ secvențe sursă **nu** conduce la nici o interclasare efectivă ci doar la înregistrarea unei monotonii fictive pe secvența de ieșire.
 - Din acest motiv, este necesar ca monotonii fictive să fie **distribuite** cât mai uniform posibil pe cele $n-1$ secvențe.
- În primul rând se va analiza **problema repartizării unui număr necunoscut de monotonii** pe $n-1$ secvențe în vederea obținerii **distribuției perfecte**.
 - Este clar că numerele Fibonacci de ordinul $n-2$ care reprezintă numărul dorit de monotonii pot fi generate în mod progresiv.
- Astfel, în cazul $n=6$, având ca reper tabelul din fig. 3.3.4.c:
 - Se pornește cu distribuția corespunzătoare lui $k=1$ $(1, 1, 1, 1, 1)$.
 - Dacă există mai multe monotonii se trece la rândul următor $(2, 2, 2, 2, 1)$, adică $k=2$.
 - Apoi la $(4, 4, 4, 3, 2)$, adică $k=3$, ș.a.m.d.
- Indexul k al rândului se numește **nivel**.
 - Pe măsură ce crește numărul monotonilor, crește și nivelul numerelor Fibonacci, nivel care în același timp precizează și **numărul de treceri** sau comutări de secvențe necesar pentru sortarea respectivă.

k	$a_1^{(k)}$	$a_2^{(k)}$	$a_3^{(k)}$	$a_4^{(k)}$	$a_5^{(k)}$	$\sum a_i^{(k)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Fig. 3.3.4.c. Distribuția perfectă a monotonilor pe 6 secvențe (reluare)

- **Algoritmul de distribuție** poate fi formulat astfel:

- (1) Fie scopul distribuției, numerele lui Fibonacci de ordin $n-2$ nivelul 1.
 - (2) Se realizează distribuția monotonilor conform acestui scop.
 - (3) Odată scopul atins, se calculează **nivelul următor** de numere Fibonacci. **Diferența** dintre acestea și numerele corespunzătoare ale nivelului anterior constituie noul **scop** al distribuției.
 - (4) Se revine la pasul 2.
 - (5) Algoritmul se termină la epuizarea monotonilor.
- Regulile după care se calculează următorul nivel al numerelor lui Fibonacci se bazează pe definiția acestora [3.3.4.e].
 - Atenția noastră se va concentra asupra pasului 2, în care, în conformitate cu scopul curent se vor **distribui** monotoniile corespunzătoare una după alta pe cele $n-1$ secvențe.
 - În acest pas apar din nou în discuție **monotoniile fictive**.
 - Se presupune că la trecerea la nivelul următor, **scopul** următor va fi înregistrat prin diferențele d_i pentru $i=1, 2, \dots, n-1$, unde d_i precizează numărul de monotonii care trebuie depus pe secvența i în acest pas.
 - Se **presupune** în continuare că pe fiecare secvență se pun inițial **doar monotonii fictive**, respectiv d_i **monotonii** pe secvența i , unde d_i este diferența calculată anterior.
 - În consecință, distribuirea **monotonilor reale** poate fi privită ca și o **reînlocuire a monotonilor fictive** cu **monotonii actuale**, de fiecare dată înregistrând o înlocuire prin scăderea lui 1 din d_i .
 - Când sursa de **monotonii actuale** se epuizează, valoarea curentă a lui d_i indică chiar **numărul de monotonii fictive** de pe secvența i .
 - Nu se cunoaște algoritmul care conduce la **distribuția optimă**, dar cel propus de **Knuth**, numit **distribuție orizontală** este foarte bun [Kn76].
 - Termenul de **distribuție orizontală** poate fi înțeles considerând monotoniile clădite în forma unor silozuri.
 - În figura 3.3.4.d apar reprezentate aceste silozuri de monotonii pentru $n=6$, nivelul 5, conform fig. 3.3.4.c.



scopul d[1]=8 d[2]=7 d[3]=7 d[4]=6 d[5]=4

Fig.3.3.4.d. Distribuția orizontală a monotoniilor

- Pentru a ajunge cât mai rapid la o distribuție egală a monotoniilor fictive, se procedează la **înlocuirea** lor cu monotonii actuale.
 - Acest proces reduce dimensiunea silozurilor prin **extragerea monotoniilor fictive** din nivelurile orizontale și **înlocuirea lor cu monotonii reale**, de la **stânga la dreapta**.
 - În acest fel monotoniile sunt distribuite pe secvențe după cum indică numărul lor de ordine fig. 3.3.4.d.
 - Se precizează că în această figură este reprezentată ordinea de distribuție a monotoniilor când se trece de la **nivelul 4** (k=4) conținând 33 de monotonii la **nivelul 5** (k=5) conținând 65 de monotonii.
 - Suprafețele hașurate reprezintă primele 33 de monotonii care au fost deja distribuite când s-a ajuns la nivelul 4.
 - Dacă spre exemplu ar fi numai 53 de monotonii inițiale, toate monotoniile numerotate cu 54 și mai mult vor fi tratate ca fictive.
 - Monotoniile se înscriu în realitate la sfârșitul secvenței, dar este mai avantajos să ne imaginăm că se scriu la început, deoarece în procesul sortării se presupune că monotoniile fictive sunt la începutul secvenței.

3.3.4.1 Implementarea algoritmului sortării polifazice

- Pentru început se abordează descrierea procedurii **SelecteazaSecventa** care este apelată de fiecare dată când trebuie copiată o nouă monotonie.
- Procedura realizează selecția **noii secvențe** pe care se va copia următoarea monotonie ținând cont și de **distribuția perfectă** a monotoniilor pe secvențele sursă.
 - Se presupune că variabila j este indexul secvenței curente de destinație. Se utilizează tipurile de date definite în [3.3.4.g]:

```
-----  
/*Sortarea polifazică - structuri de date*/  
  
/*definire structură fișier*/  
typedef struct [3.3.4.g]  
{  
    char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];  
    FILE* file;  
} tip_fisier;  
  
/*definire structură secvență*/  
typedef struct tip_secventa  
{  
    tip_fisier fisier; /*fișier suport secvență*/  
    tip_element elemCurent; /*tampon element secvență*/
```



```

        boolean termPrelucr;          /*indicator terminare
                                         prelucrare secvență*/
} tip_secventa;

int n;                                /*număr de secvențe*/
tip_secventa F[n+1];                 /*tablou de secvente*/
int a[n+1];                           /*tablou cu numărul ideal de monotonii*/
int d[n+1];                           /*tablou cu numărul de monotonii fictive*/
int j;                                /*indexul secvenței destinație curente*/
int nivel;                            /*nivelul curent de distribuție*/
-----

```

- Tablourile a și d memorează **numerele de distribuții ideale** respectiv **fictive** corespunzătoare fiecărei secvențe i .
 - Aceste tablouri se inițializează cu valorile $a_i=1$, $d_i=1$ pentru $i = 1, \dots, n-1$ respectiv $a_n=0$, $d_n=0$.
 - Variabilele j și nivel se inițializează cu valoarea 1.
- Procedura **SelecteazaSecventa** va calcula de fiecare dată când crește nivelul, valorile **rândului următor** al tabelului din figura 3.3.4.c, respectiv valorile $a_1^{(k)}, \dots, a_{n-1}^{(k)}$.
 - Tot atunci se calculează și diferențele $d_i=a_i^{(k)}-a_i^{(k-1)}$, care reprezintă **scopul următor**.
 - Algoritmul se bazează pe faptul că valorile d_i descresc odată cu creșterea indicilor (vezi fig. 3.3.4.d).
 - Se precizează că algoritmul începe cu nivelul 1 (nu cu nivelul 0).
 - Procedura se termină cu decrementarea cu o unitate a lui d_j ceea ce corespunde înlocuirii unei monotonii fictive cu una reală pe secvența j [3.3.4.h].

```

-----
/*Sortarea polifazică - procedura SelecteazăSecvența*/

```

PROCEDURE SelecteazaSecventa

[3.3.4.g.]

```

int i;
int z;

daca (d[j]<d[j+1]) j=j+1;
altfel
  daca (d[j]=0)
    nivel=nivel+1; z=a[1];
    pentru i=1 la n-1
      d[i]=z+a[i+1]-a[i]
      a[i]=z+a[i+1]
    □ /*pentru*/
  □ /*daca*/
  j=1
□ /*altfel*/
d[j]=d[j]-1
/*SelecteazaSecventa*/
-----

```

- Presupunând că se dispune de o rutină de copiere a unei monotonii de pe secvența sursă `f0` pe secvența `F[j]`, faza inițială de distribuire a monotoniilor poate fi schițată astfel [3.3.4.i]:

```
/*Sortarea polifazică - faza inițială de distribuție a monotoniilor - pasul de rafinare 0*/
```

```

repetă
    SelecteazăSecvența;                                [3.3.4.i]
    CopiazăMonotonia
pana când (f0.termPrelucr)
□ /*repetă*/

```

- Spre deosebire de interclasarea naturală, în cea polifazică este necesar să se cunoască **numărul exact de monotonii** de pe fiecare secvență, deoarece procesul de interclasare poate conduce la diminuarea numărului de monotonii.
- Pentru aceasta se va reține cheia ultimului element al ultimei monotonii de pe fiecare secvență.
 - În acest scop se introduce variabila tablou `ultim[NrSecventa]` cu elemente de `TipCheie`.
- Următoarea rafinare a algoritmului de distribuție este cea din [3.3.4.j]:

```
/*Sortarea polifazică - faza inițială de distribuție a monotoniilor - pasul de rafinare 1*/
```

```

    tip_cheie ultim[n+1]; /*reține cheia ultimului element al
                           ultimei monotonii de pe fiecare secvență*/
repetă                                                    [3.3.4.j]
    SelecteazăSecvența;
    dacă (ultim[j] <= f0.curent.cheie)
        *continuă vechea monotonie;
    CopiazăMonotonia;
    ultim[j] = f0.curent.cheie;
pana când (f0.termPrelucr)
□ /*repetă*/

```

- O problemă evidentă este aceea că `ultim[j]` se poziționează numai după copierea primei monotonii, motiv pentru care la început distribuția monotoniilor trebuie realizată fără inspectarea lui `ultim[j]`.
- Restul monotoniilor se distribuie conform secvenței [3.3.4.k]. Se consideră că asignarea lui `ultim[j]` se realizează în procedura **CopiazăMonotonia**.

```
/*Sortarea polifazică - procedura Copiază monotonia*/
```

```

while (NOT f0.termPrelucr)                                [3.4.4.k]
    SelecteazăSecvența;
    if (ultim[j] <= f0.curent.cheie)
        /*continuă vechea monotonie*/
        CopiazăMonotonia;
        if (f0.termPrelucr)

```

```

        d[j] = d[j] + 1;
    else
        CopiazaMonotonia;
    □ /*if*/
else
    CopiazaMonotonia;
□ /*while*/

```

- **Structura algoritmului** de sortare polifazică este în mare parte similară sortării bazate pe interclasare echilibrată cu n căi și include:
 - O buclă exterioară care **interclasează monotoniile** și se execută până când se epuizează sursele.
 - O buclă cuprinsă în cea anterioară care **interclasează o singură monotonie** de pe fiecare secvență sursă.
 - Și în sfârșit o a treia buclă cuprinsă în precedentă, care **selectează cheile** și transmite elementele implicate spre secvența destinație.
- Se remarcă următoarele **diferențe**:
 - Avem o singură secvență destinație (în loc de $N/2$).
 - În loc de a comuta $N/2$ secvențe la fiecare trecere, acestea sunt rotite, utilizând un tablou τ al indicilor secvențelor.
 - Numărul de secvențe de intrare variază de la monotonie la monotonie; acest număr este determinat la începutul sortării fiecărei monotonii din valorile d_i ale monotoniilor fictive.
 - Dacă $d_i > 0$, pentru toți i , atunci vor fi "*pseudo-interclasate*" $n-1$ monotonii fictive într-o monotonie fictivă, ceea ce presupune doar incrementarea contorului d_n al secvenței destinație.
 - Altfel, se interclasează câte o monotonie de pe toate secvențele care au $d_i = 0$, iar pentru restul secvențelor, d_i se decrementează indicând faptul că a fost luată în considerare o monotonie fictivă.
 - Se notează cu k_1 numărul secvențelor implicate într-o interclasare.
 - Din cauza monotoniilor fictive, terminarea unei faze **nu** poate fi dedusă din condiția de sfârșit a vreuneia din secvențele implicate, deoarece existența unor monotonii fictive pe această secvență face necesară continuarea interclasării.
 - În schimb se poate determina ușor numărul teoretic de monotonii din coeficienții a_i .
 - Acești coeficienți $a_i^{(k)}$ care au fost calculați în timpul fazei de distribuție vor fi **recalculați** în sens invers (backward) la interclasare.
- Cu aceste observații interclasarea propriu-zisă se poate formula după cum urmează [3.3.4.1]. Se presupune că:
 - Toate cele $n - 1$ secvențe care conțin monotoniile inițiale sunt resetate.
 - Tabloul indicilor secvențelor este poziționat conform relației $\tau[i] = i$.

```

/*Sortarea polifazică - faza de interclasare*/

```

```

/*[3.3.4.1]*/

```

```

repeat /*interclasare de pe F[t[1]],...,F[t[n-1]] pe F[t[n]]*/
    z=a[n-1]; d[n]=0; Rewrite(F[t[n]]);
    repeat /*interclasarea unei monotonii*/
        k1=0;
        /*se determină nr k1 al secvențelor de intrare active*/
        for t=1 to n-1
            if(d[i]>0)
                d[i]=d[i]-1;
            else
                k1=k1+1;
                td[k1]=t[i];
                □ /*else*/
        if(k1=0)
            d[n]=d[n]+1;
        else
            *se interclasează câte o monotonie de pe
              F[t[1]],...,F[t[k1]];
            z=z-1
        until (z=0)
        □
    Reset(F[t[n]]);
    *rotește secvențele în tabloul t. Calculează a[i]
      pentru nivelul următor;
    Rewrite(F[t[n]]);
    nivel= nivel-1;
until (nivel=0)
□
/*elementele sortate sunt pe F[t[1]]*/
-----

```

- Programul de **sortare polifazică** este similar cu cel de sortare prin interclasare cu n căi, cu diferența că algoritmul de eliminare al secvențelor este mai simplu.
- Rotirea indicilor secvențelor în tabloul indicilor, a indicilor d_i corespunzători, precum și calcularea valorilor coeficienților a_i , sunt rafinate în programul următor [3.3.4.m].

```

/*Sortarea polifazică - varianta finală*/
/*[3.3.4.m]*/

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

#define true (1)
#define false (0)
#define LUNGIME_MAXIMA_NUME_FISIER (256)

typedef int boolean;
typedef int tip_element;

//definire structură fișier
typedef struct
{
    char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
    FILE* file;
} tip_fisier;

//definire structură secvență
typedef struct tip_secventa
{
    tip_fisier fisier;
    tip_element elemCurent;
    boolean termPrelucr;
}

```

```

} tip_secventa;

//definire operatori prelucrare secvențe
void Rewrite(tip_secventa* secventa)
{
    fclose(secventa->fisier.file);
    secventa->fisier.file = fopen(secventa->fisier.num_fisier, "w+");
}

void Read(tip_fisier* fisier, tip_element* elem_curent)
{
    fscanf(fisier->file, "%d ", elem_curent);
}

void Write(tip_fisier* fisier, tip_element buf)
{
    fprintf(fisier->file, "%d ", buf);
}

void Reset(tip_secventa* secventa)
{
    fseek(secventa->fisier.file, 0, SEEK_SET);
    secventa->termPrelucr = false;
}

boolean Eof(tip_fisier* fisier)
{
    int current_position = ftell(fisier->file);
    fseek(fisier->file, 0, SEEK_END);
    int size = ftell(fisier->file);
    fseek(fisier->file, current_position, SEEK_SET);
    return !(fisier->file) || size == current_position;
}

void Close(tip_fisier fisier)
{
    fclose(fisier.file);
}

//definire generator de numere aleatoare pentru generarea lui f0
void generare_numere_aleatoare(const char* num_fisier, int dimensiune)
{
    FILE* destination = fopen(num_fisier, "w");
    if (!destination)
    {
        printf("\nEroare la deschidere\n");
        exit(1);
    }

    int aleat = 7789;
    do
    {
        aleat = (131071 * aleat) % 2147483647;
        int temp = aleat / 214784;

        fprintf(destination, "%d ", temp);
        dimensiune--;
    }
    while (dimensiune > 0);

    fclose(destination);
}

void InitializareUnFisier(tip_secventa* secventa, char* num_fisier)
{
    secventa->termPrelucr = false;
    strcpy(secventa->fisier.num_fisier, num_fisier);
    secventa->fisier.file = fopen(num_fisier, "w+");
}

```

```

//definire structuri de date specifice
#define n 3 //număr de secvențe
tip_secventa F[n + 1]; //tablou de secvențe
int a[n + 1]; //tablou cu numărul ideal de monotonii
int d[n + 1]; //tablou cu numărul de monotonii fictive
int j = 1; //indexul secvenței destinație curente
int nivel = 1; //atât j, cât și nivel se inițializează cu valoarea 1
boolean eob;
tip_element buf;
tip_secventa f0; //secvența inițială f0

int t[n + 1]; //tablou de mapping pentru secvențe
int ultim[n + 1]; //tablou care memorează cheia ultimului element al
//ultimei monotonii de pe fiecare secvență*/
int td[n + 1]; //tablou de mapping pentru secvențele de lucru
int i;
int mx;
int tn;
int k1;
int dn;
int x;
int min;
int z;

void initializareSecvente() //inițializează fișierele destinate procesului de
//interclasare
{
    char sir[256];
    for(i = 1; i <= n; ++i)
    {
        printf("nume fisier secventa %d ", i);
        scanf("%s", sir);
        InitializareUnFisier(&F[i], sir);
    }
}

void initializare() //inițializează tablourile a și d
{
    for(i = 1; i < n; ++i) {
        a[i] = 1; d[i] = 1; Rewrite(F[i].fisier);
    }
    a[n] = 0;
    d[n] = 0;
}

void selecteazaSecventa() //alege secvența de pe care se copiază monotonia
{
    if(d[j] < d[j + 1])
        ++j;
    else
    {
        if(d[j] == 0)
        {
            ++nivel;
            z = a[1];
            for(i = 1; i < n; ++i)
            {
                d[i] = z + a[i+1] - a[i];
                a[i] = z + a[i+1];
            }
        }
        j = 1;
    }
    --d[j];
}

void list(tip_secventa f, int N) //afisează conținutul secvenței
{
    Reset(&f);
    printf("secventa %d\n", N);
    while(!Eof(&f.fisier))

```

```

    {
        Read(&f.fisier,&buf);
        printf("%d ",buf);
    }
    printf("\n");
    Reset(&f);
}

void copiazaMonotonia()
{
    if(Eof(&f0.fisier))
        return;
    if(!ftell(f0.fisier.file))
        Read(&f0.fisier,&f0.elemCurent);
    do{
        buf = f0.elemCurent;
        Write(&F[j].fisier,buf);
        if(Eof(&f0.fisier))
            f0.termPrelucr = true;
        else
            Read(&f0.fisier,&f0.elemCurent);
    }while(buf <= f0.elemCurent && !f0.termPrelucr);
    ultim[j] = buf;
}

void distribuireMonotonii() //se pregătesc secvențele pentru distribuția
                             monotoniiilor inițiale
{
    for(i = 1;i < n;++i)
    {
        a[i] = 1;
        d[i] = 1;
        Rewrite(&F[i]);
    }
    nivel = 1;
    j = 1;
    a[n] = 0;
    d[n] = 0;
}

void copiazaMonotoniaR() //se realizează distribuția (aproape) optimă
{
    do{
        selecteazaSecventa();
        copiazaMonotonia();
    }while(!f0.termPrelucr && j != n - 1);
    while(!f0.termPrelucr)
    {
        selecteazaSecventa();
        if(ultim[j]<=f0.elemCurent)
        {
            copiazaMonotonia();
            if(f0.termPrelucr)
                ++d[j];
            else
                copiazaMonotonia();
        }
        else
            copiazaMonotonia();
    }
}

void interclasare() //interclasare de pe F[t[1]],...,F[t[n-1]] pe F[t[n]]
{
    do{
        z = a[n-1];
        d[n] = 0;
        Rewrite(&F[t[n]]);
        //interclasarea unei monotonii
        do{

```

```

    k1 = 0;
    //se determin num[rul k1 al secvențelor de intrare active
    for(i = 1; i < n; ++i)
        if(d[i] > 0)
            --d[i];
        else
        {
            ++k1;
            td[k1] = t[i];
        }
    if(!k1)
        ++d[n];
    else
    {
        //se interclasează o monotonie de pe F[t[1]],...,F[t[k1]] pe F[t[n]]
        do{
            i = 1;
            mx = 1;
            min = F[td[1]].elemCurent;
            while(i < k1){
                ++i;
                x = F[td[i]].elemCurent;
                if(x < min)
                {
                    min = x;
                    mx = i;
                }
            }
            //td[mx] conține elementul minim; se mută pe t[n]
            buf = F[td[mx]].elemCurent;
            if(Eof(&F[td[mx]].fisier))
                F[td[mx]].termPrelucr = true;
            else
                Read(&F[td[mx]].fisier,&F[td[mx]].elemCurent);
            Write(&F[t[n]].fisier,buf);
            if(buf > F[td[mx]].elemCurent || F[t[mx]].termPrelucr)
            {
                td[mx] = td[k1];
                --k1;
            }
        }while(k1);
    }
    --z;
}while(z);

Reset(f[t[n]]);
list(F[t[n]],t[n]);
//rotire secvente
tn = t[n];
dn = d[n];
z = a[n-1];
for(i = n; i > 1; --i)
{
    t[i] = t[i-1];
    d[i] = d[i-1];
    a[i] = a[i-1]-z;
}

t[1] = tn;
d[1] = dn;
a[1] = z;
//elementele sortate sunt pe t[1]
list(F[t[1]],t[1]);
--nivel;
}while(nivel);
}

void alg()//sortarea polifazica
{
    distribuireMonotonii();
    copiazaMonotoniaR();
}

```



```

    for(i = 1; i <= n; ++i)
    {
        Reset(&F[i]);
        Read(&F[i].fisier, &F[i].elemCurent);
    }
    for(i = 1; i <= n; ++i)
        t[i] = i;
    interclasare();
}

int main()
{
    InitializareUnFisier(&f0, "fis.in");
    initializareSecvente();
    generare_numere_aleatoare("fis.in", 7); //exemplu
    alg(); //ultima secvență va conține rezultatul final
    return 0;
}
-----

```

3.3.5. Concluzii

- Complexitatea metodelor de sortare externă prezentate nu permite formularea unor concluzii generalizatoare, cu atât mai mult cu cât evidențierea performanțelor acestora este dificilă.
- Se pot formula însă următoarele **observații**:
 - (1) Există o **legătură indisolubilă** între un **algoritm** care rezolvă o anumită problemă și **structurile de date** pe care acesta le utilizează, influența celor din urmă fiind uneori decisivă pentru algoritm.
 - Acest lucru este evidențiat cu preponderență în cazul sortărilor externe care sunt complet diferite ca mod de abordare în raport cu metodele de sortare internă.
 - (2) În general, **îmbunătățirea performanțelor unui algoritm** presupune elemente foarte sofisticate, chiar în condițiile utilizării unor structuri de date dedicate.
 - În mod paradoxal algoritmii cu performanțe superioare sunt mai complicați, ocupă mai multă memorie și utilizează de regulă structuri de date specializate.