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) Evidenţierea **influenţei** profunde pe care adoptarea unei anumite **structuri** o are asupra **algoritmului** care o utilizează şi asupra **tehnicilor de programare** care implementează algoritmul respectiv.
- **Sortarea** este domeniul ideal al **studiului:**
  - (1) **Construcţiei** algoritmilor.
  - (2) **Performanţelor** algoritmilor.
  - (3) **Avantajelor** şi **dezavantajelor** unor algoritmi faţă de alţii în accepţiunea unei aplicaţii concrete.
  - (4) **Tehnicilor de programare** aferente diferiţilor algoritmi.
- Prin **sortare** se înţelege în general **ordonarea** unei mulţimi 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 dicţionare, în depozitele de mărfuri şi în general în orice situaţie în care trebuiesc căutate şi 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;                             /*3.1.a*/
} tip_element;
```

- Câmpul `cheie` precizat, poate fi neesenţial din punctul de vedere al informaţiei înregistrate în articol, partea esenţială a informaţiei fiind conţinută în celelalte câmpuri.

- Din punctul de vedere al **sortării** însă, `cheie` este cel mai important câmp întrucât este valabilă următoarea **definiţie a sortării**.

  - Fiind dat un şir de elemente aparţinând tipul mai sus definit:

    $a_1,\ a_2, \ldots., a_n$

  - Prin **sortare** se înţelege **permutarea** elementelor şirului într-o anumită ordine:

    $a_{k1},\ a_{k2}, \ldots.., a_{kn}$

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

    $a_{k1}.cheie \le a_{k2}.cheie \le \ldots \le a_{kn}.cheie$

- Tipul câmpului `cheie` se presupune a fi **întreg** pentru o înţelegere 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 iniţială

  - Această stabilitate este esenţială în special în cazul în care se execută sortarea după mai multe chei.

- În cazul **operaţiei de sortare**, **dependenţa** dintre **algoritmul** care realizează sortarea şi **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 şi **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 fişierelor** (secvenţelor) numită şi **sortare externă.**

3.2. Sortarea tablourilor

- Tablourile se înregistrează în **memoria centrală** a sistemelor de calcul, motiv pentru care **sortarea tablourilor** se mai numeşte şi **sortare internă.**

- **Cerinţa fundamentală** care se formulează faţă 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 algoritmii care realizează **sortarea "in situ"**, adică chiar în zona de memorie alocată tabloului.

* Pornind de la această restricţie, în continuare algoritmii vor fi clasificaţi în funcţie de eficienţa lor, respectiv în funcţie de timpul de execuţie pe care îl necesită.

* Aprecierea **cantitativă** a eficienţei unui algoritm de sortare se realizează prin intermediul unor **indicatori specifici**.

    * (1) Un prim indicator este **numărul comparaţiilor 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 mişcări de elemente executate de algoritm, notat cu **M**.

        * Ambii indicatori depind de numărul total `n` al elementelor care trebuiesc sortate.

* În cazul unor algoritmi de sortare simpli bazaţi pe aşa-zisele **metode directe de sortare** atât C cât şi M sunt proporţionali cu n^2 adică sunt O(n^2).

* Există însă şi **metode avansate de sortare**, care au o complexitate mult mai mare şi în cazul cărora indicatorii C şi M sunt de ordinul lui $n*log_2 n$ (O$(n*log_2 n)$).

    * Raportul $n^2/(n*log_2 n)$, care ilustrează câştigul de eficienţă realizat de aceşti algoritmi, este aproximativ egal cu 10 pentru $n = 64$, respectiv 100 pentru n = 1000.

* Cu toate că ameliorarea este substanţială, **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 şi relativ uşor de înţeles.

    * (3) Deşi metodele avansate necesită mai puţine operaţii, aceste operaţii 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 **inserţie.**

    * (2) Sortarea prin **selecţie.**

    * (3) Sortarea prin **interschimbare**.

* În prezentarea acestor metode se va lucra cu tipul element definit anterior, precum şi cu următoarele notaţii [3.2.a].

```
------------------------------------------------------------
#define n  ...
typedef struct tip_element {
```

```
    int cheie;                                         /*3.2.a*/
    ... alte câmpuri;
} tip_element;

tip_element a[n];
```

3.2.1. Sortarea prin inserţie

- Această metodă este larg utilizată de jucătorii de cărţi.
  - Elementele (cărţile) sunt în mod conceptual divizate într-o secvenţă **destinaţie** $a_1...a_{i-1}$ şi într-o secvenţă **sursă** $a_i....a_n$.
  - În fiecare pas începând cu `i = 2`, elementul `i` al tabloului (care este de fapt primul element al secvenţei sursă), este luat şi transferat în secvenţa destinaţie prin **inserarea** sa la locul potrivit.
  - Se incrementează `i` şi se reia ciclul.
- Astfel la început se sortează primele două elemente, apoi primele trei elemente şi aşa 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 secvenţă deja sortată.
- În termeni formali, acest algoritm este precizat în secvenţa [3.2.1.a].

```
/*Sortarea prin inserţie*/
 tip_element a[n];

pentru i=2 to n                                       /*3.2.1.a*/
  *inserează a[i] la locul potrivit în a[1]...a[i];
□
 /*pentru*/}
```

- **Selectarea** locului în care trebuie inserat `a[i]` se face parcurgând **secvenţa destinaţie** deja sortată `a[1],...,a[i-1]` de la dreapta la stânga şi comparând pe `a[i]` cu elementele secvenţei.
  - **Simultan** cu parcurgerea, se realizează şi **deplasarea spre dreapta** cu o poziţie a fiecărui element testat până în momentul îndeplinirii condiţiei 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 poziţie.

- Acest caz tipic de repetiţie cu **două condiţii de terminare** readuce în atenţie **metoda fanionului** (&1.4.2.1).

  - Pentru aplicarea ei se introduce **elementul auxiliar** `a[0]` care se asignează iniţial cu `a[i]`.

  - În felul acesta, cel mai târziu pentru `j=0`, condiţia de a avea cheia lui `a[j]` "mai mică sau egală" cu cheia lui `a[i]` se găseşte îndeplinită şi nu mai trebuie verificată valoarea indicelui `j(>=0)`.

  - Inserţia propriu-zisă se realizează pe poziţia `j+1`.

- Algoritmul care implementează sortarea prin inserţie apare în [3.2.1.b] iar profilul său temporal în figura 3.2.1.a.

```
------------------------------------------------------------
/*Sortare prin inserţie*/

void SortarePrinInsertie                              /*3.2.1.b*/
{ /*global: tip_element a[n]*/
  int i,j;
  tip_element temp;
  for(i=2;i<n;i++)
    {
      temp=a[i]; a[0]=a[i];    /*poziţionare fanion*/
      j=i-1;
      while(a[j].key>temp.key)
        {
          a[j+1]=a[j];  j=j-1;
        } /*while*/
      a[j+1]=temp;             /*inserţie element*/
    } /*for*/
} {SortarePrinInsertie}
------------------------------------------------------------
```

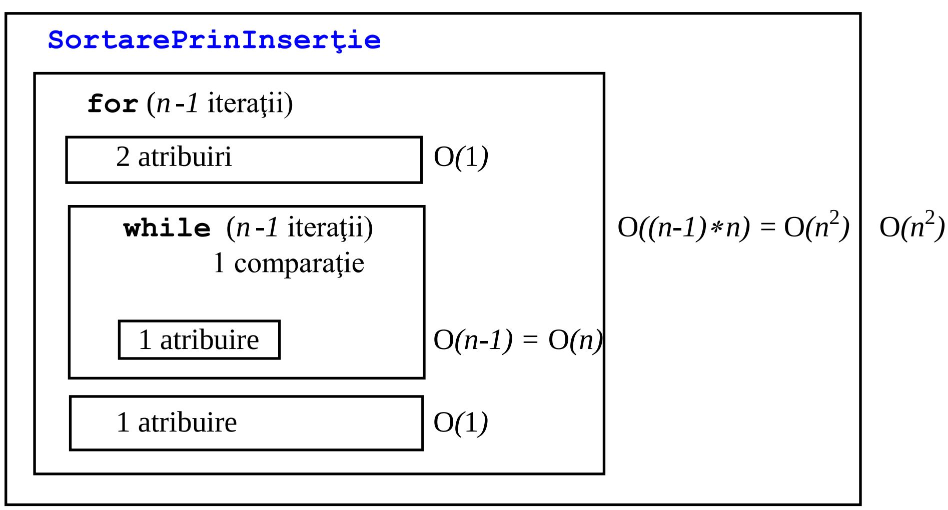


Fig.3.2.1.a. Profilul temporal al algoritmului de sortare prin inserţie

- După cum se observă, algoritmul de sortare conţine un **ciclu exterior** după `i` care se reia de `n-1` ori (bucla **`for`**).

- În cadrul fiecărui ciclu exterior se execută un **ciclu interior** de lungime variabilă după `j`, până la îndeplinirea condiţiei (bucla **`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 inserţie

- În cadrul celui de-al i-lea ciclu **`for`**, numărul C_i al **comparaţiilor de chei** executate în bucla **`while`**, depinde de ordinea iniţială a cheilor, fiind:

  - Cel puţin `1` (secvenţa ordonată).

  - Cel mult `i-1` (secvenţa 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..n`, parametrul `C` are valorile precizate în [3.2.1.c].

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

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

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

- Numărul de **atribuiri de elemente** M_i în cadrul unui ciclu **`for`** este $C_i + 3$.

  - **Explicaţia**: la numărul C_i de atribuiri executate în cadrul ciclului interior **`while`** de tip `a[j+1]= a[j]` se mai adaugă 3 atribuiri (`temp= a[i]`, `a[0]= temp` şi `a[i+1]= temp`).

  - Chiar pentru numărul minim de **comparaţii** de chei (C_i egal cu 1) cele trei atribuiri rămân valabile.

- În consecinţă, parametrul M ia următoarele valori [3.2.1.d].

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

$$M_{max} = \sum_{i=2}^{n}(C_i + 3) = \sum_{i=2}^{n}(i+2) = \sum_{i=1}^{n+2} i - (1+2+3) =$$

$$= \frac{(n+2)\cdot(n+3)}{2} - 6 = \frac{n^2 + 5\cdot n - 6}{2} \qquad [3.2.1.d]$$

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

- Se observă că atât C cât şi 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 inserţie este o **sortare stabilă**.

- În secvenţa [3.2.1.e] se prezintă o variantă C uşor modificată a acestei metode de sortare.

```
/*Sortarea prin inserţie - varianta C*/

insertie(int a[],int n)                        /*3.2.1.e*/
{
 for(int i=n-2;i>=0;i--)
  {
    a[n]=a[i];        /*fanionul pe poziţia a[n]*/
    int j=i+1;
    while(a[j]<a[n])
     {
      a[j-1]=a[j]; j++;
     }
    a[j-1]=a[n];      /*inserţie element pe poziţia a[j-1]*/
  }
} /*insertie*/
```

- Relativ la secvenţa [3.2.1.e] se fac următoarele observaţii:

  - Implementarea este varianta "**în oglindă**" faţă de varianta pseudocod.

  - Tabloul de `n` elemente este `a[0]...a[n-1]`.

  - **Secvenţa sursă** este `a[0]...a[i]`.

  - **Secvenţa destinaţie** (cea ordonată) este `a[i+1]...a[n-1]`.

  - **Fanionul** este poziţionat pe poziţia `n` a tabloului `a` in locatia `(a[n])`.

  - În procesul de căutare a locului de inserţie, în pasul curent, se parcurge cu indicele `j` secvenţa destinaţie, de la stânga la dreapta respectiv de la poziţia `i+1` până la găsirea locului inserţiei sau până la `n`.

  - Elementele întâlnite care sunt mai mici ca şi cheia de inserat se mută cu o poziţie spre stânga până la îndeplinirea condiţiei.

3.2.1.2. Sortarea prin inserţie binară

- Algoritmul de **sortare prin inserţie** poate fi îmbunătăţit pornind de la observaţia că **secvenţa destinaţie** este deja **sortată**.

- În acest caz **căutarea locului de inserţie** se poate face mult mai rapid utilizând tehnica **căutării binare**, prin împărţiri succesive în părţi egale a intervalului de căutare.

- Algoritmul modificat se numeşte **inserţie binară** [3.2.1.f].

```
/*Sortare prin inserţie binară - Varianta pseudocod*/

PROCEDURE SortarePrinInserţieBinară        /*[3.2.1.f]*/
{
  tip_element a[n];
  tip_indice i,j,stanga,dreapta,m;
  tip_element temp;
  pentru i=2 la n
    temp=a[i]; stanga=1; dreapta=i-1;
    cât timp (stanga<=dreapta)
      m=(stanga+dreapta)DIV 2;
      if (a[m].cheie>temp.cheie)
          dreapta=m-1;
        else
          stanga=m+1;
    □ /*cât timp*/
    pentru j=i-1 la stanga a[j+1]=a[j];
    a[stanga]=temp;    /*insertie element curent*/
  □ /*pentru*/
} /*SortarePrinInserţieBinară*/
```

```
/*Sortare prin inserţie binară - Varianta C */

void sortare_prin_insertie_binara()        /*[3.2.1.f]*/
{
  tip_element a[n];
  int i,j,stanga,dreapta,m;
  tip_element temp;

  for(i=2;i<=n;i++)
    {
      temp=a[i]; stanga=1; dreapta=i-1;
      while (stanga<=dreapta)
        {
          m=(stanga+dreapta)/2;
          if (a[m].cheie>temp.cheie)
              dreapta=m-1;
            else
              stanga=m+1;
        } /*while*/
      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 inserţie binară

- În cazul **sortării prin inserţie binară**, poziţia de inserat este găsită dacă

  `a[j].cheie ≤ x.cheie ≤ a[j+1].cheie ,`

  adică intervalul de căutare ajunge de dimensiune 1.

- Dacă intervalul iniţial este de **lungime** `i` sunt necesari $\lceil log_2(i) \rceil$ paşi pentru determinarea locului inserţiei.

- Întrucât procesul de sortare presupune parcurgerea prin metoda înjumătăţirii binare a unor secvenţe de lungime `i` (care conţin `i` chei), pentru `i=1,2,...,n`, **numărul total de comparaţii** C efectuate în bucla **`WHILE`** este cel evidenţiat de expresia prezentată în [3.2.1.g].

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

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

$$C = \int_{1}^{n} log_2 x \cdot dx = x \cdot (log_2 x - c) \Big|_1^n = n \cdot (log_2 n - c) + c \qquad [3.2.1.h]$$
$$c = log_2 e = 1/ln\,2 = 1.44269$$

- Numărul **comparaţiilor** de chei este independent de ordinea iniţială 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 comparaţii** şi **nu** asupra **numărului de mişcări**.

- De regulă, mutarea cheilor şi a informaţiilor asociate necesită **mai mult timp** decât **compararea** a două chei.

  - Astfel întrucât M rămâne de ordinul lui n^2, performanţele acestei metode de sortare **nu** cresc în măsura în care ar fi de aşteptat.

  - De fapt, resortarea unui tablou gata sortat, utilizând inserţia binară, consumă mai mult timp decât inserţia cu căutare secvenţială.

- În ultimă instanţă, **sortarea prin inserţie nu** este o metodă potrivită de sortare cu ajutorul calculatorului, deoarece inserţia unui element presupune deplasarea **poziţie cu poziţie** în tablou a unui număr de elemente, deplasare care este neeconomică.

- Acest **dezavantaj** conduce la ideea dezvoltării unor algoritmi în care mişcările să afecteze un număr mai redus de elemente şi să se realizeze pe **distanţe mai mari**.

- Sortarea prin **selecţie** reprezintă un pas înainte în acest sens.

3.2.2. Sortarea prin selecţie

- **Sortarea prin selecţie** foloseşte procedeul de a **selecta** elementul cu cheia minimă şi de a schimba între ele poziţia 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 inserţie** care presupune la fiecare pas **un singur element al secvenţei sursă** şi **toate elementele secvenţei destinaţie** în care se caută de fapt locul de inserţie.

- **Selecţia** în schimb presupune **toate elementele secvenţei sursă** dintre care selectează pe cel cu cheia cea mai mică şi îl depozitează ca **element următor al secvenţei destinaţie.**

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

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

```
------------------------------------------------------------
/* Sortare prin selecţie - varianta C */

void sortare_prin_selectie()                   /*3.2.2.b*/
{
    int i,j,min;
    tip_element temp;
    tip_tablou a[n];

  for(i=0;i<=n-2;i++)
    {
      /*selectează cel mai mic element*/
      min=i; temp=a[i];
      for(j=i+1;j<=n-1;j++)
        if (a[j].cheie<temp.cheie)
          {
            min=j; temp=a[j];
          } /*for*/
      a[min]=a[i]; a[i]=temp; /*interschimbare*/
    } /*for*/
} /*sortare_prin_selectie*/
```

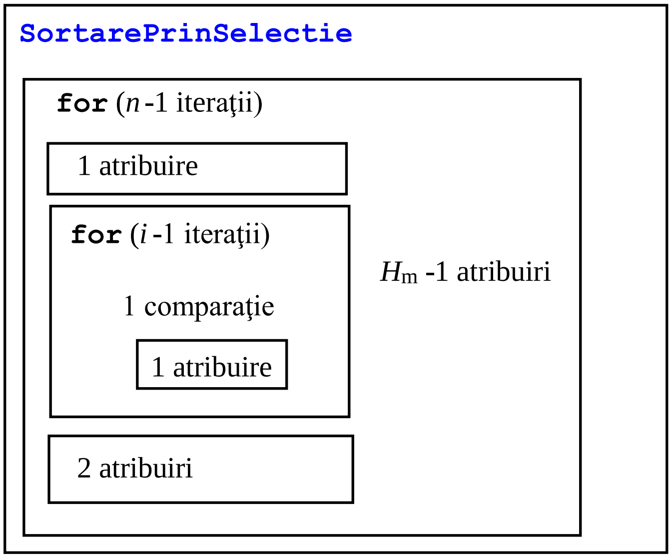


Fig.3.2.2.a. Profilul temporal al sortării prin selecţie

3.2.2.1. Analiza sortării prin selecţie

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

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

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

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

  - Acest minim poate fi atins efectiv, dacă iniţial cheile sunt deja **sortate**.

- Pe de altă parte, dacă cheile sunt **sortate iniţial în ordine inversă**, M_{max} se determină cu ajutorul **formulei empirice** [3.2.2.e] [Wi76].

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

- **Valoarea medie** a lui M **nu** este media aritmetică a lui M_{min} şi M_{max}, ea obţinându-se printr-un **raţionament probabilistic** în felul următor:

  - Se consideră o secvenţă de `m` chei.

  - Fie o **permutare oarecare** a celor `m` chei notată cu $k_1, k_2, ..., k_m$.

  - Se determină numărul termenilor k_j având proprietatea de a fi **mai mici** decât toţi termenii precedenţi $k_1, k_2, ..., k_{j-1}$.

  - Se adună toate valorile obţinute pentru cele `m!` permutări posibile şi se împarte suma la `m!`

  - Se demonstrează că rezultatul acestui calcul este H_m-1, unde H_m este **suma** parţială a **seriei armonice** [3.2.2.f] [Wi76]:

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

- Această valoare reprezintă **media numărului de atribuiri** ale lui `temp`, executate în procesul de sortare pentru o secvenţă de `m` elemente în bucla **`for`** interioară, deoarece `temp` se atribuie ori de câte ori se găseşte un element mai mic decât toate elementele care-l preced.

- Ţinând cont şi de atribuirile `temp=a[i],a[min]=a[i]` şi `a[i]=temp`, **valoarea medie a numărului total de atribuiri** la o trecere pentru `m` termeni este H_m+2.

- Se demonstrează că deşi **seria este divergentă**, o **sumă parţială** a sa se poate calcula cu ajutorul formulei [3.2.2.g]:

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

unde γ = 0.5772156649... este constanta lui **Euler** [Kn76].

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

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

- Tot acest raţionament este valabil la **o trecere** pentru o secvenţă de `m` chei.

* Întrucât în procesul de sortare se parcurg succesiv secvenţe care au respectiv lungimea `m=n,n-1,n-2,...,1,` fiecare dintre ele necesitând în medie H_m+2 atribuiri, numărul mediu total de atribuiri M_{med} va avea expresia:

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

----

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

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

----

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

$$M_{med} \approx n\cdot(\ln m+\gamma+1)+1 = O(n\cdot\ln n) \qquad [3.2.2.k]$$

----

* În concluzie, **algoritmul bazat pe selecţie** este de **preferat** celui bazat **pe inserţie**, cu toate că în cazurile în care cheile sunt ordonate, sau aproape ordonate, sortarea prin inserţie este mai rapidă.

* Optimizarea performanţei sortării prin selecţie poate fi realizată prin reducerea numărului de mişcări 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 reţine doar indicele acestuia, mutarea urmând a se realiza doar pentru ultimul element găsit, la părăsirea ciclului **`for`** interior [3.2.2.l].

```
/*Sortare prin selecţie optimizată - Varianta C*/

void selectie_optimizata()                    /*3.2.2.l*/
{
  tip_element a[n];
  int i,j,min;
  tip_element temp;

  for(i=0;i<=n-2;i++)
    {
      min=i;
      for(j=i+1;j<=n;j++)
        if(a[j].cheie<temp.cheie)   min=j;
      temp=a[min]; a[min]=a[i]; a[i]=temp;
    } /*for*/
```

```
} /*selecţie_optimizată*/
```

- **Măsurătorile experimentale** efectuate însă asupra acestei variante **nu** evidenţiază vreo ameliorare nici chiar pentru valori mari ale dimensiunii tabloului de sortat.

  - Explicaţia rezidă în faptul ca atribuirile care necesită în plus accesul la componenta unui tablou se realizează practic în acelaşi timp ca şi o atribuire normală.

3.2.3. Sortarea prin interschimbare. Sortările bubblesort şi shakersort

- Clasificarea metodelor de sortare în diferite familii ca **inserţie**, **interschimbare** sau **selecţie** nu este întotdeauna foarte bine definită.

  - Paragrafele anterioare au analizat algoritmi care deşi implementează inserţia sau selecţia, 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 şi se interschimbă perechile de elemente alăturate, până când toate elementele sunt sortate.

- Ca şi la celelalte metode, se realizează **treceri repetate** prin tablou, de la capăt spre început, de fiecare dată deplasând cel mai mic element al mulţimii rămase spre capătul din stânga al tabloului.

- Dacă se consideră tabloul în poziţie verticală şi se asimilează elementele sale cu nişte **bule de aer** în interiorul unui **lichid**, fiecare bulă având o **greutate** proporţională 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**.

- Algoritmul aferent acestei metode apare în continuare [3.2.3.a]:

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

PROCEDURE Bubblesort;                          /*[3.2.3.a]*/
{
  tip_element a[n];
  tip_indice i,j:;

  pentru i=1 la n-1
    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],temp;
  int i,j;

  for(i=1;i<n;i++)
    {
      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.a şi el conduce la o estimare a încadrării performanţei algoritmului în ordinul O(n^2).

* Se pot observa trei elemente importante:

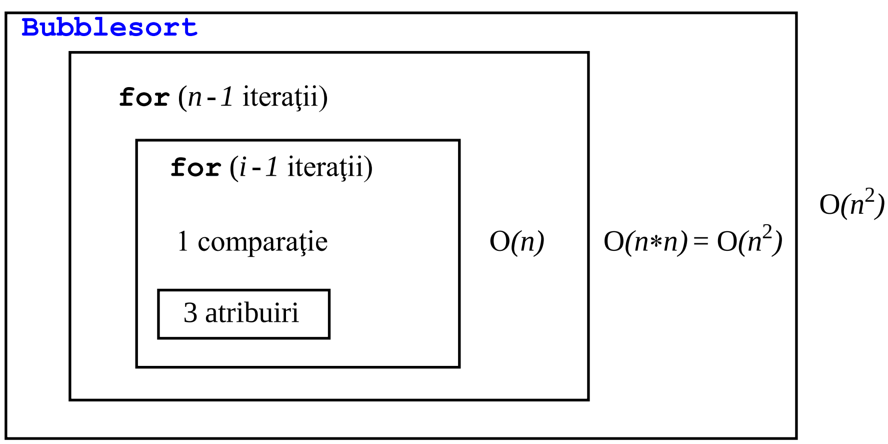


Fig.3.2.a. Profilul temporal al sortării prin interschimbare

* (1) În multe cazuri ordonarea se termină **înainte** de a se parcurge toate iteraţiile buclei `for` exterioare.

    * În acest caz, restul iteraţiilor sunt fără efect, deoarece elementele sunt deja ordonate.

    * În consecinţă, o modalitate evidentă de **îmbunătăţire** a algoritmului bazată pe această observaţie este aceea prin care se memorează **dacă a avut sau nu loc** vreo interschimbare în cursul unei treceri.

    * Şi în acest caz este însă necesară o **ultimă trecere**, fără nici o interschimbare care marchează finalizarea algoritmului.

- O variantă a sortării bubblesort bazată pe observaţia (1) apare în [3.2.3.b]. Această variantă este binecunoscută programatorilor datorită în special simplităţii sale.

```
------------------------------------------------------------
{Sortarea prin interschimbare - Varianta 2 - pseudocod}

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

  tip_element a[n];
  tip_indice i;
  modificat: boolean;

 |repetă
 |  modificat=false;
 | |pentru i=0 la n-2
 | |  dacă (a[i].cheie>a[i+1].cheie)
 | |   |*interschimba a[i] cu a[i+1];
 | |   |modificat=true
 | |   |□ /*daca*/
 | |□ /*pentru*/
 |până când NOT modificat
 |□ /*repeta*/
 /*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];
  tip_element temp;
  int i;
  boolean modificat;

  do {
    modificat=false;
    for(i=0;i<=n-2;i++)
      if (a[i].cheie>a[i+1].cheie)
        {
          temp=a[i]; a[i]=a[i+1]; a[i+1]=temp;
          modificat=true;
        }
  } while(modificat);
}    /*bubblesort1*/
/*-------------------------------------------------------*/
```

- (2) Îmbunătăţirea analizată, poate fi la rândul ei perfecţionată, 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 consecinţă 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 **uşor** plasat la capătul **greu** al tabloului este readus la locul său într-o singură trecere.
    * În schimb un element **greu** plasat la capătul **uşor** al tabloului va fi readus spre locul său doar cu câte o poziţie la fiecare trecere.
    * Spre exemplu tabloul:

      12 18 22 34 65 67 83 04

      va fi sortat cu ajutorul metodei bubblesort îmbunătăţite într-o singură trecere.
    * În schimb ce tabloul:

      83 04 12 18 22 34 65 67

      va necesita şapte treceri în vederea sortării.
    * Această neobişnuită asimetrie, sugerează o a treia îmbunătăţire: **alternarea sensurilor de parcurgere ale trecerilor consecutive**.
* Algoritmul care include aceste îmbunătăţiri se numeşte **shakersort** (sortare prin amestecare) şi este prezentat în [3.2.3.c].

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

void shakersort()                                    /*3.2.3.c*/
{
  tip_element a[n],temp;
  int j,ultim,sus,jos;

  sus=1; jos=n-1; ultim=n-1;
  do {
    for(j=jos;j>=sus;j--)    /*procesare de jos în sus*/
      if (a[j-1].cheie>a[j].cheie)
        {
          temp=a[j-1]; a[j-1]=a[j]; a[j]=temp;
          ultim=j;
        } /*for*/
    sus=ultim+1;             /*actualizare limita sus*/
    for(j=sus;j<=jos;j++)    /*procesare de sus în jos*/
      if (a[j-1].cheie>a[j].cheie)
        {
          temp=a[j-1]; a[j-1]=a[j]; a[j]=temp;
          ultim=j;
        } /*for*/
    jos=ultim-1;              /*actualizare limita jos*/
  } while (!(sus>jos));
} /*shakersort*/
/*---------------------------------------------------------*/
```

3.2.3.1. Analiza sortărilor bubblesort şi shakersort

* **Numărul comparaţiilor** la **algoritmul bubblesort** este constant şi are valoarea:

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

- Valorile minimă, maximă şi medie ale **numărului de mişcări** sunt:

$$M_{min} = 0$$

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

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

- Analiza metodei îmbunătăţite **shakersort** arată că $C_{min} = n-1$.
  - Pentru ceilalţi indicatori, Knuth ajunge la un **număr mediu de treceri** proporţional cu $n - k_1\sqrt{n}$ şi la un **număr mediu de comparaţii** de chei proporţional cu $C_{med} = 1/2(n^2 - n(k_2 + \ln n))$.
- Trebuie însă remarcat faptul că toate îmbunătăţirile 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 îmbunătăţirile atent studiate au un efect **mult mai redus** decât s-ar aştepta în mod intuitiv.
- Analiza comparativă a performanţelor algoritmilor de sortare prezentaţi, scoate în evidenţă următoarele:
  - **Sortarea prin interschimbare** este mai puţin performantă decât sortările prin **inserţie** sau **selecţie**, 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 **distanţa 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 excepţia sortării prin selecţie), fiecare element îşi modifică doar cu un singur loc poziţia la fiecare pas elementar, este necesar un număr de treceri proporţional cu n^2.
- O **îmbunătăţire** efectivă a performanţei trebuie să aibă în vedere deplasarea elementelor pe distanţe mai mari într-un singur pas.

3.2.4. Sortarea prin inserţie cu diminuarea incrementului. Sortarea shellsort

- **D.L. Shell** a propus în 1959, o perfecţionare a metodei de **sortare prin inserţie** directă.

- **Ideea** acestei metode numite **sortare prin inserţie cu diminuarea incrementului** este următoarea:

    - La început, toate articolele care sunt despărţite prin câte **4 poziţii**, sunt grupate şi sortate separat prin metoda inserţiei.

        - Acest proces se numeşte ***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` poziţii.

    - După această primă trecere, sunt grupate elementele sunt separate prin câte **două poziţii** şi din nou sunt sortate prin inserţie.

        - Acest proces se numeşte ***sortare-2***.

    - În final, la cea de-a treia trecere, elementele sunt sortate obişnuit (***sortare-1***).

        - Se precizează faptul că fiecare **k-sortare** este de fapt o **sortare prin inserţie** la care pasul este `k` (nu `1` ca la inserţia normală).

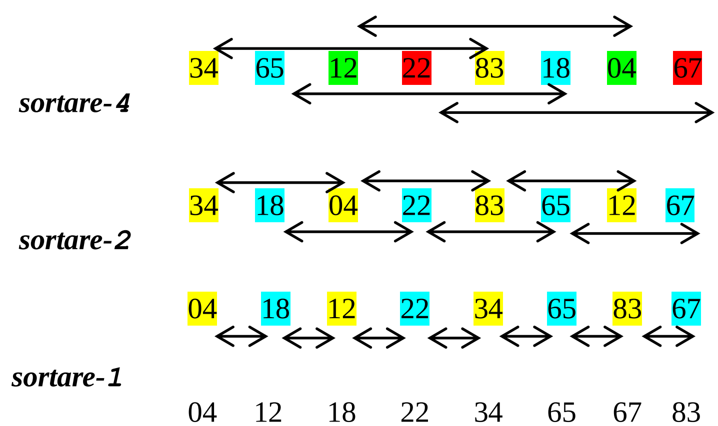


Fig. 3.2.4. Sortare prin inserţie cu diminuarea incrementului

- Deşi la prima vedere această metodă care presupune câteva treceri asupra tuturor elementelor, nu pare foarte performantă, totuşi la o analiză mai atentă, fiecare trecere realizează relativ puţine modificări ale poziţiilor elementelor.

- Este evident că metoda conduce la sortarea elementelor şi că fiecare pas profită de cei anteriori deoarece fiecare **sortare-*i*** combină grupuri deja **sortate-*j***.

- Este posibilă orice secvenţă de incremenţi atâta timp cât ultimul este egal cu unitatea.

    - În cel mai rău caz, procesul de sortare se va realiza în întregime în acest pas.

- Este mai puţin evident, dar practica demonstrează că o secvenţă de incremenţi descrescători care **nu** sunt puteri ale lui 2 asigură o eficienţă 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 cunoaşte nici măcar **secvenţa de incremenţi** cea mai potrivită.
    - Ceea ce este deosebit de interesant este faptul că incremenţii **nu** trebuie să fie unii **multiplii** altora.
- Pentru o eficienţă sporită a sortării este de dorit ca între diferitele lanţuri de parcurgere să aibă loc cât mai multe interacţiuni.
- De asemenea este valabilă următoarea **teoremă** pe care de fapt se bazează metoda:
    - Dacă o secvenţă **sortată-*k*** este **sortată-*i*** ea rămâne şi **sortată-*k***.
    - Cu alte cuvinte, procesul de sortare cu diminuarea incrementului este **cumulativ**.
- **Knuth** indică drept cele mai potrivite secvenţe de incremenţi cele prezentate în [3.2.4.c] respectiv [3.2.4.d] (furnizate în ordine crescătoare):

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

$h_t, h_{t-1}, ..., h_k, h_{k-1}, ..., h_1$ [3.2.4.c]

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

1, 3, 7, 15, 31, ... [3.2.4.d]

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

- Pentru ultima secvenţă, analiza matematică a sortării a `n` elemente cu **metoda Shellsort** demonstrează necesitatea unui efort proporţional cu $n^{1.2}$ ($O(n^{1.2})$).
- În [3.2.4.e] se prezintă o variantă de implementare acestei metode.
- Algoritmul utilizează incremenţi bazaţi pe formula [3.2.4.c], unde incrementul maxim se calculează funcţie de dimensiunea tabloului în prima buclă **`do-while`** [Se88].
    - Incremenţii descrescători se calculează automat la reluarea fiecărui ciclu de sortare, respectiv a doua buclă **`do-while`**.

* Avem de fapt o suită de **sortări prin inserţie** cu incrementul variabil descrescător h.

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

void shellsort()                                 /*3.2.4.e*/
{
  int i,j,h;
  tip_element a[n],temp;

  /*calcul increment maxim*/
  h=1;
  do {
       h=3*h+1;
     } while (!(h>n));
  /*sortare shellsort*/
  do {
    h=h/3;              /*calcul increment curent*/
    /*sortare prin inserţie 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];  j=j-h;
          } /*while*/
        a[j]=temp;     /*inserţie*/
      } /*for*/
   } while (!(h==1));
} /*shellsort*/
```

3.2.5. Sortarea prin metoda ansamblelor. Sortarea heapsort

* Metoda **sortării prin selecţie** se bazează pe selecţia 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 comparaţii, dintre n-1 elemente necesită n-2 comparaţii, etc.

* Activitatea de selecţie poate fi **îmbunătăţită**, dacă la fiecare trecere se vor reţine **mai multe informaţii** şi nu doar elementul cu cheia cea mai mică.

    * Astfel spre exemplu din n/2 comparaţii se poate determina **cea mai mică cheie a fiecărei perechi de elemente**.

    * Din alte n/4 comparaţii, **cea mai mică cheie a fiecărei perechi de chei mici** deja determinate şi aşa mai departe.

    * În final, utilizând doar n/2 + n/4 + · · · + 4 + 2 + 1= n-1 comparaţii, se poate construi un **arbore de selecţii** având drept rădăcină cheia cea mai mică (fig.3.2.5.a).

    * Arborele de selecţii este de fapt un **arbore binar parţial ordonat**.

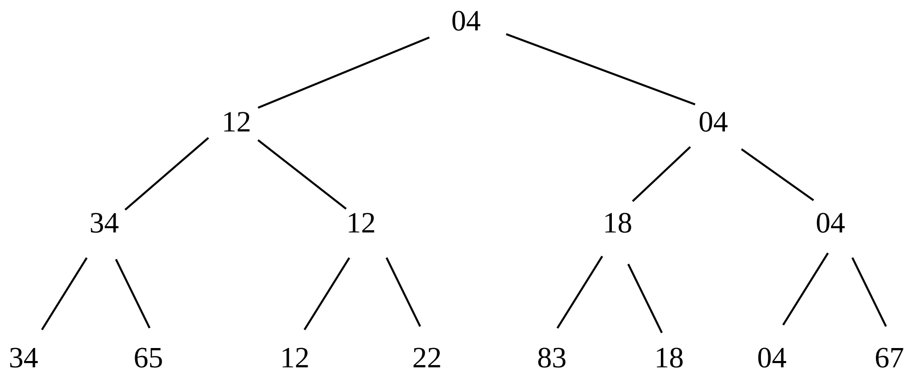


Fig.3.2.5.a. Arbore de selecţii

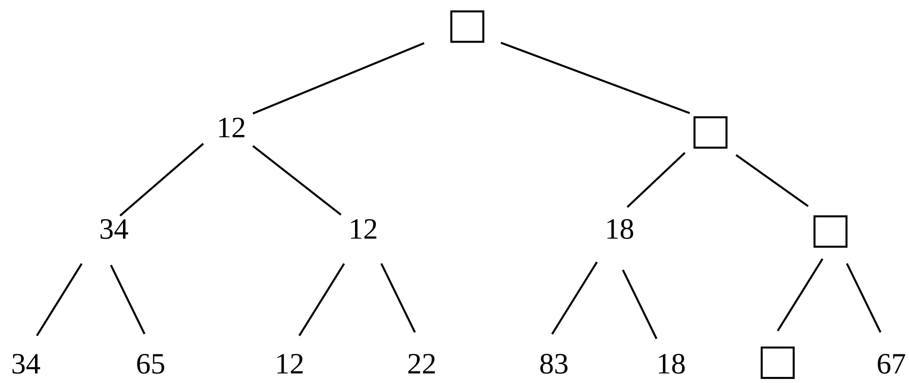


Fig. 3.2.5.b. Selecţia traseului celei mai mici chei

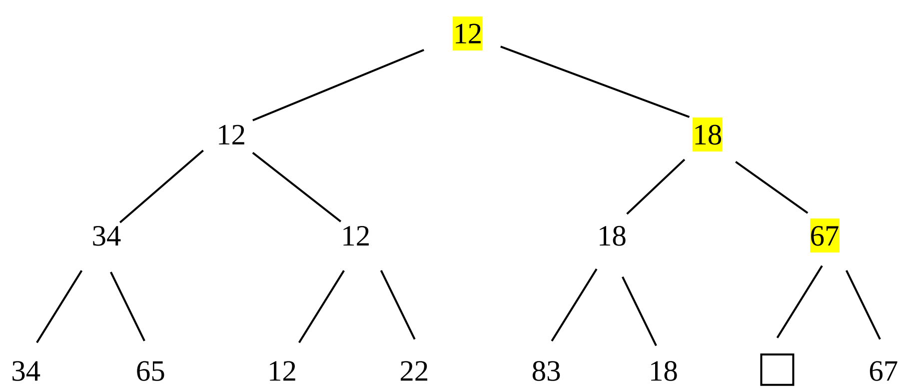


Fig. 3.2.5.c. Completarea locurilor libere

- 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 şi se elimină succesiv această cheie (Fig. 3.2.5.b).

    * Pe acest parcurs cheia se înlocuieşte (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 şi procesul se repetă.

* După `n` astfel de paşi de selecţie, s-au extras succesiv cele `n` elemente ale mulţimii în ordine crescătoare, arborele devine vid şi procesul de sortare este încheiat.

* Trebuie notat faptul că fiecare din cei `n` paşi de selecţie necesită **numai** $\log_2 n$ comparaţii, adică un număr de comparaţii egal cu înălţimea arborelui.

* În consecinţă, **procesul de sortare integrală** necesită:

    * Un număr de `n` paşi pentru construcţia arborelui.

    * Un număr de operaţii elementare de ordinul lui $n \cdot \log_2 n$ pentru sortarea propriu-zisă.

* Aceasta este o îmbunătăţire considerabilă faţă de metodele directe care necesită un efort de ordinul $O(n^2)$ şi chiar faţă de Shellsort care necesită $O(n^{1.2})$.

* Este evident faptul că în cazul metodei de sortare bazată pe structura arbore, **complexitatea** paşilor de sortare individuali creşte.

* De asemenea, în vederea reţinerii unei cantităţi sporite de informaţie, trebuie concepută o **structură de date** aparte care să permită organizarea eficientă a informaţiei.

* Respectiva **structură de date** trebuie să respecte următoarea **specificaţie**:

    * (1) În primul rând, să elimine **locurile goale**, care pe de o parte sporesc dimensiunea arborelui, iar pe de altă parte sunt sursa unor comparaţii care nu sunt necesare.

    * (2) În al doilea rând, arborele ar trebui reprezentat utilizând locaţii de memorie pentru `n` elemente şi nu pentru `2n -1` elemente aşa 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 ansamble*).

* Metoda în sine, reprezintă o realizare de **excepţie** printre metodele convenţionale de sortare şi utilizează o reprezentare specială a unui **arbore binar parţial ordonat**, numită ***"heap"*** sau "***ansamblu***".

* Un **ansamblu** ("**heap**") este definit ca o secvenţă de chei $h_{stanga}, h_{stanga+1}, ..., h_{dreapta}$ care se bucură de proprietăţile [3.2.5.a]:

$$h_i \le h_{2i} \quad h_i \le h_{2i+1} \qquad \text{pentru toţi} \quad i = stanga, ..., dreapta/2 \qquad [3.2.5.a]$$

- Un **ansamblu** poate fi asimilat cu un **arbore binar parţial ordonat** şi reprezentat printr-un **tablou**.

- Spre exemplu, ansamblul $h_1, h_2, ..., h_{15}$ poate fi asimilat cu arborele binar din figura 3.2.5.d şi poate fi reprezentat prin tabloul `h` în baza următoarei tehnici:

  - (1) Se **numerotează** elementele ansamblului, nivel cu nivel, de sus în jos şi de la stânga la dreapta.

  - (2) Se **asociază** elementelor ansamblului, locaţiile unui **tablou** de elemente `h`, astfel încât elementului h_i al ansamblului îi corespunde locaţia `h[i]` din tablou.

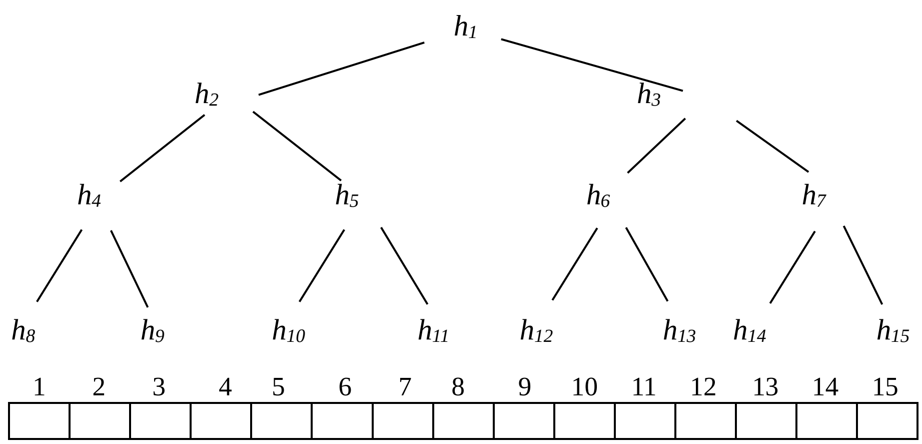


Fig. 3.2.5.d. Reprezentarea unui ansamblu printr-un tablou liniar `h`

- Un ansamblu se bucură de proprietatea că **primul** său element este cel mai mic dintre toate elementele ansamblului adică $h_1 = \min(h_1, ..., h_n)$.

- Se presupune un **ansamblu parţial** $h_{s+1}, h_{s+2}, ..., h_d$ definit prin indicii `s+1` şi `d`.

  - Acestui ansamblu i se adaugă la stânga, pe poziţia h_s un nou element `x`, obţinându-se un **ansamblu extins spre stânga** $h_s, ..., h_d$.

- În figura 3.2.5.e.(a) apare ca exemplu ansamblul $h_2, ..., h_7$, iar în aceeaşi figură (b), ansamblul extins spre stânga cu un element `x=34`.

  - Noul ansamblu se obţine din cel anterior plasând pe *x* în vârful ansamblului şi deplasându-l "*în jos*" de-a lungul drumului indicat de componentele cele mai mici, care în acelaşi 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 uşurinţă că această deplasare conservă condiţiile care definesc un **ansamblu** [3.2.5.a].

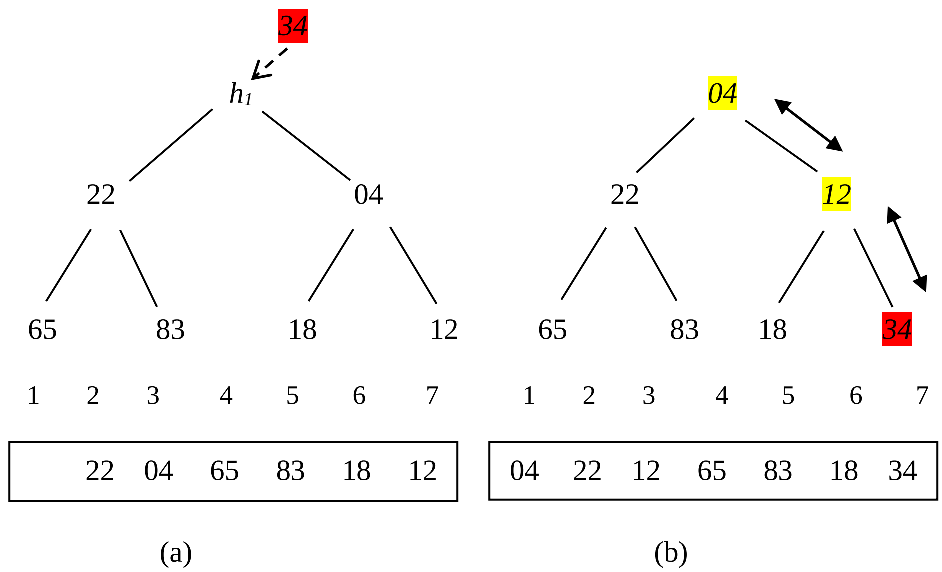


Fig. 3.2.5.e. Deplasarea unei chei într-un ansamblu

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

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

procedure Deplasare(TipIndice stanga,dreapta)  [3.2.5.b]
  /*stânga şi 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) şi 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);
-------------------------------------------------------------
```

- Funcţia C care implementează algoritmul de deplasare apare în secvenţa [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 şi dreapta sunt limitele ansamblului*/
  /*elementul de deplasat se află în poziţia h[stanga]*/
  tip_indice i,j;
  tip_element temp;
  boolean ret;

  i=stanga;    /*indică elementul curent*/
  j=2*i;       /*indică fiul stâng al elementului curent*/
  temp=h[i];   /*elementul care se deplasează*/
  ret=false;
  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 pe temp 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 parţial 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** şi operatorul ***deplasare*** prezentat mai sus:

    - Se consideră un tablou $h_1,...,h_n$ care conţine cele n elemente din care se va construi ansamblul.

    - În mod evident, elementele $h_{n/2},...,h_n$ formează deja un **ansamblu** deoarece **nu** există nici o pereche de indici `i` şi `j` care să satisfacă relaţia `j=2*i` (sau `j=2*i+1`).

    - Aceste elemente formează cea ce poate fi considerat drept **şirul de bază** al **ansamblului** asociat.

- În continuare, ansamblul $h_{n/2}, ..., h_n$ este **extins spre stânga**, la fiecare pas cu câte un element, introdus în vârful ansamblului şi deplasat până la locul său.

- Prin urmare, considerând că tabloul iniţial este memorat în `h`, procesul de generare *"in situ"* al unui **ansamblu** poate fi descris prin secvenţa [3.2.5.d].

```
/*Faza creare ansamblu*/

  stanga=(n DIV 2)+1;                                  [3.2.5.d]
  cât timp stanga>1
    stanga=stanga-1;
    deplasare(stanga,n);
  □
```

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

- În vederea **sortării elementelor**, se execută `n` paşi 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 şi se reia procesul.

  - În final rezultă tabloul sortat în ordine descrescătoare.

- În termenii operatorului ***`deplasare`*** această tehnică poate fi descrisă ca în secvenţa [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;
    deplasare(1,dreapta);
  □
```

- Cheile se obţin sortate în **ordine inversă**, lucru care poate fi uşor remediat modificând sensul relaţiilor de comparaţie 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]*/

static void deplasare1(tip_indice* stanga, tip_element*
            temp, tip_indice* dreapta)
  {
    tip_indice i,j; boolean ret;

    i=*stanga; j=2*i; *temp=h[i]; ret=false;
    while((j<=*dreapta)&&(!ret))
      {
        if(j<*d)
          if(h[j].cheie<h[j+1].cheie)  j=j+1;
        if(temp->cheie<h[j])
            {
              h[i]=h[j]; i=j; j=2*i;
            }
          else
            ret=true;
      } /*while*/
    h[i]= *temp;
  } /*deplasare1*/

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

  /*faza construcţie ansamblu*/
  stanga= (n/2)+1; dreapta= n;
  while(stanga>1)
    {
      stanga=stanga-1;
      deplasare1(&stanga,&temp,&dreapta);
    } /*while*/
  /*faza sortare ansamblu*/
  while(dreapta>1)
    {
      temp=h[1]; h[1]=h[dreapta]; h[dreapta]=temp;
      dreapta=dreapta-1;
      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 performanţei metodei heapsort** contrazice însă această părere.

    - (1) La **faza de construcţie a ansamblului** sunt necesari `n/2` paşi de deplasare.

        - În **fiecare pas** se mută elemente de-a lungul a respectiv `log2(n/2),log2(n/2 +1),... ,log2(n-1)` poziţii, (în cel mai defavorabil caz), unde logaritmul se ia în baza 2 şi se trunchiază la prima valoare întreagă.

- (2) În continuare, **faza de sortare** necesită `n-1` deplasări fiecare cu cel mult respectiv $\log_2(n-2), \log_2(n-1), \ldots, 1$ mişcări.

- (3) În plus mai sunt necesare $3\cdot(n-1)$ mişcări pentru a **aşeza** elementele sortate în ordine.

- Toate acestea dovedesc că în cel mai defavorabil caz, tehnica **heapsort** are nevoie de un număr de paşi 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 şi 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 mişcări** este aproximativ egal cu $1/2 \cdot n \cdot \log_2 n$, adică o mişcare la 2 paşi de sortare, deviaţiile 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, eficienţa metodei crescând o dată cu creşterea lui `n`.

- În secvenţa [3.2.5.h] se prezintă varianta C a algoritmului de sortare prin metoda ansamblelor.

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

/*globale: int a[],int n*/                    /*[3.2.5.h]*/
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() {
 /*construcţie 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 partiţionare. Sortarea quicksort

- După cum s-a demonstrat, **metoda de sortare bubblesort** bazată pe principiul interschimbării este cea mai puţin performantă dintre metodele de sortare studiate.

- **C.A.R. Hoare** însă, pornind de la acelaşi principiu, a conceput o metodă de sortare cu performanţe spectaculare pe care a denumit-o **quicksort** (sortare rapidă).

- Metoda se bazează pe aceeaşi idee de a creşte eficienţa interschimbărilor prin mărirea distanţei dintre elementele implicate.

- **Sortarea prin partiţionare** se bazează pe următorul **algoritm**:

  - Fie *x* un **element oarecare** al tabloului de sortat $a_1,...,a_n$.

  - Se parcurge tabloul de la **stânga spre dreapta** până se găseşte primul element $a_i>x$.

  - În continuare se parcurge tabloul de la **dreapta spre stânga** până se găseşte primul element $a_j<x$.

  - Se **interschimbă** între ele elementele a_i şi 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, ş.a.m.d.

  - Procesul se termină când cele două parcurgeri se *"**întâlnesc**"* undeva în interiorul tabloului.

- **Efectul final** este acela că şirul iniţial este **partiţionat** într-o **partiţie stânga** cu chei mai mici decât *x* şi o **partiţie dreapta** cu chei mai mari decât *x*.

- Considerând elementele şirului memorate în tabloul `a`, **principiul partiţionării** apare prezentat sintetic în [3.2.6.a].

```
/*Partiţionarea unui tablou - varianta pseudocod – pas 1 de
rafinare*/

procedure Partiţionare                        /*[3.2.6.a]*/
/*partiţionează tabloul a[s..d]*/

 *selectează elementul x (de regulă de la mijlocul
                         intervalului de partiţionat);
 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 **partiţionării** în forma unei proceduri [3.2.6.b].

- Se precizează că relaţiile > respectiv < au fost înlocuite cu ≥ respectiv ≤ ale căror negate utilizate în instrucţiunile **WHILE** sunt < respectiv >.

  - În acest caz x joacă rol de **fanion** pentru ambele parcurgeri.

```
/*Procedura Partiţionare - varianta pseudocod - pas 2 de
rafinare*/

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

      i=stanga; j=dreapta;
      x=a[stanga+dreapta DIV 2];
       repeta
       |  cât timp (a[i].cheie<x.cheie)  i=i+1;
       |  cât timp (a[j].cheie>x.cheie)  j=j-1;
       |  dacă (i<=j)
       |   |*interschimba a[i] cu a[j];
       |   |i=i+1; j=j-1;
       |   |□
       |până când (i>j)
       |□
     } /*Partitionare*/
```

- În continuare, cu ajutorul partiţionării, **sortarea** se realizează simplu:

  - După o primă partiţionare a secvenţei de elemente se aplică acelaşi procedeu celor două partiţii rezultate.

  - Apoi celor patru partiţii ale acestora, ş.a.m.d.

  - Procesul se termină când fiecare partiţie se reduce la un singur element.

- **Tehnica sortării bazată pe partiţionare** este ilustrată în secvenţa [3.2.6.c].

```
*Sortarea prin partiţionare -quicksort - varianta pseudocod

procedure QuickSort(stanga,dreapta)              [3.2.6.c]
  *partiţionează intervalul stanga,dreapta faţă de Mijloc;
  dacă există partiţie stânga atunci
    QuickSort(stanga,Mijloc-1);
  dacă există partiţie dreapta atunci
    QuickSort(Mijloc+1,dreapta);
```

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

```
/*Sortarea prin partiţionare - quicksort*/

/*globale: int a[],int n*/
```

```
quicksort(int stanga,int dreapta) {        /*[3.2.6.e]*/
 int i=stanga,j=dreapta,x=a[(stanga+dreapta)/2];
 do { /*partiţionarea intervalului curent stânga, dreapta*/
   while(a[i]<x) i++;
   while(a[j]>x) j--;
   if(i<=j) {
     int temp=a[i];
     a[i]=a[j];
     a[j]=temp;
     i++;j--;
   }
 } while(i<=j);
 if(s<j)quicksort(stanga,j); /*procesează intervalul stâng*/
 if(d>i)quicksort(i,dreapta);/*procesează intervalul drept*/
} /*quicksort*/
```

- În continuare se prezintă o manieră de implementare a aceluiaşi algoritm utilizând o procedură **nerecursivă**.
- Elementul cheie al **soluţiei iterative** rezidă în menţinerea unei **liste a cererilor de partiţionare.**
  - La fiecare trecere apar două noi partiţii.
  - 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 şi invers.
    - Cu alte cuvinte lista se tratează ca o **stivă**, de fapt ea chiar este o stivă.
  - În secvenţa [3.2.6.f] apare o schiţă 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 iniţial
      de sortare (amorsarea procesului);
   repetă
     *se extrage intervalul din vârful stivei care
      devine IntervalCurent;
     *se reduce stiva cu o poziţie;
     repetă
       repetă
         *se partiţionează IntervalCurent;
       până când terminare partiţionare;
       □
       dacă există interval drept atunci
         *se introduc limitele sale în stivă;
       □
      *se face intervalul stâng IntervalCurent;
     până când intervalul ajunge de lăţime 1 sau 0;
     □
   până când stiva se goleşte;
   □
```

- În procedura care implementează algoritmul din secvenţa anterioară [3.2.6.f]:

  - (1) O **cerere de partiţionare** este reprezentată printr-o pereche de indici care delimitează zona de tablou ce urmează a fi partiţionată.

  - (2) **Stiva** este modelată cu ajutorul unui **tablou** cu dimensiunea variabilă numit `stiva` şi 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*/

/*globale: int a[],int n */
void qsort_nerec()                                /*[3.2.6.g]*/
{
    enum {m= 15};
    int i,j,stanga,dreapta;
    tip_element x,temp;
    unsigned char is; /*indicator stiva*/
    struct {
        int stanga,dreapta;
    } stiva[m];  /*definire stivă*/

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

  do {     /*se ia cererea din vârful stivei*/
    stanga=stiva[is].stanga;
    dreapta=stiva[is].dreapta; is=is-1;
    do {   /*partiţionarea intervalului a[stânga],
              a[dreapta]*/
      i=stanga; j=dreapta; x=a[(stanga+dreapta)/2];
      do {
        while(a[i].cheie<x.cheie)  i=i+1;
        while(x.cheie<a[j].cheie)  j=j-1;
        if(i<=j)
          {
            temp=a[i]; a[i]=a[j]; a[j]=temp;
            i=i+1; j=j-1;
          }
      } while(!(i>j));
      if(i<dreapta)
        {  /*partiţia dreapta se introduce în stivă*/
          is=is+1;
          stiva[is].stanga=i;
          stiva[is].dreapta=dreapta;
        }
      dreapta=j; /*partiţia stânga devine curentă*/
    } while(!(stanga>=dreapta)); /*sfârşit partiţionare
                                   curentă*/
  } while(!(is==-1)); /*stiva este vida*/
} /*quicksort_nerecursiv*/
------------------------------------------------------------
```

3.2.6.1. Analiza metodei quicksort

- Pentru a analiza performanţa acestei metode, se analizează mai întâi **partiţionarea**.
    - Se presupun pentru simplificare următoarele **precondiţii**:
        - (1) Setul ce urmează a fi partiţionat constă din n chei **distincte** şi **unice** cu valorile $\{1,2,3,...,n\}$.
        - (2) Dintre cele n chei a fost selectată cheia cu valoarea x în vederea partiţionării.
    - În consecinţă această cheie ocupă a ***x*-a poziţie** în mulţimea ordonată a cheilor şi 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 partiţiei 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** şi numărul de cazuri **posibile**).
- (2) Care este **numărul de interschimbări** necesar pentru o partiţionare a unei secvenţe de n chei, dacă s-a selectat ca **pivot** cheia situată pe **poziţia** x ?
    - La dreapta pivotului există $n-x$ poziţii care vor fi procesate în procesul de partiţionare.
    - **Numărul posibil de interschimbări** în acest context este deci egal cu **produsul** dintre numărul de chei care vor fi selectate $(n-x)$ şi probabilitatea determinată anterior ca o cheie selectată să fie interschimbată [3.2.6.h].

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

- (3) Care este **numărul mediu de interschimbări** pentru partiţionarea unei **secvenţe de *n* chei**?
    - La o partiţionare poate fi selectată oricare din cele n chei ca şi pivot, deci x poate lua orice valoare cuprinsă între 1 şi n.
    - **Numărul mediu** M **de interschimbări** pentru **partiţionarea unei secvenţe** de n chei se obţine astfel:

- (1) Pentru fiecare valoare a lui x selectat ca pivot, cuprinsă între 1 şi 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 obţinută la numărul total de chei n [3.2.6.i].

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

- Presupunând în mod exagerat că întotdeauna va fi selectată **mediana** partiţiei (mijlocul său valoric), fiecare partiţionare va divide tabloul în două jumătăţi egale.

  - Se face precizarea că **mediana** este elementul situat ca şi valoare în mijlocul partiţiei, dacă aceasta este ordonată.

- În aceste condiţii, 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 paşi de partiţionare integrală a tabloului.

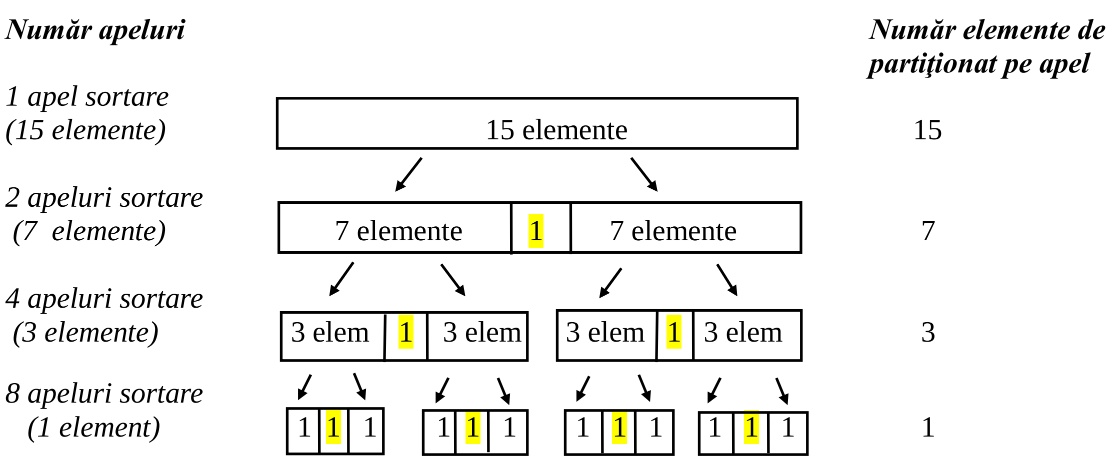


Fig. 3.2.6.a. Funcţionarea principială a sortării prin partiţionare

- Din păcate **numărul de apeluri recursive** ale procedurii este egal cu 15, adică exact cu numărul de elemente.

- Rezultă că **numărul total de comparaţii** este $n \cdot \log_2 n$ deoarece la o trecere sunt comparate **toate** cheile [3.2.6.j].

* **Numărul de mişcări** este $n/6 \cdot log_2\ n$ deoarece conform formulei [3.2.6.i] la partiţionarea a n chei sunt necesare în medie $n/6$ mişcări [3.2.6.j].

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

---

* Aceste rezultate sunt **excepţional** 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 datoreşte însă faptului surprinzător că **performanţa sa medie**, la care alegerea pivotului se face la întâmplare, este inferioară performanţei optime doar cu un factor egal cu $2 \cdot ln(2) =$ 1.4 deci cu aproximativ 40 % [Wi76].

* Tehnica prezentată are însă şi **dezavantaje**.

    * (1) În primul rând ca şi la toate metodele de sortare avansate, performanţele ei sunt **moderate** pentru valori mici ale lui n.

        * Acest dezavantaj poate fi contracarat prin încorporarea unor **metode de sortare directe** pentru partiţiile 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 performanţa metodei scade catastrofal.

        * Acest caz apare când la fiecare partiţionare este selectată cea mai mare valoare ca şi pivot.

        * În acest caz, fiecare pas va partaja secvenţa formată din n elemente, într-o partiţie stânga cu $n-1$ elemente şi o partiţie dreapta cu un singur element.

        * Vor fi necesare astfel n partiţionări în loc de $log(n)$, iar performanţa obţine valori de ordinul $O(n^2)$.

* În mod aparent, elementul esenţial al acestei metode îl reprezintă **selecţia pivotului** x [GG78].

* În exemplul prezentat, pivotul a fost ales la **mijlocul partiţiei**.

    * El poate fi însă ales la extremitatea stângă sau dreaptă a acesteia, situaţie în care, cazul cel mai defavorabil îl reprezintă partiţia deja sortată.

* Tehnica quicksort se comportă straniu:

    * Are **performanţe slabe** în cazul sortărilor banale.

    * Are **performanţe 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 performanţa medie este cea mai bună în cazul alegerii **pivotului** la **mijlocul partiţiei**.

* **Hoare** sugerează ca alegerea să se facă:

    * (1) Fie la "întâmplare".

    * (2) Fie prin selecţia medianei unui număr redus de chei (spre exemplu trei chei).

    * O astfel de alegere judicioasă a pivotului influenţează serios în mod **negativ performanţa medie** a algoritmului quicksort, dar **îmbunătăţeşte** în mod considerabil **performanţa cazului cel mai defavorabil**.

* Pentru programator, în multe situaţii **cazul cel mai defavorabil** are o influenţă deosebită.

    * Spre exemplu în secvenţa [3.2.6.g] care implementează **sortarea quiqsort** în manieră **iterativă**, în cazul cel mai defavorabil, la fiecare partiţionare rezultă o partiţie 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, situaţie care nu este acceptabilă.

    * Acest lucru este şi mai grav în cazul **recursivităţii** unde stiva gestionată în mod automat este mult mai substanţială, fiind necesar câte un nod pentru fiecare apel recursiv, nod care presupune spaţiu de memorie pentru stocarea valorilor parametrilor locali ai apelului la care se adaugă de regulă şi codul efectiv al procedurii.

    * Această situaţie se poate rezolva în cazul **implementării iterative**, introducând întotdeauna în stivă **cererea de sortare a partiţiei mai mari** şi continuând cu partiţionarea partiţiei mai mici.

        * În acest caz dimensiunea m a stivei poate fi limitată la $m=\log_2 n$.

    * Pentru a implementa această tehnică, secvenţa [3.2.6.g] se modifică în porţiunile în care se procesează cererile de partiţionare 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) /*partiţia dreapta este mai mare decat
                      stanga*/
        {  /*cerere sortare partiţie dreapta în stivă*/
          is=is+1; stiva[is].stanga=i;
          stiva[is].dreapta=dreapta;
        }
      dreapta=j  /*se continuă  sortarea partiţiei stânga*/
    }
  else
```

```
    {
      if(stanga<j)  /*partiţia stânga este mai mare*/
        {   /*cerere sortare partiţie stânga în stivă*/
          is=is+1; stiva[is].stanga=stanga;
          stiva[is].dreapta=j;
        }
      stanga=i  /*se continuă sortarea partiţiei 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 şi este mai mare (sau egal) decât cealaltă jumătate.
    - Spre exemplu mediana secvenţei 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 partiţionării** 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 acelaşi 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** funcţionează după cum urmează.
    - Se presupune că elementele avute în vedere sunt memorate în **tabloul** `a` cu dimensiunea `n`.
    - Pentru început se realizează o **partiţionare** cu limitele `stanga=0`, `dreapta=n-1` şi cu `a[k]` selectat pe post de pivot `x`.
    - În urma acestei **partiţionări** rezultă valorile index `i` şi `j` care satisfac relaţiile [3.2.7.a].

----

1) `x=a[k]`
2) `a[h]≤x` pentru toţi `h<i`
3) `a[h]≥x` pentru toţi `h>j` [3.2.7.a]
4) `i>j`

- Sunt posibile trei **situaţii**:

- (1) Valoarea pivotului `x` este prea **mică**, astfel încât limita dintre cele două partiţii este sub valoarea dorită `k`.
    - Procesul de partiţionare se reia pentru elementele **partiţiei dreapta** `a[i],...,a[dreapta]` (fig.3.2.7.a (a)).
- (2) Valoarea pivotului `x` este prea **mare**.
    - Operaţia de partiţionare se reia pentru elementele **partiţiei stânga** `a[stanga],...,a[j]` (fig.3.2.7.a (b)).
- (3) `j<k<i.`
    - În acest caz elementul `a[k]` separă tabloul în două partiţii, el desemnând **mediana** (fig.3.2.7.a (c)).

- Procesul de partiţionare se repetă până la realizarea cazului (3).

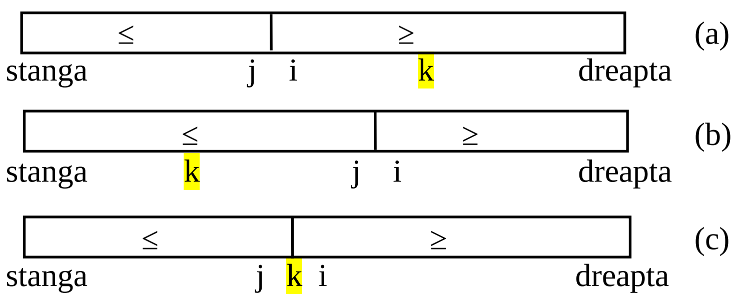


Fig. 3.2.7.a. Determinarea medianei

- Algoritmul aferent este prezentat în variantă peseudocod în secvenţa [3.2.7.b] respectiv o primă rafinare în secvenţa de program [3.2.7.c].

```
------------------------------------------------------------
/*Aflarea medianei - Varianta pseudocod*/

  procedure Mediana (stanga,dreapta,k)          [3.2.7.b]
  | cât timp există partiţie
  |    *alege pivotul (elementul din poziţia k);
  |    *partiţionează intervalul curent faţa de valoarea
  |       Pivotului;
  |     dacă limită partiţionare<k atunci *selectează
  |         partiţia dreapta;
  |     dacă limită partiţionare>k atunci *selectează
  |         partiţia stânga;
  | □
------------------------------------------------------------
{Procedura Mediana - Pas rafinare 1}

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

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

```
/*Procedura Mediana - Implementare C*/

void mediana (int k)                              /*[3.2.7.d]*/
  {
    tip_element a[n],x,temp;
    int stanga,dreapta,i,j;

    stanga=0; dreapta=n-1;
    while(stanga<dreapta)
      {
        x=a[k]; i=stanga; j=dreapta;
        do {    /*partiţionarea în raport cu a[k]*/
          while(a[i]<x) i++;
          while(x<a[j]) j--;
          if (i<=j)
           {
              temp=a[i]; a[i]=a[j]; a[j]=temp;
              i++1; j--;
            }
        } while (!(i>j));
        if(j<k)  stanga=i; /*selectează partiţia stânga*/
        if(k<i)  drapta=j; /*selectează partiţia dreapta*/
      }
  }
/*mediana*/
```

3.2.7.1. Analiza determinării medianei

- Dacă se presupune că în medie fiecare partiţionare înjumătăţeşte partiţia în care se găseşte elementul căutat, atunci **numărul** necesar **de comparaţii** *C* este de ordinul O*(n)* [3.2.7.e].

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

- **Numărul de mişcări** M **nu** poate depăşi numărul de comparaţii, el fiind de regulă mai mic, ca atare tot O*(n)*.
- Valorile indicatorilor C şi M estimaţi pentru **determinarea medianei** subliniază superioritatea acestei metode.
- În acelaşi timp explică **performanţa superioară** a metodei faţă de metodele bazate pe sortarea tabloului şi extragerea celui de-al *k*-lea element, a căror performanţă în cel mai bun caz este de ordinul $O(n \cdot log_2 n)$.
- În cel mai **defavorabil** caz:

* Fiecare partiţionare reduce setul de candidaţi numai cu 1, rezultând un număr de comparaţii de ordinul $O(n^2)$.

* Şi în acest caz metoda **medianei** este indicată pentru valori mari ale lui *n* (*n*>10).

3.2.8. Sortarea binsort. Determinarea distribuţiei cheilor

* În general algoritmii de sortare bazaţi pe metode avansate au nevoie de $O(n \cdot \log_2 n)$ paşi pentru a sorta *n* elemente.

* Trebuie precizat însă faptul că acest lucru este valabil în situaţia în care:

  * Nu există nici o altă informaţie suplimentară referitoare la chei, decât faptul că pe mulţimea acestora este definită o **relaţie 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 şi **mai rapid** decât în limitele performanţei $O(n \cdot \log_2 n)$, **dacă**:

  * (1) Există şi **alte informaţii** referitoare la **cheile** care urmează a fi sortate.

  * (2) Se **renunţă** 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 şi aparţin intervalului de la `1` la `n`.

  * Dacă `a` şi `b` sunt **tablouri** cu câte `n` elemente, `a` conţinând cheile care urmează a fi sortate, atunci sortarea se poate realiza direct în tabloul `b`, într-o singură trecere, conform secvenţei [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]` şi se plasează elementul exact la locul potrivit în tabloul `b`.

  * Întregul ciclu necesită $O(n)$ paşi.

  * 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 aceeaşi cheie va fi introdus tot în `b[x]` distrugând elementul anterior.

* Acest tip de sortare poate fi realizat şi ***"in situ"*** (secvenţa [3.2.8.b])

    * Astfel, fiind dat tabloul `a` de dimensiune *n*, ale cărui elemente au respectiv cheile `0,...,n-1`,

    * Se baleează pe rând elementele sale (bucla **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 locaţia `i` exact la locul său în tabloul ordonat, fiind necesare în cel mai rău caz $3 \cdot n$ mişcări pentru întreg procesul de sortare.

    * Secvenţa 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];
------------------------------------------------------------
```

* Secvenţele [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 fiecare element de sortat.

    * (2) Se introduce în **bin**-ul corespunzător valorii cheii.

        * În secvenţa [3.2.8.a] bin-urile sunt chiar elementele tabloului `b`, unde `b[i]` este binul cheii având valoarea `i`.

        * În secvenţa [3.2.8.b] bin-urile sunt chiar elementele tabloului `a` după reaşezare.

* Tehnica aceasta simplă şi performantă se bazează pe următoarele **cerinţe apriorice**:

    * (1) **Domeniul limitat** al cheilor `[0, n-1]`.

    * (2) **Unicitatea** fiecărei chei.

* Dacă cea de-a doua cerinţă **nu** este respectată, şi de fapt acesta este cazul obişnuit, este necesar ca într-un **bin** să fie memorate **mai multe elemente** având aceeaşi cheie.

    * Acest lucru se realizează fie prin **înşiruire**, fie prin **concatenare**, fiind utilizate în acest scop **structuri listă**.

    * Această situaţie **nu** deteriorează prea mult performanţele 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 situaţii este cea bazată pe **determinarea distribuţiei 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 distribuţiei 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 poziţia lor ordonată în tabloul `b`.

* Formularea **algoritmului de sortare cu determinarea distribuţiilor cheilor** este cea din secvenţa [3.2.8.c].

    * Pentru simplificare se presupune că tabloul `a` conţine doar chei.

```
------------------------------------------------------------
/*Sortare cu determinarea distribuţiilor cheilor - Varianta
C*/

enum {n = 100, m = 10};

typedef int tip_tablou[n];                    /*[3.2.8.c]*/
int contor[m];
tip_tablou a,b;
int i,j;

int main(int argc, const char* argv[])
{
  /*iniţializare contoare*/
  for(j=0;j<m-1;j++) contor[j]=0;
  /*determinarea distribuţiei 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 distribuţiei cheilor în b*/
  for(i=n-1;i>0;i--)
    {
     b[contor[a[i]]-1]=a[i];
```

```
      contor[a[i]]=contor[a[i]]-1;
    }
  /*mută tabloul b în a*/
  for(i=0;i<n-1;i++) a[i]=b[i];
  return 0;
}  /*Sortare cu determinarea distribuţiei cheilor*/
```

- Funcţionarea algoritmului:
  - Contoarele asociate cheilor sunt memorate în tabloul `contor` de dimensiune `m`.
  - Iniţial locaţiile tabloului `contor` sunt iniţializate 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 sfârşit 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 şi **decrementarea** contoarelor specifice astfel încât în final, cheile identice apar în binul specific în **ordinea relativă** în care apar în secvenţa iniţială.
  - Ultima buclă **`for`** realizează **mutarea** integrală a elementelor tabloului `b` în tabloul `a`, (dacă acest lucru este necesar).
- Deşi se realizează mai multe treceri prin elementele tabloului totuşi în ansamblu, **performanţa algoritmului de sortare baza pe determinarea distribuţiei 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 funcţionare a **algoritmului de sortare bazat pe determinarea distribuţiei cheilor**.

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

1) Se consideră iniţial că tabloul `a` are următorul conţinut:

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 iniţializează 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 şi se introduc pe rând în tabloul `b`, fiecare în poziţia 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 performanţei sortării cu determinarea distribuţiei cheilor

- Deşi sunt necesare mai multe treceri pentru a sorta un tablou, performanţa de ansamblu a sortării cu **determinarea distribuţiei 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 şi raţiunea pentru care această metodă stă la baza mai multor metode de sortare de tip **radix**.

3.2.9. Sortarea bazată pe baze de numeraţie. Radix sort

- Metodele de sortare prezentate până în prezent, concep cheile de sortat ca **entităţi** pe care le prelucrează integral prin comparare şi interschimbare.

- În unele situaţii însă se poate profita de faptul că aceste chei sunt de fapt **numere** exprimate prin **cifre** aparţinând unui **domeniu mărginit**.

- Metodele de sortare care iau în considerare ***proprietăţile digitale*** ale numerelor sunt **metodele de sortare bazate pe baze de numeraţie** ("**radix sort**").

- Algoritmii de tip **bază de numeraţie**:

    - (1) Consideră cheile ca şi numere reprezentate într-o **bază de numeraţie** *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 şi 199, etc., adică se realizează o sortare după **cifra sutelor**.

    * În continuare se sortează pe rând grupele formate aplicând aceeaşi metodă, după **cifra zecilor**.

    * Apoi fiecare grupă nou formată, se sortează după **cifra unităţilor**.

* 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, **operaţia fundamentală** este extragerea unui **set contiguu de biţi** din numărul binar care reprezintă **cheia.**

    * Spre **exemplu:**

        * Pentru a extrage primii 2 biţi ai unui număr binar format din 10 cifre binare:

            * (1) Se realizează o **deplasare la dreapta** cu 8 poziţii a reprezentării binare a numărului.

            * (2) Se operează configuraţia obţinută, printr-o **operaţie** "**şi**" cu masca 0000000011.

* Aceste operaţii:

    * (1) Pot fi **implementate direct** cu ajutorul facilităţilor de prelucrare a configuraţiilor la nivel de biţi puse la dispoziţie de limbajele de programare.

    * (2) Pot fi **simulate** cu ajutorul operatorilor întregi `DIV` şi `MOD`.

        * Spre **exemplu**, în situaţia anterioară, dacă `x` este numărul binar în cauză, primii doi biţi se obţin prin expresia `(x DIV 2`^{8}`)MOD 2`^{2}.

* În continuare, pentru implementarea **sortărilor radix**, se consideră definit un **operator** ***biti***`(int x,k,j)` care combină cele două operaţii **returnând** valorile a **`j`** **biţi** care apar la **`k`** **poziţii** de la marginea dreaptă a lui `x`.

* O posibilă implementare în limbajul C a acestui **operator** apare în secvenţa [3.2.9.a].

```
------------------------------------------------------------
{operator care returnează j biţi care apar la k poziţii de
marginea dreaptă a lui x}

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

* Există **două metode de bază** pentru implementarea **sortării radix**.

* (1) Prima metodă examinează biţii cheilor de la **stânga la dreapta** şi se numeşte **sortare radix prin interschimbare** ("**radix exchange sort**").

    * Se bazează pe observaţia că:

        * Rezultatul comparaţiei a două chei este determinat de valoarea biţilor din **prima poziţie** la care ele diferă în parcurgere de la stânga la dreapta.

    * Astfel, elementele ale căror chei au primul bit 0 sunt trecute în faţa celor care au primul bit 1.

* În continuare în fiecare grup astfel format se aplică aceeaşi metodă pentru bitul următor şi aşa mai departe.

* Sortarea propriu-zisă se realizează prin schimbarea sistematică a elementelor în maniera precizată.

* (2) A doua metodă se numeşte **sortare radix directă** ("**straight radix sort**").

  * Ea examinează biţii din cadrul cheilor de la **dreapta la stânga.**

  * Se bazează pe principiul interesant care reduce sortarea cheilor de *b*-biţi 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 faţa celor ale căror chei încep cu 1.

    * Aceast proces va avea drept consecinţă formarea a două **partiţii** ale tabloului iniţial.

  * Cele două partiţii la rândul lor se sortează independent, conform aceleaşi metode **după cel de-al doilea bit** al cheilor elementelor, rezultând 4 partiţii.

  * Cele 4 partiţii rezultate se sortează similar **după al 3-lea bit**, ş.a.m.d.

* Acest mod de lucru sugerează abordarea **recursivă** a implementării metodei de sortare.

* **Procesul de sortare radix prin interschimbare** se desfăşoară exact ca şi la **partiţionare**:

  * Se balează tabloul de la **stânga spre dreapta** până se găseşte un element a cărui cheie începe cu 1.

  * Se baleează tabloul de la **dreapta spre stânga** până se găseşte un element a cărui cheie începe cu 0.

  * Se **interschimbă** cele două elemente.

  * Procesul **continuă** până când indicatorii de parcurgere **se întâlnesc** formând două partiţii.

  * Se reia aceeaşi procedură pentru **cel de-al doilea bit** al cheilor elementelor în cadrul **fiecăreia dintre cele două partiţii rezultate** ş.a.m.d. [3.2.9.1.a].

```
/*Sortare radix prin interschimbare - Varianta C*/
                                           /*[3.2.9.1.a]*/
void radix_interschimb (tip_indice stanga,tip_indice
     dreapta, int b) /*globale: a[n]*/
 /*stanga, dreapta - limitele curente ale tabloului de
   sortat*/
 /*b - lungimea în biţi a cheii de sortat*/

  {
    tip_indice i,j;
    tip_element t_;
```

```
    if((dreapta>stanga)&&(b>=0))
     {
       i=stanga; j=dreapta; b=b-1; /*iniţializare sortare*/
       /*partiţionare 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_;
       } while(!(j==i));
       if(biti(a[dreapta].cheie,b,1)==0)  j=j+1; /*dacă
        ultimul bit testat este 0 se reface lungimea
        partiţiei*/
       radix_interschimb(stanga,j-1,b-1);/*procesează
                                           partiţia stânga*/
       radix_interschimb(j,dreapta,b-1); /*procesează
                                          partiţia dreapta*/
     } /*if*/
   }
/*radix_interschimb*/
------------------------------------------------------------
```

* Spre exemplu, se presupune că tabloul `a[0..14]` conţine 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.

* O problemă serioasă care afectează această metodă se referă la **degenerarea** partiţiilor.

    * **Degenerarea partiţiilor** apare de obicei în situaţia în care cheile sunt reprezentate prin numere mici (care încep cu multe zerouri).

    * Această situaţie apare frecvent în cazul caracterelor interpretate pe 8 biţi.

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

- Din punctul de vedere al **performanţei**, **metoda de sortare radix prin interschimbare** sortează n chei de b biţi utilizând un număr de comparaţii de biţi egal cu $n \cdot b$.
    - Cu alte cuvinte, **sortarea radix prin interschimbare** este **liniară** cu **numărul de biţi ai unei chei**.
    - Pentru o **distribuţie normală** a biţilor 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 biţii cheilor elementelor de la **dreapta la stânga**.
    - Este metoda utilizată de vechile maşini de sortat cartele.
        - Teancul de cartele trece de 80 de ori prin maşină, câte odată pentru fiecare coloană începând de la dreapta spre stânga, fiecare trecere realizând sortarea după o coloană.
- Aceată metodă se numeşte **sortare radix directă**.
- **Ideea** metodei de **sortare radix directă**:
    - Se sortează cheile după un bit examinând biţii lor **de la dreapta spre stânga.**
    - Sortarea după bitul i constă în extragerea tuturor elementelor ale căror chei au zero pe poziţia i şi plasarea lor în faţa celor care au `1` pe aceeaşi poziţie.
    - Când se ajunge la bitul i venind dinspre dreapta, cheile sunt gata sortate pe ultimii $i-1$ biţi ai lor.
- **Nu** este uşor de demonstrat că metoda este corectă: de fapt ea este corectă **numai** în situaţia în care **sortarea după 1 bit** este o **sortare stabilă**.
    - Datorită acestei cerinţe, **interschimbarea normală nu** poate fi utilizată deoarece **nu** este o **metodă de sortare stabilă**.
- În ultimă instanţă trebuie **sortat stabil** un tablou cu numai două valori 0 şi 1.
- Metoda bazată pe **determinarea distribuţiilor cheilor** (**distribution counting**) poate fi utilizată cu succes în acest scop.
    - Se consideră **algoritmul de sortare** bazat pe **determinarea distribuţiei cheilor** în care:
        - (1) Se ia $m = 2$.
        - (2) Se înlocuieşte `a[i].cheie` cu ***`biti`***`(a[i].cheie,k,1)` - adică se extrage din cheie `1` bit situat la distanţa k faţă de sfârşitul cheii.

* (3) Se iterează sortarea bazată pe determinarea distribuţiilor pentru fiecare bit al cheii, de la **dreapta la stânga**, respectiv pentru toate valorile lui k = 0,1,2,...,b-1.

* Se obţine 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 creşte performanţa 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 biţi **odată:**

    * Timpul de sortare **scade** prin reducerea numărului de treceri.

    * Tabela de distribuţii însă **creşte** ca dimensiune ea trebuind să conţină `m1`=2^m locaţii, deoarece cu `m` biţi se pot alcătui 2^m configuraţii binare.

* În acest mod, **sortarea radix-directă** devine o **generalizare** a **sortării bazate pe determinarea distribuţiilor.**

* Algoritmul din secvenţa [3.2.9.2.a] sortează după această metodă tabloul `a[1..n]`.

    * Cheile sunt de b biţi lungime.

    * Cheile sunt parcurse de la **dreapta la stânga**, procesând câte m biţi odată. În consecinţă, pentru sortarea tabloului vor fi necesare b/m treceri.

    * Pentru sortare se utilizează tabloul suplimentar `t[1..n]`.

    * Procedura funcţionează **numai** dacă b este **multiplu** de m deoarece algoritmul de sortare divizează biţii cheilor într-un număr întreg de părţi de dimensiuni egale care se procesează deodată.

* Dacă se ia m=b se obţine **sortarea bazată pe determinarea distribuţiilor**;

* Dacă se ia m=1 rezultă **sortarea radix directă**.

* Implementarea propusă sortează tabloul `a` în tabloul `t` şi după fiecare pas de sortare recopiază tabloul `t` în `a` (ultima buclă **`for`**).

* Acest lucru poate fi evitat, **concatenând** în aceeaşi procedură **două copii** ale algoritmului de sortare: una care sortează din `a` în `t`, cealaltă din `t` în `a`.

```
/*Sortare radix directă - Varianta C*/

/*globale: a[n],t[n]*/
void radix_direct()                              /*[3.2.9.2.a]*/
  {
      int i,j,trecere;
      int contor[m1];             /*m1:=2m*/
```

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

- **Performanţa** sortării radix directe:
    - Sortează n elemente cu chei de b biţi în b/m treceri.
- **Dezavantajele** metodei de **sortare radix directă**:
    - (1) Utilizează un **spaţiu 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. Performanţa sortărilor radix

- **Timpul de execuţie** al celor două metode de sortare radix fundamentale, pentru n elemente având chei de b biţi este în esenţă proporţional cu $n \cdot b$.
- Pe de altă parte, timpul de execuţie poate fi aproximat ca fiind $n \cdot log_2(n)$, deoarece dacă toate cheile sunt diferite, b trebuie să fie cel puţin $log_2(n)$.
- 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 opreşte când apare o diferenţă între chei.
    - Metoda de la **dreapta la stânga** poate prelucra mai mulţi biţi deodată.

* Sortările radix se bucură de următoarele **proprietăţi** [Se88].

  * **Proprietatea 1**. Sortarea **radix prin interschimbare** examinează în medie $n \cdot log_2(n)$ biţi.

  * **Proprietatea 2.** La sortarea a n chei de câte b biţi, **ambele sortări radix** examinează de regulă mai puţini decât $n \cdot b$ biţi.

  * **Proprietatea 3.** Sortarea **radix directă** poate sorta n elemente cu chei de b biţi, în b/m treceri utilizând un spaţiu suplimentar pentru 2^m contoare şi 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 obţine o metodă de sortare foarte eficientă, cu condiţia să existe un spaţiu suplimentar de memorie cu 2^m locaţii.

* 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 distribuţiilor, 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 locaţii).

  * În consecinţă fiecare pas este liniar şi deoarece sunt necesari numai 8 (4) paşi pentru chei de 32 de biţi, 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 **distribuţiei uniforme** a cheilor.

  * (2) Necesitatea unor **spaţii suplimentare** de memorie pentru **tabloul de contoare** şi pentru **zona de sortare**.

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

* În situaţia 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]`.
    - Iniţial 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 construcţia sintactică `a[p[i]]` în loc de `a[i]`.
    - Accesul la `a[i]` prin `p[i]` se va realiza numai pentru **comparaţii**, 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 conţine 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`.

- Principial 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 secvenţa [3.2.10.a] se prezintă un algoritm care realizează **sortarea indirectă** bazată pe **metoda inserţiei** a unui tablou `a`.

  * Poziţia `0` a tablului `a[0]` este utilizată drept fanion.

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

tip_element a[n+1];
tip_indice p[n+1];

void insertie_indirecta()                    /*[3.2.10.a]*/
  {
    tip_indice i,j,v;

    /*iniţializarea tabloului de indici*/
    for(i=1;i<=n;i++) p[i]=i;
    /*sortare prin inserţie a tabloului de indici p*/
    for(i=2;i<=n;i++)
      {
        v=p[i]; a[0]=a[i];   /*a[0] este poziţia 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 excepţia atribuirii fanionului, accesele la tabloul `a` se realizează **numai** pentru comparaţii.

* În multe aplicaţii este suficientă numai obţinerea tabloului `p` nemaifiind necesară şi 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 reaşezare "**in situ**" din secvenţa [3.2.10.b].

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

void reasezare_in_situ()                     /*[3.2.10.b]*/
  {
      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_;
        }
  } /*reaşezare_in_situ*/
------------------------------------------------------------
```

- În cazul unor aplicaţii particulare, viabilitatea acestei tehnici depinde de lungimea relativă a cheilor şi articolelor.

    - Metoda descrisă **nu** se justifică pentru **articole de mici dimensiuni** deoarece necesită o zonă de memorie suplimentară pentru tabloul `p` şi timp suplimentar pentru comparaţiile 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 şi cu C şi M numărul de comparaţii respectiv de mişcări necesare în procesul de sortare.

- În figura 3.2.11 apar relaţiile analitice ale valorilor minime, medii şi maxime ale indicatorilor C şi M pentru trei dintre **metodele directe de sortare**, relaţii valabile pentru cele $n!$ permutări ale celor n elemente.

- Pentru metodele avansate de sortare, formule sunt mult mai complicate.

    - Este însă esenţial de reţinut că efortul de calcul în vederea sortării este $c_1 \cdot n^{1.2}$ în cazul tehnicii **shellsort** şi $c_1 \cdot n \cdot log(n)$ în cazul tehnicilor **heapsort** şi **quicksort**.

- Aceste formule care aproximează în mod grosier performanţele funcţie 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 proporţional cu n^2.

    - (2) **Tehnici avansate** sau **logaritmice** la care efortul este proporţional cu $n \cdot log(n)$.

Metodă		Min	Med	Max
Inserţie	C	$n-1$	$\frac{n^2+n-2}{4}$	$\frac{(n-1)\cdot n}{2}$
	M	$3\cdot(n-1)$	$\frac{n^2+11\cdot n-12}{4}$	$\frac{n^2+5\cdot n-6}{2}$
Selecţie	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. Relaţii referitoare la performanţele metodelor de sortare directe

- Analizele şi măsurătorile efectuate asupra metodelor de sortare prezentate în acest capitol, au permis evidenţierea următoarelor **concluzii** [Wi76]:

  - Beneficiile pe care le aduce **inserţia binară** faţă de **inserţia simplă** sunt nesemnificative şi chiar negative în cazul secvenţelor deja sortate.

  - **Tehnica bubblesort** este **cea mai puţin performantă** tehnică de sortare. Chiar versiunea ei îmbunătăţită **shakersort** este inferioară inserţiei sau selecţiei directe (mai puţin 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 informaţiei utile conţinute în element, în raport cu câmpul `cheie`, nu influenţează semnificativ performanţele relative ale tehnicilor prezentate.

- Cu toate acestea, pentru această situaţie se evidenţiază următoarele concluzii suplimentare:

  - Performanţele **selecţiei directe** se îmbunătăţesc, situând această tehnică pe primul loc în rândul metodelor directe.

  - **Tehnica bubbesort** este şi în acest caz cea mai puţin performantă, chiar pierde teren, iar varianta ei **shakersort** obţine performanţe ceva mai bune în cazul tablourilor ordonate invers.

  - **Tehnica quicksort** îşi întăreşte poziţiile ca fiind cea mai rapidă metodă şi 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 informaţii suplimentare** referitoare la cheile de sortat.

- În condiţiile în care se dispune de astfel de informaţii, respectiv cheile de sortat îndeplinesc anumite condiţii apriorice şi / sau se utilizează zone de memorie suplimentare renunţându-se la restricţia **in situ,** performanţele sortării pot fi îmbunătăţite.

    - În aceasta categorie pot fi incluse **tehnica binsort** care limitează domeniul cheilor şi varianta sa extinsă **sortarea bazată pe determinarea distribuţiei cheilor**, ambele tehnici apropiindu-se de performanţa liniară O*(n)*.

- În aceeaşi categorie pot fi incluse şi tehnicile de **sortare radix** cu variantele prin **interschimbare** şi **directă** ambele concurând puternic tehnica quicksort.

    - Utilizând tablouri suplimentare de dimensiuni corespunzătoare pentru memorarea distribuţiei cheilor, aceste metode se pot apropia ca şi performanţe 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 îmbunătăţesc substanţial performanţele.

- Ca şi o remarcă finală se reaminteşte faptul că această secţiune a abordat doar **sortarea structurilor de date tablou**, deşi unele dintre tehnicile prezentate pot fi aplicate şi altor tipuri de structuri.

3.3. Sortarea secvenţelor. 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 secvenţiale** cum ar fi **benzile magnetice** sau **discurile magnetice** în alocare secvenţială.

- În acest caz datele pot fi modelate cu ajutorul unei **structuri secvenţă** având drept **caracteristică esenţială** faptul că **accesul la componente** se realizează în manieră **strict secvenţială**.

    - Aceasta este o **restricţie** 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 secvenţelor** este **sortarea prin interclasare** ("**merging**").

3.3.1. Sortarea prin interclasare

- **Interclasarea** presupune **combinarea** a două sau mai multe secvenţe ordonate într-o singură secvenţă ordonată, prin selecţii repetate ale componentelor curent accesibile.

    - Interclasarea este o operaţie simplă, care este utilizată ca **auxiliar** în procesul mult mai complex al **sortării secvenţiale**.

- O **metodă de sortare** bazată pe **interclasare** a unei secvenţe `a` este următoarea:

    - (1) Se **împarte** secvenţa de sortat `a` în două jumătăţi `b` şi `c`.

    - (2) Se **interclasează** `b` cu `c`, combinând câte un element din fiecare, în **perechi ordonate** obţinându-se o nouă secvenţă `a`.

    - (3) Se **repetă** cu secvenţa interclasată `a`, paşii (1) şi (2) de această dată combinând perechile ordonate în **qvadruple ordonate**.

    - (4) Se **repetă** paşii iniţiali, interclasând qvadruplele în **8-uple**, ş.a.m.d, de fiecare dată **dublând** lungimea subsecvenţelor de interclasare până la sortarea întregii secvenţe.

- Spre exemplu fie secvenţa:

    a: 34 65 12 22 83 18 04 67

    - Execuţia pasului 1 conduce la două jumătăţi de secvenţă:

        b: 34 65 12 22

        c: 83 18 04 67

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

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

    - Înjumătăţind din nou:

        b: 34 83 | 18 65

        c: 04 12 | 22 67

    - Şi interclasând perechile în **qvadruple** se obţine:

        a: 04 12 34 83 | 18 22 65 67

    - Cea de-a treia înjumătăţire:

        b: 04 12 34 83

        c: 18 22 65 67

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

        a: 04 12 18 22 34 65 67 83

- Se observă faptul extrem de semnificativ că, **accesul** la componentele succesive ale secvenţelor se realizează în manieră **strict secvenţială.**

    - Fiecare operaţie care tratează întregul set de date se numeşte **fază.**

* Procesul prin repetarea căruia se realizează sortarea se numeşte **trecere**.

* Procesul de sortare anterior descris constă din 3 treceri fiecare cuprinzând 2 faze: o fază de **înjumătăţire** şi una de **interclasare**.

* Pentru a realiza sortarea sunt necesare trei secvenţe motiv pentru care sortarea se numeşte **interclasare cu trei secvenţe**.

    * Aceasta este de fapt o **interclasare neechilibrată cu 3 secvenţe**.

3.3.1.1. Interclasarea neechilibrată cu trei secvenţe

* **Interclasarea neechilibrată cu trei secvenţe** reprezintă implementarea procedeului de sortare precizat anterior.

* **Schema de principiu** a acestui procedeu apare în figura 3.3.1.1.

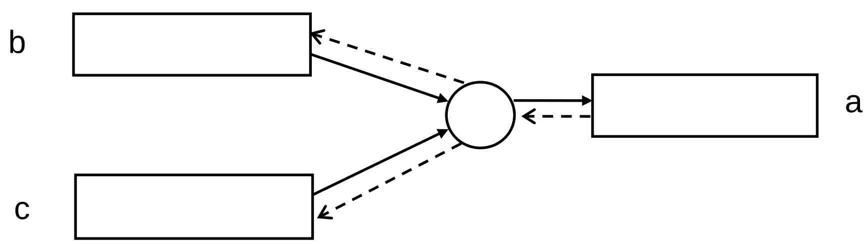


Fig. 3.3.1.1. Interclasare neechilibrată cu trei secvenţe

* Într-o primă etapă în secvenţele [3.3.1.1.a, b] se prezintă structurile de date şi schiţa de principiu a algoritmului.

```
------------------------------------------------------------
/*Interclasarea neechilibrată cu trei secvenţe - 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 secvenţe - Pas rafinare
0*/

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

```
  tip_secventa a,b,c;
  int p; /*p este dimensiunea n-uplu*/
  int k; /*k este numărul n-upluri interclasate*/

  p=1;
  repetă
    *înjumătăţire; /*distribuie n-uplele de pe a pe b şi c*/
    *interclasare; /*interclasează n-uplele de pe b şi 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** conţine două faze:
  - (1) O **fază de înjumătăţire** adică de **distribuţie** a *n*-uplelor secvenţei `a` pe cele două secvenţe `b` şi `c`.
  - (2) O **fază de interclasare** în care *n*-uplele de pe secvenţele `b` şi `c` se interclasează în *n*-uple de dimensiune dublă pe secvenţa `a`.
  - Variabila `p` iniţializată pe 1, precizează **dimensiunea** *n*-uplelor curente, dimensiune care după fiecare trecere se dublează.
    - În consecinţă **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 şi incrementală** a celor două faze.
- În secvenţa [3.3.1.1.c] apare primul pas de rafinare al fazei de **`Injumătăţire`**, iar în secvenţa următoare rafinarea enunţului "`scrie un` *n*`-uplu de dimensiune p în secvenţa d`".

```
/*Procedura Înjumătăţire - Pas rafinare 1*/

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

  tip_secventa a,b,c;

  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 secvenţa d
 citirea elementelor se face de pe secvenţa a*/

 tip_secventa a,d;
 tip_element x;
 int i,p;

 i=0; /*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 şi `p-1`.
  - Scrierea se **termină** la atingerea numărului `p` de elemente sau la terminarea secvenţei sursă.
- Rafinarea **fazei de interclasare** apare în secvenţa [3.3.1.1.e].

```
------------------------------------------------------------
/*Procedura Intercalsare - Pas rafinare 1*/

PROCEDURE Interclasare(int p, int k)      /*[3.3.1.1.e]*/
  /*p - dimensiune n-uplu, k - contor n-uple*/

 tip_secventa a,b,c;
 tip_element x,y;
 int k;

 Rewrite(a); Reset(b); Reset(c);
 k= 0;   /*k este contorul de n-uple interclasate pe a*/
 *iniţializare 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 şi 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 secvenţelor `b` şi `c`.

- Datorită particularităţilor de implementare a fişierelor sunt necesare câteva **precizări**:
  - (1) Variabila `Eof(f)` se poziţionează pe **`true`** la citirea ultimului element al fişierului f.
  - (2) Citirea dintr-un fişier cu `Eof` poziţionat pe **`true`** conduce la **eroare**.

        * (3) Din punctul de vedere al algoritmului de interclasare, terminarea prelucrării unui fişier **nu** coincide cu poziţionarea 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 şi momentul prelucrării unui element.

        * Astfel în fiecare moment se prelucrează elementul citit în **pasul anterior** şi se citeşte un **nou element**.

    * În acest scop pentru fiecare fişier implicat în prelucrare se utilizează:

        * (1) O variabilă specială de `TipElement` care **memorează** elementul curent.

        * (2) O variabilă booleană `EndPrelucr` a cărei valoare **`true`** semnifică **terminarea prelucrării** ultimului element al fişierului.

* Rafinarea enunţului “`interclasează câte un` *n*`-uplu de pe b şi c pe a şi incrementează pe k`” apare în secvenţa [3.3.1.1.f] care aplică tehnica anterior precizată.

* Variabilele specifice asociate secvenţelor `b` şi `c` sunt `x` şi `y` respectiv `EndPrelucr_b` şi `EndPrelucr_c` [3.3.1.1.g].

```
------------------------------------------------------------
/*interclaseaza câte un n-uplu de pe b si c pe a şi
incrementează pe k*/
                                          /*[3.3.1.1.f]*/
  tip_secventa a,b,c;
  tip_element x;   /*x - elementul curent al secvenţei b*/
  tip_element y;   /*y - elementul curent al secvenţei c*/
  int i,j,k;

  i=0;    /*contor n-uplu curent pe b*/
  j=0;    /*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);/*citeşte elementul următor din
 | |    |□                     b*/
 | |  altfel
 | |    |*scrie(a,y); j=j+1;
 | |    |daca Eof(c) EndPrelucr_c=true;/*y este ultimul elem
 | |    |  altfel                      din c*/
 | |    |    *citeste(c,y);/*citeste 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 secvenţa a*/
------------------------------------------------------------
```

* O variantă de implementare integrală a procedeului de **sortare neechilibrată cu 3 benzi** incluzând structurile de date aferente şi funcţiile implicate apare în secvenţa [3.3.1.1.g] iar programul integral în [3.3.1.1.h].

```
//Interclasare neechilibrată cu 3 secvenţe

#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 secvenţă
typedef struct tip_secventa
{
      tip_fisier fisier;
      tip_element elemCurent;
      boolean termPrelucr;

} tip_secventa;

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


// definirea funcţiilor referitoare la interclasarea neechilibrată

void ScrieNuplu(tip_secventa d)
// scrie un n-uplu de dimensiune p pe secvenţa d, citirea elementelor
se face de pe secvenţa 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 şi 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 şi c pe a şi
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); // citeşte următorul element din b
        }
        else
        {
            Write(&a.fisier,y);
            j=j+1;
            if(Eof(&c.fisier))
                EndPrelucr_c=true;
            else
```

```
                Read(&c.fisier,&y); // citeşte 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); // citeşte 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); // citeşte următorul element din c
    }

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

  }while(!EndPrelucr_b || !EndPrelucr_c);

}

// operatorul interclasare cu 3 secvenţe
void Interclasare3Secvente()
{
  p=1;
  do
  {
    Injumatatire(p);     // distribuie n-uplele de pe a pe b şi c
    Interclasare(p,&k); // interclasează n-uplele de pe b şi 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 procedeului de **sortare neechilibrată cu 3 benzi** apare în secvenţa [3.3.1.1.h]. Această variantă include pe lângă structurile de date aferente şi funcţiile implicate în sortare, operatorii referitori la procesarea fişierelor, funcţii de generare a numerelor aleatoare, funcţii de iniţializare şi generare de secvenţe utilizate în vederea creării iniţiale a secvenţelor de sortat precum şi programul principal.

```
------------------------------------------------------------------------
//Program Interclasare neechilibrată cu 3 secvenţe - variantă integrală
```

```c
#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 secvenţă
typedef struct tip_secventa
{
      tip_fisier fisier;
      tip_element elemCurent;
      boolean termPrelucr;

} tip_secventa;

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

// definirea operatorilor referitori la prelucrarea fişierelor
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.nume_fisier,
"w+");
}

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

// definirea funcţiilor referitoare la interclasarea neechilibrată

void ScrieNuplu(tip_secventa d)
// scrie un n-uplu de dimensiune p pe secvenţa d, citirea elementelor
se face de pe secvenţa 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 şi 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 şi c pe a şi
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);    // citeşte următorul element din b
          }
          else
          {
              Write(&a.fisier,y);
              j=j+1;
              if(Eof(&c.fisier))
                  EndPrelucr_c=true;
              else
                  Read(&c.fisier,&y); // citeşte 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);     // citeşte 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);     // citeşte următorul element din c
      }

      *k=*k+1; // incrementeaza contorul de n-uple interclasate pe
  secvenţa 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);
}

// functii generare numere aleatoare
void generare_numere_aleatoare(const char* nume_fisier, int dimensiune)
{
      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)
{
      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 iniţializare şi generare secvenţe
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);
}

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

3.3.1.2. Interclasarea echilibrată cu 4 secvenţe (două căi)

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

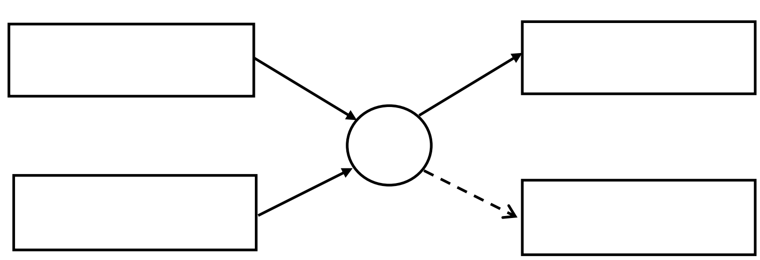


Fig. 3.3.1.2.a. Interclasare echilibrată cu patru secvenţe

- Într-o primă etapă, se va **modela** interclasarea echilibrată cu 4 secvenţe utilizând drept suport de modelare o **structură tablou**, care va fi parcurs în manieră strict secvenţială.

- Într-o etapă ulterioară, interclasarea va fi aplicată unor **structuri secvenţă**, permiţând compararea celor două abordări şi în acelaşi timp demonstrând puternica dependenţă a formei **algoritmului** în raport cu **structurile de date** pe care le utilizează.

- Un **tablou** poate **modela** două fişiere, dacă este privit ca o secvenţă cu două capete.

  - Astfel, în procesul de interclasare a celor două fişiere 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 `DESTINATIE`

  - Faza combinată înjumătăţire-interclasare apare reprezentată schematic în figura 3.3.1.2.b.

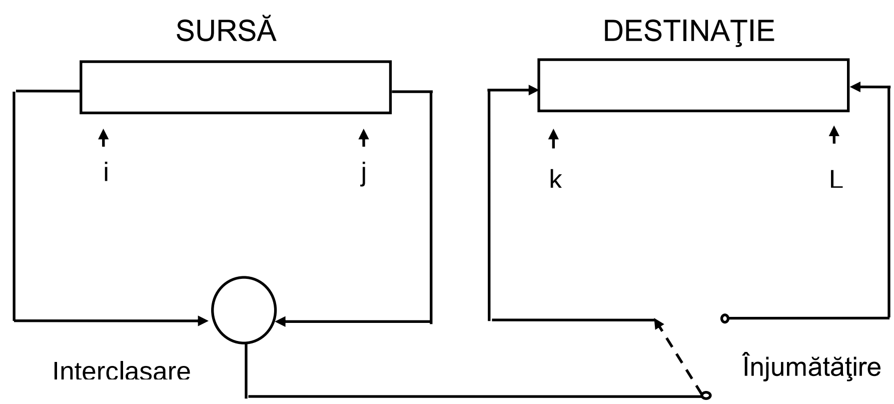


Fig. 3.3.1.2.b. Model pentru interclasarea echilibrată

  - Destinaţia articolelor interclasate este **comutată** după fiecare pereche ordonată la prima trecere, după fiecare quadruplu la a doua trecere, ş.a.m.d, astfel încât cele două secvenţe destinaţie, sunt de fapt cele două capete ale unui singur tablou.

  - După fiecare trecere cele două tablouri se **interschimbă** sursa devenind noua destinaţie şi 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` şi `j` precizează două elemente **sursă**, iar `k` şi `L` două **destinaţii**.

* **Secvenţa iniţială** va fi conţinută de prima parte a tabloului `a[0],...,a[n-1]`.

* Se introduce o **variabilă booleană** `sus` care precizează sensul mişcării elementelor:

    * De la **sursa** `a[0],...,a[n-1]` spre **destinaţia** `a[n],...,a[2*n-1]` când `sus` este **adevărat**.

    * De la **sursa** `a[n],...,a[2*n-1]` spre **destinaţia** `a[0],...,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 subsecvenţelor** ce urmează să fie interclasate.

    * `p` ia valoarea iniţială `1` şi se dublează după fiecare trecere.

    * Pentru a simplifica lucrurile se presupune momentan că `n` este o putere a lui `2`.

* În aceste condiţii, în secvenţa [3.3.1.2.a] apare **varianta pseudocod** a procedurii de interclasare iar în secvenţa [3.3.1.2.b] primul pas de rafinare.

```
------------------------------------------------------------
{Procedura Interclasare echilibrată cu 4 secvenţe - varianta
pseudocod}

PROCEDURE Interclasare                       {[3.3.1.2.a]}

  boolean  sus;  {indicator  boolean  pentru  sensul  mişcării
elementelor}
  int p;     {p este lungimea n-uplului curent}
  int n;     {n este numărul iniţial de elemente de sortat}


  sus=true;  p=1; {p este lungimea n-uplului curent}
  repeta
     daca sus atunci
         stânga este sursa; dreapta este destinaţie;
       altfel
         dreapta este sursa; stânga este destinatie;
     □{daca}
     *se interclasează n-uple de lungime p de la
       două capete  ale sursei, alternativ în cele
       două capete ale destinaţiei;
     sus=not sus;
     p=2*p;         {dublare lungime n-uplului curent}
  până când (p=n) {n este numarul iniţial de elemente de
                   sortat}
  □{repeta}
------------------------------------------------------------
{Procedura Interclasare echilibrată cu 4 secvenţe - Pasul 1
de rafinare}

PROCEDURE Interclasare                       {[3.3.1.2.b]}
```

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


  sus= true; p= 1;
  repetă {iniţializare indici}
    dacă sus
        i=0; j=n-1;     {stanga este sursă}
        k=n; L=2*n-1;   {dreapta este destinaţie}
        □
      altfel
        k=0; L=n-1;     {stanga este sursă}
        i=n; j=2*n-1;   {dreapta este destinaţie}
        □ {altfel}
    □{daca}
    *interclasează p-tuplele secvenţelor i şi j, alternativ
      în secvenţa k respectiv L;
    sus=NOT sus;
    p=2*p; {dublare lungime n-uplului curent}
  până când (p=n) {n- numarul iniţial de elemente de sortat}
  □{repeta}
{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 **treceri**:

    - Funcţie de variabila `sus` se asignează indicii sursă-destinaţie.

    - Se interclasează *p*-tuplele secvenţelor **sursă** în *p*-tuple de dimensiune dublă şi se depun în secvenţa **destinaţie**.

- În pasul următor al detalierilor succesive se rafinează, enunţul `"interclasează` *`p`*`-tuplele secvenţelor i şi j, alternativ, în secvenţa k respectiv L"`.

- Este clar că interclasarea a `n` elemente este la rândul ei o succesiune de interclasări parţiale ale unor subsecvenţe de lungime precizată `p`, în particular *p*-tuple, în subsecvenţe de lungime `2p`.

- După fiecare astfel de **interclasare** parţială, destinaţia este **comutată** de la un capăt al tabloului destinaţie la celălalt, asigurând astfel **distribuţia egală** a subsecvenţelor .

    - Dacă destinaţia elementului interclasat este capătul inferior al tabloului destinaţie, atunci `k` este indicele destinaţie şi valoarea sa este **incrementată** cu 1 după fiecare mutare.

    - Dacă mutările se execută la capătul superior, atunci `L` este indicele destinaţie şi valoarea sa va fi **decrementată** cu 1 după fiecare mutare.

* Pentru simplificare, **destinaţia** va fi precizată **tot timpul** de indicele `k`, intercomutând valorile lui `k` şi `L` după interclasarea fiecărui *p*-tuplu şi asignând pentru incrementul permanent `h` valoarea 1 respectiv -1.

* Rezultă următoarea rafinare [3.3.1.2.c]:

```
------------------------------------------------------------
{Rafinare enunţ: interclasează p-uplele secvenţelor i şi j,
alternativ în secvenţa k respectiv L}
                                                {[3.3.1.2.c]}
 int m; {m este numărul de elemente de interclasat}
 int n; {n este număr iniţial de elemente de sortat}
 int h; {h este incrementul curent al secvenţei destinaţie}
 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; {iniţializare increment destinaţie}
 m=n; {iniţializare număr de elemente de interclasat cu
         numărul iniţial de elemente de sortat}
 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 sortat}
   *interclasează q elemente de la sursa-i cu r elemente de
      la sursa-j; indicele destinaţie este k cu raţia h;
   h=-h; {comutare valoare increment destinaţie}
   *interschimbă indicii destinaţie (k şi L) pentru
    alternanţă;
 până când (m=0)
 □ {repeta}
------------------------------------------------------------
```

* Referitor la secvenţa [3.3.1.2.c] se precizează următoarele:

    * S-au notat cu `r` respectiv `q` **lungimile secvenţelor** 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.

        * Iniţial `m=n` şi el scade după fiecare interclasare a două subsecvenţe cu `2*p`.

    * Procesul de interclasare pentru o trecere se încheie când `m=0`.

* Pasul următor de detaliere rezolvă **interclasarea**.

    * Se precizează faptul că în situaţia în care procesul de interclasare parţială **nu** epuizează o subsecvenţă, restul rămas neprelucrat trebuie adăugat secvenţei destinaţie printr-o operaţie de copiere [3.3.1.2.d].

```
------------------------------------------------------------
{Rafinare enunţ: interclasează q elemente de la sursa i cu r
elemente de la sursa j; indicele destinaţie este k}
                                                  {[3.3.1.2.d]}
 tip_element a[2n]; {tablou secvenţe}
 int q;             {număr elemente secvenţa i}
 int r;             {număr elemente secvenţa j}
 int i,j;           {indici surse}
 int k;             {indice destinaţie}

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

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

    - În acest caz, se continuă interclasarea *p*-tuplelor până când secvenţele sursă care rămân au lungimea mai mică decât `p`.

    - Se observă uşor că singura parte a algoritmului care este influenţată de această situaţie este aceea în care se determină valorile lui `q` şi `r`, care sunt lungimile secvenţelor de interclasat din [3.3.1.2.c].

    - În consecinţă cele trei instrucţii:

        `q=p;  r=p;  m=m-2*p;`

    - Vor fi înlocuite cu:

        dacă `m>=p` **atunci** `q=p;` **altfel** `{q=m; m=m-q;}`

        dacă `m>=p` **atunci** `r=p` **altfel** `{r=m; m=m-r;}`

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

    - În plus, condiţia care controlează **terminarea algoritmului** `p= n`, trebuie modificată în `p ≥ n`.

- Varianta finală de refinare a procedurii **`Interclasare`** împreună cu structurile de date implicate apare în [3.3.1.2.e].

```
------------------------------------------------------------
// Implementare interclasare echilibrată cu 4 secvenţe 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 mişcării 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 si 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] şi 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 destinaţie (k şi l)
            h=-h; t=k; k=L; L=t;

        }while(m!=0);

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

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

    // mutare elemente sortate în secvenţa iniţială (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 secvenţe (2 căi)

- Deoarece la fiecare trecere `p` se dublează, îndeplinirea condiţiei `p>n` presupune $\lceil \log_2 n \rceil$ **treceri**, deci o **eficienţă** de ordinul O(*log* $_2$ *n)*.

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

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

    - **Numărul de comparaţii** C este chiar mai mic decât M deoarece operaţia de copiere a resturilor **nu** presupune nici o comparaţie.

- Cu toate astea, întrucât tehnica interclasării presupune utilizarea unor dispozitive periferice, efortul de calcul necesar **operaţiilor de mutare** poate domina efortul necesar **operaţiilor de comparare** cu mai multe ordine de mărime, motiv pentru care **analiza numărului de comparaţii nu** prezintă interes.

- În aparenţă **tehnica interclasării echilibrate** aplicată unor **structuri tablou**, obţine performanţe 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 performanţelor metodei **heapsort**, deci sub performanţele **quicksort**-ului.

3.3.2. Sortarea prin interclasare naturală

- **Tehnica de sortare prin interclasare nu** ia în considerare faptul că **datele iniţiale** pot fi **parţial sortate**, subsecvenţele interclasate având o lungime fixă predeterminată adică 2^k în trecerea k.
  - De fapt, oricare două subsecvenţe ordonate de lungimi m şi n pot fi interclasate într-o singură subsecvenţă ordonată de lungime $m+n$.
- **Tehnica de interclasare** care în fiecare moment combină **cele mai lungi secvenţe ordonate posibile** se numeşte **sortare prin interclasare naturală.**
  - În cadrul acestei tehnici un rol central îl joacă noţiunea de **monotonie**, care va fi clarificată pe baza următorului exemplu.
- Se consideră următoarea secvenţă de chei:

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

  - Se pun linii verticale la extremităţile secvenţei precum şi între elementele a_j şi a_{j+1}, ori de câte ori $a_j>a_{j+1}$.
  - În felul acesta secvenţa a fost defalcată în **secvenţe parţiale monotone**.
  - Secvenţele obţinute sunt de **lungime maximă**, adică **nu** pot fi prelungite fără a-şi pierde proprietatea de a fi **monotone**.
- În general fie $a_1, a_2, \ldots, a_n$ o **secvenţă** oarecare de numere întregi.
- **Formal**, se înţelege prin **monotonie** orice secvenţă parţială $a_i,\ldots,a_j$ care satisface următoarele condiţii [3.3.2.a]:

----

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

- Această definiţie include şi monotoniile cu un singur element, deoarece în acest caz $i=j$ şi proprietatea 2) este îndeplinită, neexistând nici un k cuprins între i şi $j-1$.
- **Sortarea naturală** este acea sortare care **interclasează monotonii**.
- **Sortarea naturală** se bazează pe următoarea **proprietate**:
  - Dacă se intercalează două secvenţe a câte n monotonii fiecare, rezultă o secvenţă cu exact n monotonii.
  - Ca atare, la fiecare trecere numărul monotoniilor se înjumătăţeşte şi în consecinţă **numărul maxim de treceri** pentru sortare va fi $\lceil \log_2 n \rceil$.
- În consecinţă, pentru procesul integral de sortare:
  - **Numărul de mişcări** este $n*\lceil \log_2 n \rceil$, în medie mai redus.

    * **Numărul de comparaţii** este însă mult mai mare deoarece pe lângă comparaţiile necesare interclasării elementelor sunt necesare comparaţii între elementele consecutive ale fiecărei secvenţe pentru a determina sfârşitul fiecărei monotonii.

* În continuare în dezvoltarea programului aferent acestei tehnici va fi utilizată aceeaşi **metodă a detalierilor succesive** (*”Stepwise refinement”*)

* Se utilizează o structură de date **fişier secvenţial** asupra căreia se aplică **modelul** de **sortare prin interclasare neechilibrată în două faze**, utilizând **trei secvenţe**.

* Algoritmul lucrează cu secvenţele `a,b` şi `c`.

    * Secvenţa `c` este cea care trebuie procesată şi care în final devine secvenţa sortată.

    * În practică, din motive de securitate, `c` este de fapt o copie a secvenţei iniţiale.

* 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 a,b,c;
------------------------------------------------------------
```

* Secvenţele `a` şi `b` sunt auxiliare şi ele servesc la defalcarea provizorie a lui `c` pe monotonii.

* **Fiecare trecere** constă din **două faze alternative** care se numesc **defalcare** respectiv **interclasare**.

    * În faza de **distribuire** monotoniile secvenţei `c` se distribuie alternativ pe secvenţele `a` şi `b`.

    * În faza de **interclasare** se recombină în `c`, monotoniile de pe secvenţele `a` şi `b` (fig. 3.3.2).

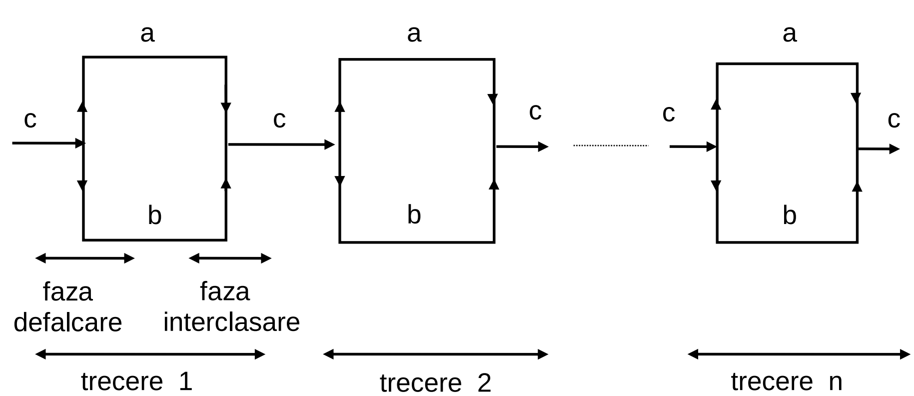


Fig. 3.3.2. Modelul sortării prin interclasare naturală. Treceri şi faze

- Sortarea se termină în momentul în care numărul monotoniilor secvenţei `c` devine egal cu 1.

- Pentru numărarea monotoniilor se utilizează variabila `L`.

- Prima formă a algoritmului de sortare naturală apare în secvenţa [3.3.2.c].

  - Cele două faze apar ca două **enunţuri**, care urmează să fie **rafinate** în continuare .

```
------------------------------------------------------------
{Sortarea prin interclasare naturală – Pas de rafinare 0}

PROCEDURE InterclasareNaturala                  {[3.3.2.c]}

  int L;                     {număr de monotonii interclasate}
  tip_secvenţă a,b,c;        {cele 3 secvenţe implicate}

  repetă
    Rewrite(a); Rewrite(b); Reset(c);
    Distribuire;
    Reset(a); Reset(b); Rewrite(c);
    L= 0;
    Interclasare;
  până când (L=1)
  □ {repeta}
{InterclasareNaturala}
------------------------------------------------------------
```

- **Procesul de rafinare** poate fi realizat:

  - (1) Prin **substituţia directă** a enunţurilor cu secvenţele care le corespund - proces cunoscut sub denumirea de **rafinare prin „tehnica inserţiei”**.

  - (2) Prin **interpretarea** enunţurilor **ca proceduri sau funcţii** şi procedând în consecinţă la dezvoltarea lor proces denumit **rafinare prin „tehnica selecţiei”**.

- În continuare se va continua procesul de rafinare prin **tehnica selecţiei**.

- În secvenţele [3.3.2.d] respectiv [3.3.2.e] apar primii paşi de rafinare pentru **`Distribuire`** respectiv **`Interclasare`**.

```
------------------------------------------------------------
{Sortarea prin interclasare naturală – rafinarea procedurii
Distribuire}

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

  tip_secvenţă a,b,c;   {cele 3 secvenţe implicate}

  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 secvenţele `a` respectiv `b`, dacă numărul de monotonii de pe secvenţa `c` este par.

    * (2) Cu o monotonie mai mult pe secvenţa `a`, dacă numărul de monotonii de pe secvenţa `c` este impar.

    * (3) După interclasarea monotoniilor 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 şi b pe c}

    tip_secvenţă a,b,c;     {cele 3 secvenţe implicate}
    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** şi **Interclasare** sunt redactate la rândul lor în termenii unor **proceduri subordonate** (**InterclasareMonotonie**, **CopiazaMonotonia**) care se referă la **o singură monotonie** şi care vor fi rafinate în continuare în [3.3.2.f] respectiv [3.3.2.g].

* Se introduce variabila booleană `sm` (sfârşit monotonie) care specifică dacă s-a ajuns sau **nu** la sfârşitul unei monotonii.

    * La epuizarea uneia dintre monotonii restul celeilalte este copiat în secvenţa destinaţie.

```
------------------------------------------------------------
{Sortarea prin interclasare naturală - rafinarea procedurii
CopiazaMonotonia}

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

  boolean sm;    {indicator sfârşit monotonie}

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

{Sortarea prin interclasare naturală - rafinarea procedurii InterclasareMonotonie}

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

    tip_secvenţă a,b,c;   {cele 3 secvenţe implicate}
    boolean sm;           {indicator sfârşit 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**(TipSecventa x,y), care transferă elementul curent al secvenţei sursă x în secvenţa destinaţie y, poziţionând variabila sm funcţie de atingerea sau **nu** a sfârşitului monotoniei.

- În acest scop se utilizează **tehnica** "**lookahead**" (tehnica scrutării), bazată pe citirea în pasul curent a elementului pentru pasul următor

    - În consecinţă, **primul element** de prelucrat se introduce în tamponul secvenţei înaintea demarării procesului de defalcare respectiv de interclasare.

- Pentru acest scop se modifică şi **structura de date** aferentă **secvenţei** după cum urmează [3.3.2.h].

{Sortarea prin interclasare naturală - rafinarea structurilor de date}

```
#define LUNGIME_MAXIMA_NUME_FISIER (256)   /*[3.3.2.h]*/

typedef int tip_element;

typedef struct
{
     char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
     FILE* file;
} tip_fisier;

typedef struct tip_secventa
{
   tip_fisier fisier;        /*fişier suport secvenţă*/
   tip_element elemCurent;   /*tamponul secvenţei*/
   boolean termPrelucr;      /*indicator terminare prelucrare
```

```
                                             secvenţă*/
} tip_secventa;
```

Procedura **CopiazaElement** apare în secvenţa [3.3.2.i].

```
{Sortarea prin interclasare naturală - rafinarea procedurii
CopiazaElement}

PROCEDURE CopiazaElement(tip_secvenţă x,y)   {[3.3.2.i]}

  {copiază un element de pe secvenţa x pe secvenţa y}

    tip_element aux;

    *scrie(y.secvenţa,x.elemCurent); {scrie elementul curent
                                       al lui x pe y}}
    dacă (Eof(x.secventa))  {x este ultimul element}
        sm= true;
        x.termPrelucr= true;
        □
      altfel
        aux= x.elemCurent; {salvare element curent}
        *citeste(x.secventa,x.elemCurent); {citire element
                                            următor}
        sm= aux.cheie>x.elemCurent.cheie;
        □ {altfel}
    □ {dacă}
{CopiazaElement}
```

- După cum se observă:
    - La momentul **curent** se scrie pe secvenţa destinaţie `y` elementul `x.elemCurent` citit în pasul anterior.
    - Dacă `x.elemCurent` a fost ultimul element al secvenţei, atunci prelucrarea s-a terminat.
    - Dacă `x.elemCurent` nu a fost ultimul element al secvenţei, el se salvează în variabila `aux` de tip `TipElement` şi se citeste elementul următor `x.elemCurent` în vederea determinării sfârşitului de monotonie `sm`.
- Desigur unii dintre paşii de rafinare precizaţi pot suferi anumite modificări, funcţie de natura secvenţelor reale care se utilizează şi de setul de operaţii disponibile asupra acestora.
- Din păcate, programul dezvoltat cu ajutorul acestei metode **nu** este corect în toate cazurile.
    - Spre exemplu, defalcarea secvenţei `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 consecinţă, datorită distribuţiei cheilor, formarea a 5 monotonii pe secvenţa `a` şi a unei singure monotonii pe secvenţa `b`, în loc de 5 cum era de aşteptat.

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 secvenţă 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 sfârşitul secvenţei `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 deşi procesul de distribuire repartizează în mod egal monotoniile pe secvenţele `a` respectiv `b`, **numărul real de monotonii** pe cele două secvenţe poate **diferi** foarte mult de **numărul preconizat,** datorită distribuţiei cheilor.

  - Pentru a **remedia** această situaţie, este necesar ca procedura **`Interclasare`** să fie **modificată** astfel încât, în momentul în care se ajunge la sfârşitul unei secvenţe, să copieze în `c` **tot** restul celeilalte secvenţe.

- 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;

typedef struct
{
       char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
       FILE* file;
} tip_fisier;

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 sfârşit monotonie
tip_secventa a,b,c;            //cele trei secvenţe

//copiază un element de pe secvenţa x pe secvenţa 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;
	}
}

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

//distribuie monotoniile de pe c alternativ pe a şi b
void Distribuire()
{
	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 şi 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 şi 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 secvenţei a
        if (!b.termPrelucr)
                Read(&b.fisier, &b.elemCurent); //primul element al secvenţei b

        //interclasează monotoniile de pe a şi 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 comparaţiilor de chei **nu** are importanţă practică, deoarece durata prelucrărilor în unitatea centrală a sistemului de calcul este neglijabilă faţă de durata acceselor la memoriile externe.

    - Din acest motiv **numărul mutărilor** M va fi considerat drept **unic** indicator de performanţă.

- În cazul **sortării prin intercalsare naturală**:

* La o **trecere**, în fiecare din cele două faze (defalcare şi interclasare) se mută toate elementele, deci **numărul de mişcări** $M = 2 \cdot n$.

* După fiecare trecere **numărul monotoniilor** se micşorează de două ori, uneori chiar mai substanţial, motiv pentru care a fost necesară şi modificarea anterioară a procedurii **`Interclasare`**.

* Ştiind că numărul de monotonii iniţiale este `n`, **numărul maxim de treceri** este $\lceil \log_2 n \rceil$, astfel încât în cel mai defavorabil caz **numărul de mişcări** $M=2 \cdot n \cdot \lceil \log_2 n \rceil$, în medie simţitor mai redus.

3.3.3. Sortarea prin interclasare multiplă echilibrată

* Întrucât efortul de sortare este proporţional cu **numărul de treceri**, o cale de **reducere,** respectiv **de creştere a eficienţei** a acestuia este aceea de a **distribui monotoniile** pe mai mult decât două secvenţe.

* În consecinţă:

    * Primul pas al interclasării a r **monotonii** care sunt distribuite pe `N` **secvenţe** conduce la r/N monotonii.

    * Al doilea pas conduce la r/N^2 monotonii.

    * Al treilea la r/N^3 monotonii ş.a.m.d.

* Această metodă de sortare se numeşte **interclasare multiplă-N**.

* **Numărul total de treceri** k, în cazul sortării a n secvenţe prin **interclasare multiplă-N** este $k = \lceil \log_N n \rceil$, iar **numărul total de mişcări** $M = n \cdot \lceil \log_N n \rceil$.

    * O modalitate de implementare a acestei situaţii 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 secvenţe de intrare şi de ieşire, monotoniile de pe primele fiind interclasate şi imediat redistribuite pe celelalte.

    * Dacă se utilizează `N` secvenţe (`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.

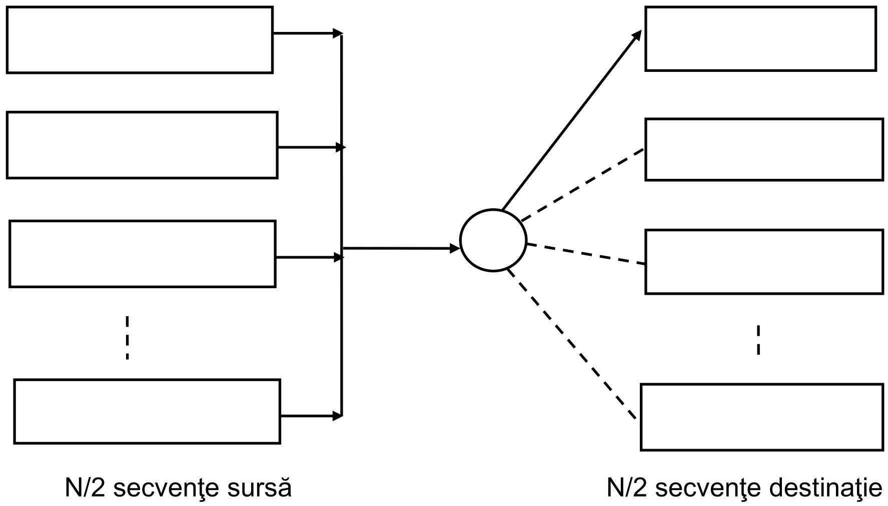


Fig. 3.3.3.a. Modelul interclasării multiple echilibrate cu N/2 căi

- **Algoritmul** care va fi dezvoltat în continuare se bazează pe o structură specifică de date şi anume, **tabloul de secvenţe**.

- Faţă de **procedeul interclasării multiple echilibrate cu 2 căi** utilizat anterior, trecerea la mai multe căi presupune **modificări esenţiale**:

  - (1) Procesul de interclasare trebuie să gestioneze o listă a **secvenţelor active**, din care va elimina pe rând secvenţele ale căror monotonii s-au epuizat.

  - (2) Trebuie implementată **comutarea** grupelor de **secvenţe de intrare** şi **ieşire** după fiecare trecere.

- Pentru aceasta se definesc structurile de date din secvenţa [3.3.3.a].

```
------------------------------------------------------------
/*Structuri de date pentru sortarea prin interclasare
multiplă echilibrată*/
                                           /*[3.3.3.a]*/

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

/*definire structură tip secvenţă*/
typedef struct tip_secventa
{
  tip_fisier fisier;        /*fişier support secvenţă*/
  tip_element elemCurent;   /*tampon secvenţă*/
  boolean termPrelucr;      /*indicator terminare procesare
                              secvenţă*/
} tip_secventa;
```

```
int Nr_Secvenţe=n;/*număr de secvenţe utilizate în sortare*/
tip_secventa f0;                   /*secvenţa iniţială*/
tip_secventa F[Nr_Secvenţe];       /*tablou de secvenţe*/
int t[Nr_Secvenţe],td[Nr_Secvenţe]; /*tablouri de indici*/
------------------------------------------------------------
```

- Se introduce **o nouă structură de date** şi anume **tabloul de secvenţe** `F` ale cărui elemente aparţin lui `Tip_Secvenţa`.

- Variabila `Nr_Secvenţe` memorează numărul de **secvenţe** utilizate în sortare.

- Se presupune de asemenea că secvenţa de elemente care urmează să fie sortată este furnizată ca o variabilă `f0` şi că pentru procesul de sortare se utilizează `Nr_Secvenţe` secvenţe (număr par).

- **Problema comutării secvenţelor** se poate rezolva introducând **tabloul de indici** `t` care are drept componente indici care identifică secvenţele.
  - Astfel în loc de a adresa o secvenţă din tabloul `F` direct prin indicele `i`, aceasta va fi adresată **via** tabloul `t`, respectiv `F[t[i]]` în loc de `F[i]`.
  - Iniţial `t[i]=i` pentru toate valorile lui `i`.
  - **Comutarea secvenţelor** constă de fapt în **interschimbarea** perechilor de componente ale tabloului `t` după cum urmează, unde `NH= Nr_Secvenţe/2`.

```
t[1]  <-> t[NH+1]
t[2]  <-> t[NH+2]
      ....
t[NH] <-> t[N]
```

- În continuare secvenţele `F[t[1]],...,F[t[NH]]` vor fi considerate **întotdeauna secvenţe de intrare**, iar secvenţele `F[t[NH+1]],...,  F[t[N]]` drept **secvenţe de ieşire** (fig.3.3.3.b).

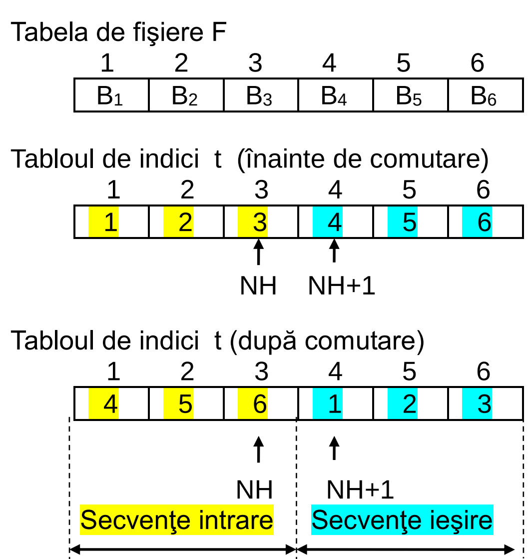


Fig. 3.3.3.b. Comutarea secvenţelor în interclasarea multiplă echilibrată

- Forma iniţială a algoritmului de **sortare prin interclasare multiplă echilibrată** este cea din secvenţa [3.3.3.b].

```
/*Sortarea prin interclasare multiplă echilibrată - varianta
iniţială*/

PROCEDURE SortareMultiplaEchilibrata;      /*[3.3.3.b]*/

    int i,j;  /*indici de lucru*/
    int NH;   /*NH= Nr_Secvenţe/2*/
    int L;    /*numărul monotoniilor distribuite*/
    tip_secventa f0; /*secvenţa iniţială*/
    tip_secventa F[Nr_Secvenţe];  /*tablou de secvenţe*/
    int t[Nr_Secvenţe],td[Nr_Secvenţe]; /*tablouri de
                                          indici*/
    NH=Nr_Secvenţe/2;
    j=NH; /*indice parcurgere secvenţe*/
    L=0;  /*iniţializare număr de monotonii distribuite*/

    repetă  /*se distribuie monotoniile iniţiale de pe f0 pe
             secvenţele de intrare t[1],...,t[NH]*/
      dacă j<NH j=j+1; altfel j=1;   /*parcurgere secvenţe*/
      *copiază o monotonie de pe f0 pe secvenţa F[j];
      L=L+1;
    până când SfarsitPrelucr(f0);
    □ /*repetă*/

    pentru i=1 la n  t[i]=i;  /*iniţializare tablou t*/

    repetă /*interclasează de pe secvenţele de intrare
    F[t[1]],...,F[t[NH]]    pe    secvenţele    de    ieşire
    F[t[NH+1]],...,F[t[N]]*/
      *iniţializare secvenţe de intrare;
      L=0;     /*iniţializare număr monotonii distribuite*/
      j=NH+1;  /*j este indicele secvenţei de ieşire*/
      repetă
        L=L+1;
        *interclasează câte o monotonie de pe fiecare
           intrare activă pe F[t[j]];
        dacă j<Nr_Secvenţe j=j+1; altfel j=NH+1;
      până când *toate intrările active au fost epuizate;
      □ /*repetă*/
      *comută secvenţele;
    până când L=1;
    □ /*repetă*/
   /*secvenţa sortată este t[1]*/
 /*SortareMultiplaEchilibrata*/
```

- După cum se observă, procesul de **sortare multiplă echilibrată** constă din **trei paşi**.

- (1) Primul pas realizează **distribuţia monotoniilor iniţiale** şi este materializat de prima buclă **`repetă`**. În acest pas:

    - Monotoniile de pe secvenţa iniţială `f0` sunt distribuite succesiv pe secvenţele sursă care sunt indicate de `j`.

- După copierea fiecărei monotonii, indicele `j` care parcurge ciclic domeniul `[1..NH]` este incrementat.
- Distribuţia se termină când se epuizează `f0`.
- Numărul de monotonii distribuite se contorizează în `l`.

- (2) Pasul al doilea realizează **iniţializarea** tabloului `t` (bucla **pentru**).
- (3)Pasul al treilea realizează **interclasarea** secvenţelor de intrare `F[t[1]]...F[t[NH]]` în secvenţele de ieşire `F[t[NH+1]]...F[t[N]]`.
- **Principiul** interclasării este următorul:
  - Se iau toate intrările active şi se interclasează câte o monotonie de pe fiecare într-o singură monotonie care se depune pe secvenţa `F[t[j]]`.
  - Se avansează la secvenţa de ieşire următoare, `j` parcurgând ciclic domeniul `[NH+1..N]`.
  - Se reia procedeul până la epuizarea tuturor intrărilor (bucla **repetă** interioară).
  - În acest moment se comută secvenţele, intrările devin ieşiri şi invers, şi se reia interclasarea.
  - Procesul continuă până când numărul de monotonii de interclasat ajunge egal cu 1 (bucla **repetă** exterioară).
- În continuare se prezintă **rafinarea** enunţurilor implicate în algoritm.
- În secvenţa [3.3.3.c] apare rafinarea enunţului `*copiază o monotonie de pe f0 în secvenţa F[j]`, utilizat în distribuţia iniţială a monotoniilor.

```
------------------------------------------------------------
/*rafinare enunţ: copiază o monotonie de pe secvenţa f0 pe
secvenţa 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
       *citeste(f0.fisier,f0.curent);  /*citeşte elementul
                                         următor*/
   □ /*dacă*/
   /*scrie elementul curent pe secvenţa F[j]*/
   *scrie(F[j].fisier,buf);
 până când ((buf.cheie>f0.curent.cheie) OR f0.termPrelucr);
 □ /*repetă*/
------------------------------------------------------------
```

- Urmează rafinarea enunţului `*iniţializare secvenţe de intrare`.

- Pentru început trebuiesc identificate secvenţele **curente** de intrare deoarece numărul **secvenţelor active** poate fi mai mic decât numărul total al secvenţelor sursă `NH`.

- Prin **secvenţă activă** se înţelege o secvenţă pe care mai există monotonii de interclasat.

- De fapt, **numărul secvenţelor** se reduce pe măsură ce se reduce **numărul monotoniilor**.

  - Practic **nu** pot exista mai multe secvenţe decât monotonii astfel încât sortarea se termină când mai rămâne **o singură secvenţă**.

- Se introduce variabila `k1` pentru a preciza **numărul actual de secvenţe** de intrare **active,** adică acelea care mai au monotonii.

- Cu aceste precizări, iniţializarea **secvenţelor de intrare** va fi [3.3.3.d]:

```
/*rafinare enunţ: iniţializare secvenţe intrare*/

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

- Datorită procesului repetat de interclasare, tendinţa lui `k1` este de a **descreşte** odată cu epuizarea monotoniilor, deci condiţia `*toate intrările epuizate` se poate exprima prin relaţia `k1=0`.

- Enunţul `*interclasează câte o monotonie de pe fiecare intrare pe F[t[j]]` este mai complicat.

  - El constă în selecţia repetată a elementului cu cea mai mică cheie dintre **sursele disponibile** şi trecerea lui pe **secvenţa de ieşire curentă**.

  - De la fiecare **secvenţă sursă** se parcurge câte **o monotonie** al cărei sfârşit poate fi atins fie drept consecinţă a uneia din următoarele două **condiţii**:

    - (1) **Epuizarea** secvenţei curente (eliminare secvenţă).

    - (2) **Citirea unei chei mai mici** decât cea curentă (închidere monotonie).

- În cazul (1) **secvenţa** este **eliminată** şi `k1` decrementat.

- În cazul (2) secvenţa este **exclusă temporar** de la selecţia cheilor, până când se termină crearea monotoniei curente pe secvenţa de ieşire, operaţie numită **"închidere monotonie".**

  - În acest scop se utilizează un al doilea indice `k2` care precizează **numărul** de **secvenţe sursă disponibile curent** pentru selecţia cheii următoare.

  - Valoarea sa iniţială egală cu `k1`, se decrementează ori de câte ori se epuizează o monotonie din cauza condiţiei (2).

  - Când `k2` devine `0`, s-a **terminat** interclasarea câte unei monotonii de la fiecare **intrare activă** pe **secvenţa de ieşire curentă**.

- Primul pas de rafinare al acestui enunţ apare în [3.3.3.e].

```
/*rafinare enunţ: interclasează câte o monotonie de pe
fiecare intrare pe F[t[j]]*/
                                          /*[3.3.3.e]*/
 int k1; /*numărul actual de secvenţe de intrare active*/
 int k2; /*numărul de secvenţe sursă care mai au monotonii*/

 pentru i=1 la k1  Reset(F[t[i]].fisier);
 k2= k1;
 repetă
   *selectează cheia minimă. Fie td[mx] indicele
     secvenţei care o conţine;
   buf= F[td[mx]].curent; /*tehnica look-ahead*/
   dacă(Eof(F[td[mx]].fisier))
       F[td[mx]].termPrelucr=true;
     altfel
       *citeste(F[td[mx].fisier,F[td[mx].curent);
   □ /*dacă*/
   *scrie(F[T[j]].fisier,buf);
   dacă (Eof(F[td[mx]].fisier))
       *elimină secvenţa;    /*detectare sfârşit secvenţă*/
     altfel
       dacă (buf.cheie>F[t[mx]].curent.cheie)
         *închide monotonia; /*detectare sfârşit monotonie*/
   □ /*dacă*/
 până când k2=0;
 □ /*repetă*/
------------------------------------------------------------
```

- Din păcate, introducerea lui `k2` **nu** este suficientă, deoarece pe lângă numărul secvenţelor disponibile trebuie **să se ştie exact** care sunt acestea.

- O soluţie în acest sens poate să o constituie utilizarea unui **tablou cu componente booleene** care indică disponibilitatea secvenţelor.

- O soluţie mai eficientă însă este utilizarea unui al doilea **tablou** de **indici** `td`. Indicii respectivi se referă la secvenţele de intraare care se procesează.

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

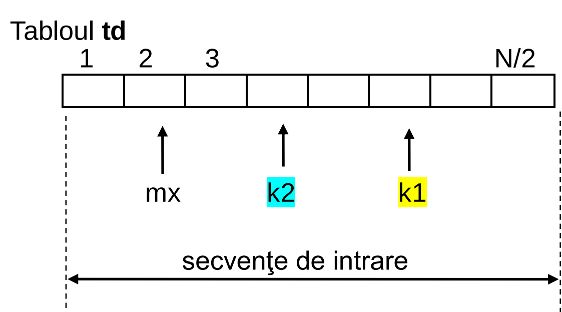


Fig. 3.3.3.c. Tabloul `td` utilizat ca auxiliar în procesul de interclasare

    - Acest tablou este utilizat **în locul tabloului** `t` pentru **secvenţele de intrare** astfel încât `td[1],...,td[k2]` sunt indicii **secvenţelor disponibile**.

    - Tabloul `td` se iniţializează la începutul fiecărei interclasări prin copierea indicilor secvenţelor de intrare din tabloul `t` (`t[1],...,t[k1]`).

- Indicele `k1` se iniţializează cu:
    - Valoarea `Nr_Secvente/2` dacă numărul de monotonii `L` este mai mare ca `Nr_Secvente/2`.
    - Valoarea curenta 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 secvenţe **active**.
    - Restul secvenţelor, (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 iniţializează cu valoarea lui `k1`, precizează **numărul de secvenţe active** care mai **au monotonii** în trecerea curentă.
    - Secvenţele cuprinse între indicii `k2` şi `k1` şi-au epuizat monotoniile în trecerea curentă **fără** a fi însă epuizate fizic.

- În aceste condiţii, **rafinarea enunţului** `*închide monotonia` corespunzător condiţiei (2), se implementează după cum urmează.
    - Fie `mx` indicele secvenţei pentru care s-a terminat monotonia curentă:
        - (1) Se **interschimbă** în tabloul `td` poziţia `mx` cu poziţia `k2`.
        - (2) Se decrementează `k2`.

- În ceea ce priveşte enunţul `*elimina secvenţa`, corespunzător condiţiei (1), considerând ca tot `mx` este indicele secvenţei care s-a epuizat, rafinarea presupune următoarele. Se menţionează că epuizarea fizică a secvenţei presupune şi închiderea ultimei monotonii.
    - (1) Se mută în tabloul `td` secvenţa precizată de `k2` în locul secvenţei precizate de `mx` (închidere monotonie).
    - (2) Se mută secvenţa precizată de `k1` în locul secvenţei indicate de `k2` (eliminare secvenţă).
    - (3) Se decrementează `k1`.
    - (4) Se decrementează `k2`.
    - (5) Toate operaţiile se reiau similar şi pentru tabloul `t`.

- Forma finală a **sortării multiple echilibrate** apare în secvenţa [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 fişier
typedef struct
{
       char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
       FILE* file;
} tip_fisier;

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

// definire operatori referitori la prelucrarea secvenţelor
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 = sursa
        // t = destinatie
        int t[Nr_Secvente + 1], td[Nr_Secvente + 1]; // tablouri de indici
        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

        do // distribuirea monotoniilor inițiale pe t[1], ..., t[NH]
        {
                if (j < NH) j = j + 1;   // parcurgere secvențe
                  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];
                        Read(&F[td[i]].fisier, &F[td[i]].elemCurent);
                }
```

```
   L = 0;      // numărul de monotonii interclasate
   j = NH + 1; // j este indexul secvenţei de ieşire

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

   do // interclasează câte o monotonie de pe t[1], ...t[k2] pe t[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] conţine 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 secvenţa
                       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
                                             secvenţă destinaţie
          else j = NH + 1;
   }
   while (k1 != 0); // toate intrările au fost epuizate

   for (i = 1; i <= NH; i++) // comută secvenţele
   {
          tx = t[i];
          t[i] = t[i + NH];
          t[i + NH] = tx;
   }
```

```
        }
        while (L != 1); // fişierul sortat se găseşte 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

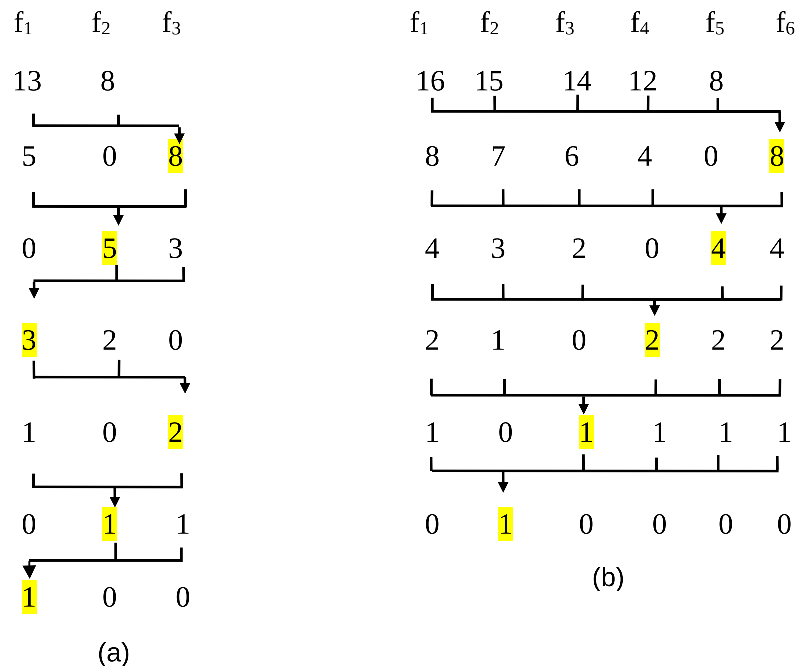
- **Metoda interclasării echilibrate** contopeşte operaţiile de distribuire şi interclasare într-o aceeaşi fază, utilizând mai multe secvenţe de intrare şi mai multe secvenţe de ieşire, care **nu** sunt folosite în totalitate.

- R.L. Gilstad, a propus **metoda sortării polifazice** care înlătură acest neajuns.
    - În plus în cadrul acestei metode însăşi noţiunea de trecere devine difuză.

- Pentru început se consideră un **exemplu** cu trei secvenţe.
    - În fiecare moment, sunt interclasate monotoniile de pe două secvenţe pe cea de-a treia.
    - Ori de câte ori una din secvenţele sursă se epuizează, ea devine imediat destinaţia operaţiei de interclasare a celorlalte două secvenţe (secvenţa neterminată şi fosta destinaţie).
    - Procesul se termină când rămâne o singură monotonie.

- După cum se cunoaşte, din procesul de interclasare a `n` monotonii de pe fiecare dintre secvenţele de intrare, rezultă `n` monotonii pe secvenţa de ieşire.
    - În cadrul acestei metode, pentru simplificare, se va vizualiza numai **numărul de monotonii** de pe fiecare secvenţă în locul cheilor propriu-zise.

- Astfel, în fig. 3.3.4.a (a) se presupune că secvenţele de intrare `f1` şi `f2` conţin 13 şi respectiv 8 monotonii.
    - La prima "*trecere*", sunt interclasate de pe `f1` şi `f2` pe `f3` 8 monotonii,
    - La a doua "*trecere*" sunt interclasate de pe `f1` şi `f3` pe `f2` cele 5 monotonii rămase, etc.
    - În final `f1` este secvenţa sortată.



  - În aceeaşi figură (b) se prezintă un exemplu de sortare polifazică a 65 de monotonii pe 6 secvenţe.

- Această tehnică este **mai eficientă** decât interclasarea echilibrată deoarece, fiind date `N` secvenţe, ea lucrează la interclasare cu `N-1` secvenţe 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 operaţiei de interclasare** (numărul de secvenţe sursă).

- În exemplele prezentate însă, **distribuţia iniţială a monotoniilor** a fost aleasă cu mare grijă.

- Pentru a determina care dintre distribuţiile iniţiale ale monotoniilor asigură o funcţionare corespunzătoare a algoritmului, se alcătuiesc tabelele din figurile 3.3.4.b şi 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.

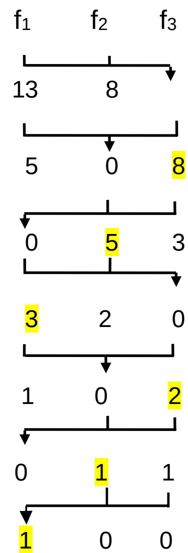
  - Trecerile se parcurg de jos în sus în figura 3.3.4.a, iar tabela se completează în ordine inversă, adică de sus în jos.

  - Fiecare rând din tabel, cu excepţia ultimului, conţine pe prima poziţie (a_1), **numărul de monotonii** ale **secvenţei destinaţie** 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 secvenţa destinaţie) şi se consideră circulară.

* Ultimul rând din tabel conţine **situaţia iniţială** a distribuţiei monotoniilor adică cea dinaintea demarării procesului de sortare.



k	$a_1^{(k)}$	$a_2^{(k)}$	$\sum a_i^{(k)}$
0	1	0	**1**
1	1	1	**2**
2	2	1	**3**
3	3	2	**5**
4	5	3	**8**
5	8	5	**13**
6	13	8	**21**

Fig. 3.3.4.b. Distribuţia perfectă a monotoniilor pe trei secvenţe

* Din tabloul din figura 3.3.4.b se pot deduce următoarele relaţii [3.3.4.a]:

$$a_2^{(k+1)} = a_1^{(k)}$$

$$a_1^{(k+1)} = a_1^{(k)} + a_2^{(k)} = a_1^{(k)} + a_1^{(k-1)} \quad \text{pentru} \quad k > 0 \qquad [3.3.4.a]$$

$$\text{unde} \quad a_1^{(0)} = 1 \quad \text{şi} \quad a_2^{(0)} = 0$$

----

* Dacă facem $a_1^{(i)} = f_i^{(1)}$ prin înlocuire rezultă [3.3.4.b]:

$$f_{i+1}^{(1)} = f_i^{(1)} + f_{i-1}^{(1)} \qquad \text{pentru} \quad i \geq 1$$

$$f_1^{(1)} = 1 \qquad [3.3.4.b]$$

$$f_0^{(1)} = 0$$

----

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

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

  * Adică, fiecare număr din acest şir este **suma celor doi predecesori** ai săi.

* În **consecinţă,** pentru cazul **sortării cu trei secvenţe**:

  * **Numerele iniţiale de monotonii** pe cele două secvenţe trebuie să fie **două numere Fibonacci de ordinul 1** consecutive.

- **Numărul total iniţial 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 obţin 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. Distribuţia perfectă a monotoniilor pe 6 secvenţe

----

$$a_5^{(k+1)} = a_1^{(k)}$$
$$a_4^{(k+1)} = a_1^{(k)} + a_5^{(k)} = a_1^{(k)} + a_1^{(k-1)}$$
$$a_3^{(k+1)} = a_1^{(k)} + a_4^{(k)} = a_1^{(k)} + a_1^{(k-1)} + a_1^{(k-2)} \qquad [3.3.4.c]$$
$$a_2^{(k+1)} = a_1^{(k)} + a_3^{(k)} = a_1^{(k)} + a_1^{(k-1)} + a_1^{(k-2)} + a_1^{(k-3)}$$
$$a_1^{(k+1)} = a_1^{(k)} + a_2^{(k)} = a_1^{(k)} + a_1^{(k-1)} + a_1^{(k-2)} + a_1^{(k-3)} + a_1^{(k-4)}$$

- Dacă facem $a_1^{(i)} = f_i^{(4)}$ rezultă [3.3.4.d] adică numerele **Fibonacci de ordinul 4**.

$$f_{i+1}^{(4)} = f_i^{(4)} + f_{i-1}^{(4)} + f_{i-2}^{(4)} + f_{i-3}^{(4)} + f_{i-4}^{(4)} \quad \text{pentru} \quad i \geq 4$$
$$\text{unde} \quad f_4^{(4)} = 1\ , \quad f_i^{(4)} = 0 \quad \text{pentru} \quad i < 4 \qquad [3.3.4.d]$$

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

$$f_{i+1}^{(p)} = f_i^{(p)} + f_{i-1}^{(p)} + \cdots + f_{i-p}^{(p)} \quad \text{pentru} \quad i \geq p$$
$$\text{unde} \quad f_p^{(p)} = 1\ , \quad f_i^{(p)} = 0 \quad \text{pentru} \quad 0 \leq i < p \qquad [3.3.4.e]$$

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

- În cazul **sortării polifazice** cu `n` secvenţe, **numerele iniţiale de monotonii** de pe cele $n-1$ secvenţe sursă sunt:
    - O **sumă** de $n-1$ numere Fibonacci consecutive de ordinul $n-2$ pe prima secvenţă.
    - O **sumă** de $n-2$ numere Fibonacci consecutive de ordinul $n-2$ pe a doua secvenţă.
    - O **sumă** de $n-3$ numere Fibonacci consecutive de ordinul $n-2$ pe a treia secvenţă.

* Ş.a.m.d.
* Pe ultima secvenţă (cea numerotată cu $n-1$) trebuie să existe iniţial 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 secvenţei al cărei **număr iniţial de monotonii** este **suma** a $n-1$ de astfel de **sume Fibonacci**.
  * O astfel de distribuţie a monotoniilor iniţiale se numeşte **distribuţie perfectă.**

* În cazul sortării polifazice cu 6 secvenţe ($n=6$), sunt necesare numere Fibonacci de ordinul $n-2=4$:

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

* **Distribuţia iniţială** a monotoniilor, prezentată în fig. 3.3.4.a, se stabileşte după cum urmează, pornind de la numărul Fibonacci **8** (cel de-al 9-lea din şir):
  * Secvenţa f_1 va conţine un număr de monotonii egal cu suma a $n-1=5$ numere Fibonacci consecutive: 1 + 1 + 2 + 4 + 8 = 16.
  * Secvenţa f_2 va conţine un număr de monotonii egal cu suma a $n-2=4$ numere Fibonacci consecutive: 1 + 2 + 4 + 8 = 15.
  * Secvenţa f_3, o sumă de $n-3=3$ numere: 2 + 4 + 8 = 14.
  * Secvenţa f_4, o sumă de $n-4=2$ numere: 4 + 8 = 12.
  * Secvenţa f_5 , o sumă de $n-5=1$ numere Fibonacci de ordinul 4, adică 8 monotonii.

* În total **secvenţa iniţială** care urmează să fie sortată trebuie să conţină :

  16 + 15 + 14 + 12 + 8 = 65 monotonii

* Aceste monotonii se distribuie iniţial pe cele 5 secvenţe sursă în concordanţă cu numerele determinate anterior realizându-se astfel o **distribuţie perfectă**.

* Se observă simplu că în fig. 3.3.4.c, **distribuţia monotoniilor** pentru **fiecare nivel** k se obţine aplicând acelaşi algoritm, alegând ca bază, numere consecutive din şirul Fibonacci 1, 1, 2, 4, 8, ... pentru respectiv k = 1, 2, 3, 4, 5, ...

* Dacă numărul iniţial de monotonii **nu** satisface condiţia 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 **distribuţiei iniţiale a monotoniilor**, iar apoi cea a **distribuţiei monotoniilor fictive.**

* Este evident faptul că selecţia unei **monotonii fictive** de pe secvenţa `i` înseamnă că secvenţa respectivă este ignorată în timpul interclasării curente, rezultând o interclasare de pe mai puţin de $n-1$ secvenţe sursă.

    - Interclasarea unei **monotonii fictive** de pe toate cele $n-1$ secvenţe sursă **nu** conduce la nici o interclasare efectivă ci doar la înregistrarea unei monotonii fictive pe secvenţa de ieşire.

    - Din acest motiv, este necesar ca monotoniile fictive să fie **distribuite** cât mai uniform posibil pe cele $n-1$ secvenţe.

- În primul rând se va analiza **problema repartizării** unui **număr necunoscut de monotonii** pe $n-1$ secvenţe în vederea obţinerii **distribuţiei 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 porneşte cu distribuţia 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 $k=3$, ş.a.m.d.

- Indexul k al rândului se numeşte **nivel**.

    - Pe măsură ce creşte numărul monotoniilor, creşte şi nivelul numerelor Fibonacci, nivel care în acelaşi timp precizează şi **numărul de treceri** sau comutări de secvenţe 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. Distribuţia perfectă a monotoniilor pe 6 secvenţe (reluare)

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

    - (1) Fie scopul distribuţiei, numerele lui Fibonacci de ordin $n-2$ nivelul `1`.

    - (2) Se realizează distribuţia monotoniilor conform acestui scop.

    - (3) Odată scopul atins, se calculează **nivelul următor** de numere Fibonacci. **Diferenţa** dintre acestea şi numerele corespunzătoare ale nivelului anterior constituie noul **scop** al distribuţiei.

    - (4) Se revine la pasul 2.

    - (5) Algoritmul se termină la epuizarea monotoniilor.

- Regulile după care se calculează următorul nivel al numerelor lui Fibonacci se bazează pe definiţia acestora [3.3.4.e].

- Atenţia 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 secvenţe.

    - În acest pas apar din nou în discuţie **monotoniile fictive**.

- Se presupune că la trecerea la nivelul următor, **scopul** următor va fi înregistrat prin diferenţele d_i pentru `i=1,2,...,n-1`, unde d_i precizează numărul de monotonii care trebuie depus pe secvenţa `i` în acest pas.

- Se **presupune** în continuare că se pun iniţial d_i **monotonii fictive** pe fiecare secvenţă `i`.

- În consecinţă, distribuirea **monotoniilor reale** poate fi privită ca şi o **reînlocuire a monotoniilor 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 secvenţa `i`.

- Nu se cunoaşte algoritmul care conduce la **distribuţia optimă**, dar cel propus de **Knuth**, numit **distribuţie orizontală** este foarte bun [Kn76].

- Termenul de **distribuţie orizontală** poate fi înţeles 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.

- Pentru a ajunge cât mai rapid la o distribuţie egală a monotoniilor fictive, se procedează la **înlocuirea** lor cu monotonii actuale.

    - Acest proces reduce dimensiunea silozurilor prin **extragerea monotoniilor fictive** din nivelurile orizontale şi **înlocuirea lor cu monotonii reale**, de la **stânga la dreapta**.

    - În acest fel monotoniile sunt distribuite pe secvenţe după cum indică numărul lor de ordine fig. 3.3.4.d.

    - Se precizează că în această figură este reprezentată ordinea de distribuţie a monotoniilor când se trece de la **nivelul 4** (`k=4`) conţinând 33 de monotonii la **nivelul 5** (`k=5`) conţinând 65 de monotonii.

    - Suprafeţele haşurate 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 iniţiale, toate monotoniile numerotate cu 54 şi mai mult vor fi tratate ca fictive.

    - Monotoniile se înscriu în realitate la sfârşitul secvenţei, dar este mai avantajos să ne imaginăm că se scriu la început, deoarece în procesul sortării se presupune că monotoniile fictive sunt la începutul secvenţei.

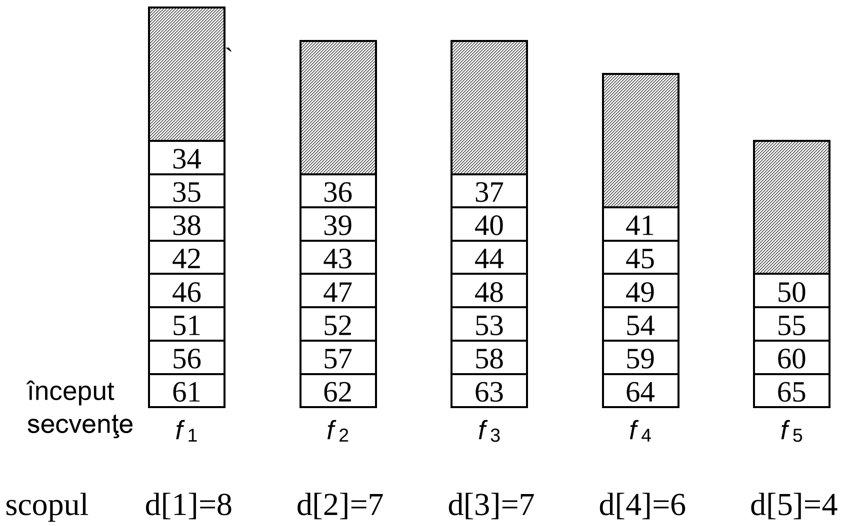


Fig.3.3.4.d. Distribuţia orizontală a monotoniilor

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 selecţia **noii secvenţe** pe care se va copia următoarea monotonie ţinând cont şi de **distribuţia perfectă** a monotoniilor pe secvenţele sursă.

    - Se presupune că variabila `j` este indexul secvenţei curente de destinaţie. Se utilizează tipurile de date definite în [3.3.4.g]:

```
------------------------------------------------------------
/*Sortarea polifazică - structuri de date*/

/*definire structură fişier*/
typedef struct                                [3.3.4.g]
{
   char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
   FILE* file;
} tip_fisier;

/*definire structură secvenţă*/
typedef struct tip_secventa
{
     tip_fisier fisier;        /*fişier support secvenţă*/
     tip_element elemCurent;   /*tampon element secvenţă*/
     boolean termPrelucr;      /*indicator terminare
                                 prelucrare secvenţă*/
} tip_secventa;

int n;                         /*număr de secvenţe*/
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 secvenţei destinaţie curente*/
int nivel;          /*nivelul curent de distribuţie*/
```

- Tablourile a şi d memorează **numerele de distribuţii ideale** respectiv **fictive** corespunzătoare fiecărei secvenţe i .
  - Aceste tablouri se iniţializează cu valorile $a_i=1$, $d_i=1$ pentru $i = 1,...,n-1$ respectiv $a_n=0$, $d_n=0$.
  - Variabilele j şi nivel se iniţializează cu valoarea 1.

- Procedura **SelecteazaSecventa** va calcula de fiecare dată când creşte nivelul, valorile **rândului următor** al tabelei din figura 3.3.4.c, respectiv valorile $a_1^{(k)},...,a_{n-1}^{(k)}$.
  - Tot atunci se calculează şi diferenţele $d_i=a_i^{(k)}-a_i^{(k-1)}$, care reprezintă **scopul următor**.
  - Algoritmul se bazează pe faptul că valorile d_i descresc odată cu creşterea 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 secvenţa j [3.3.4.h].

```
------------------------------------------------------------
/*Sortarea polifazică - procedura SelecteazăSecvenţa*/

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 secvenţa sursă f0 pe secvenţa F[j], faza iniţială de distribuire a monotoniilor poate fi schiţată astfel [3.3.4.i]:

```
------------------------------------------------------------
/*Sortarea polifazică - faza iniţială de distribuţie a
monotoniilor - pasul de rafinare 0*/
```

```
repeta
  SelecteazaSecventa;                                [3.3.4.i]
  CopiazaMonotonia
pana cand (f0.termPrelucr)
□ /*repeta*/
```

- Spre deosebire de interclasarea naturală, în cea polifazică este necesar să se cunoască **numărul exact de monotonii** de pe fiecare secvenţă, deoarece procesul de interclasare poate conduce la diminuarea numărului de monotonii.

- Pentru aceasta se va reţine cheia ultimului element al ultimei monotonii de pe fiecare secvenţă.

    - În acest scop se introduce variabila tablou `ultim[NrSecventa]`cu elemente de `TipCheie`.

- Următoarea rafinare a algoritmului de distribuţie este cea din [3.3.4.j]:

```
/*Sortarea polifazică – faza iniţială de distribuţie a
monotoniilor – pasul de rafinare 1*/

  tip_cheie ultim[n+1]; /*reţine cheia ultimului element al
                 ultimei monotonii de pe fiecare secvenţă*/

  repeta                                            [3.3.4.j]
    SelecteazaSecventa;
    daca(ultim[j]<= f0.curent.cheie)
      *continuă vechea monotonie;
    CopiazaMonotonia;
    ultim[j]= f0.curent.cheie;
  pana cand (f0.termPrelucr)
  □ /*repeta*/
```

- O problemă evidentă este aceea că `ultim[j]` se poziţionează numai după copierea primei monotonii, motiv pentru care la început distribuţia monotoniilor trebuie realizată fără inspectarea lui `ultim[j]`.

- Restul monotoniilor se distribuie conform secvenţei [3.3.4.k]. Se consideră că asignarea lui `ultim[j]` se realizează în procedura **CopiazaMonotonia**.

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

  while (NOT f0.termPrelucr)                         [3.4.4.k]
    SelecteazaSecventa;
    if(ultim[j]<=f0.curent.cheie)
        /*continuă vechea monotonie*/
        CopiazaMonotonia;
        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 şi include:
  - O buclă exterioară care **interclasează monotoniile** şi se execută până când se epuizează sursele.
  - O buclă cuprinsă în cea anterioară care **interclasează o singură monotonie** de pe fiecare secvenţă sursă.
  - Şi în sfârşit o a treia buclă cuprinsă în precedenta, care **selectează cheile** şi transmite elementele implicate spre secvenţa destinaţie.

- Se remarcă următoarele **diferenţe**:
  - Avem o singură secvenţă destinaţie (în loc de $N/2$).
  - În loc de a comuta $N/2$ secvenţe la fiecare trecere, acestea sunt rotite, utilizând un tablou `t` al indicilor secvenţelor.
  - Numărul de secvenţe 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 toţi `i`, atunci vor fi "*pseudo-interclasate*" $n-1$ monotonii fictive într-o monotonie fictivă, ceea ce presupune doar incrementarea contorului d_n al secvenţei destinaţie.
    - Altfel, se interclasează câte o monotonie de pe toate secvenţele care au $d_i=0$, iar pentru restul secvenţelor, d_i se decrementează indicând faptul că a fost luată în considerare o monotonie fictivă.
    - Se notează cu `k1` numărul secvenţelor implicate într-o interclasare.
  - Din cauza monotoniilor fictive, terminarea unei faze **nu** poate fi dedusă din condiţia de sfârşit a vreuneia din secvenţele implicate, deoarece existenţa unor monotonii fictive pe această secvenţă face necesară continuarea interclasării.
  - În schimb se poate determina uşor numărul teoretic de monotonii din coeficienţii a_i.
  - Aceşti coeficienţi $a_i^{(k)}$ care au fost calculaţi în timpul fazei de distribuţie vor fi **recalculaţi** în sens invers (backward) la interclasare.

- Cu aceste observaţii interclasarea propriu-zisă se poate formula după cum urmează [3.3.4.l]. Se presupune că:
  - Toate cele $n-1$ secvenţe care conţin monotoniile iniţiale sunt resetate.
  - Tabloul indicilor secvenţelor este poziţionat conform relaţiei `t[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 determ. nr k1 al secventelor 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 interclaseaza câte o monotonie de pe
           F[t[1]],...,F[t[k1]];
    z=z-1
  until(z=0)
  □
  Reset(F[t[n]]);
  *roteşte secvenţele în tabloul t. Calculează a[i]
      pentru nivelul urmator;
  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 diferenţa că algoritmul de eliminare al secvenţelor este mai simplu.

- Rotirea indicilor secvenţelor în tabloul indicilor, a indicilor d_i corespunzători, precum şi calcularea valorilor coeficienţilor 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 fişier
typedef struct
{
       char nume_fisier[LUNGIME_MAXIMA_NUME_FISIER];
       FILE* file;
} tip_fisier;

//definire structură secvenţă
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.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;
}

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* nume_fisier, int dimensiune)
{
	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 InitializareUnFisier(tip_secventa* secventa, char* nume_fisier)
{
	secventa->termPrelucr = false;
```

```
        strcpy(secventa->fisier.nume_fisier, nume_fisier);
        secventa->fisier.file = fopen(nume_fisier, "w+");
}

//definire structuri de date specifice
#define n 3             //număr de secvenţe
tip_secventa F[n + 1]; //tablou de secvenţe
int a[n + 1];           //tablou cu numărul ideal de monotonii
int d[n + 1];           //tablou cu numărul de monotonii fictive
int j = 1;              //indexul secvenţei destinaţie curente
int nivel = 1;          //atât j, cât şi nivel se iniţializează cu valoarea 1
boolean eob;
tip_element buf;
tip_secventa f0;        //secvenţa iniţială f0

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

void initializareSecvente() //iniţializează fişierele 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() //initializează tablourile a şi 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 secvenţa 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 conţinutul secvenţei
```

```
{
    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 secvenţele pentru distribuţia
                                monotoniilor iniţiale
{
    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 distribuţia (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 secvenţelor 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] conţine 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 secvenţă va conţine 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 evidenţierea performanţelor acestora este dificilă.

- Se pot formula însă următoarele **observaţii**:
    - (1) Există o **legătură indisolubilă** între un anumit **algoritm** care rezolvă o anumită problemă şi **structurile de date** pe care acesta le utilizează, influenţa celor din urmă fiind uneori decisivă pentru algoritm.
        - Acest lucru este evidenţiat cu preponderenţă în cazul sortărilor externe care sunt complet diferite ca mod de abordare în raport cu metodele de sortare internă.
    - (2) În general, îmbunătăţirea performanţelor unui algoritm presupune elemente foarte sofisticate, chiar în condiţiile utilizării unor structuri de date dedicate.
        - În mod paradoxal algoritmii cu performanţe superioare sunt mai complicaţi, ocupă mai multă memorie şi utilizează de regulă structuri de date specializate.