9. Structura de date mulţime

9.1. Introducere

- O altă structură de date considerată uneori fundamentală alteori avansată, este **structura mulţime**, care se defineşte generic astfel:

```
------------------------------------------------------------------
TYPE TipMultime = SET OF TipDeBaza;            [9.1.a]
------------------------------------------------------------------
```

- Valorile posibile ale unei variabile `x` a tipului `TipMultime`, sunt mulţimi de elemente ale lui `TipDeBaza`.

- Vom numi **mulţime de bază** mulţimea tuturor elementelor lui `TipDeBaza`.

- În aceste condiţii, **mulţimea tuturor submulţimilor** de elemente ale lui `TipDeBaza` formează **puterea** mulţimii de bază.

- Tipul `TipMultime` are ca domeniu de valori **puterea mulţimii de bază** asociată lui `TipDeBaza`.

- Cu alte cuvinte fiind dată mulţimea de bază, prin **mulţime** vom înţelege **orice submulţime** a acesteia, inclusiv mulţimea vidă.

    - Spre exemplu dacă se alege drept mulţime de bază `{a,b,c}`, atunci se pot utiliza următoarele opt submulţimi drept constante ale tipului mulţime asociat tipului de bază [9.1.b].

```
------------------------------------------------------------------
TYPE TipMultime = SET OF (a,b,c);                     [9.1.b]
  []; [a]; [b]; [c]; [a,b]; [a,c]; [b,c]; [a,b,c];
------------------------------------------------------------------
```

- Cardinalitatea unui tip mulţime este:

  $\mathtt{Card(TipMultime)=2^{Card(TipDeBaza)}}$

    - Această formulă poate fi dedusă simplu din faptul că fiecare dintre elementele lui `TipDeBaza` (al căror număr este egal cu cardinalitatea lui `TipDeBaza`), poate fi reprezentat printr-una din valorile "absent" sau "prezent" şi că toate elementele sunt independente unele faţă de altele.

- În manieră clasică peste un tip mulţime se definesc următoarele legi de compoziţie internă, valabile pentru două mulţimi de acelaşi tip: **atribuire**, **reuniune**, **scădere**, **intersecţie**, **negare**.

- De asemenea se mai definesc şi operatorii relaţionali care conduc la valori booleene: **egalitate**, **inegalitate** , **incluziune**, **incluziune inversă**.

- În sfârşit, dacă `e` este instanţă a lui `TipDeBaza` şi `m` este o variabilă de tip mulţime având asociat acelaşi tip de bază, atunci este definită şi **relaţia de apartenenţă** a lui `e` la `m`.

- Capitolul de faţă îşi propune:
  - (1) Să extindă setul de operatorii clasici şi asupra altor categorii de mulţimi cu caracter mai deosebit.
  - (2) Să studieze şi să prezinte câteva dintre posibilităţile de implementare ale acestui tip de date abstract.

- Deşi în matematică **noţiunea de mulţime** nu se defineşte fiind considerată o **noţiune primară**, în cadrul cursului de faţă vom înţelege prin **mulţime** o colecţie de elemente.
  - Fiecare element al unei mulţimi poate fi la rândul său o mulţime sau un element primitiv numit **atom**.
  - Toate elementele unei mulţimi sunt diferite, adică o mulţime **nu** conţine două copii ale aceluiaşi element.
  - Atunci când sunt utilizaţi în proiectarea algoritmilor şi a structurilor de date, **atomii** sunt de regulă întregi, caractere, sau şiruri de caractere.
  - În orice mulţime toate elementele sunt de acelaşi **tip**.
  - De multe ori, se consideră că atomii sunt **ordonaţi liniar** printr-o relaţie de precedenţă. [Cr87].

9.2. Tipul de date abstract mulţime

- Considerând mulţimea un **tip de date abstract**, asupra ei pot fi imaginate diferite tipuri de operaţii derivate din operaţiile clasice definite asupra mulţimilor.

- În continuare pentru aplicaţiile avute în vedere se consideră următorul **set de operatori** [AH85].

Tipul de date abstract Mulţime

Modelul matematic: Mulţime definită în sens matematic.

Notaţii: [9.2.a]

```
TipMultime   - tipul de date abstract mulţime
TipElement   - tipul de date asociat  elementelor
               multimii (tipul de bază)
A,B,C        - mulţimi încadrate în TipMultime
x            - valoare (obiect) de TipElement
b            - valoare booleană
```

Operatori:

1. ***Reuniune****(TipMultime A, TipMultime B, TipMultime C)* - operator care primeşte ca date de intrare mulţimile *A* şi *B* şi atribuie rezultatul *A* **∪** *B* variabilei mulţime *C*.

2. ***Intersectie****(TipMultime A, TipMultime B, TipMultime C)* - operator care primeşte ca date de intrare mulţimile *A* şi *B* şi atribuie rezultatul *A* ∩ *B* variabilei mulţime *C*.

3. ***Diferenta****(TipMultime A, TipMultime B, TipMultime C)* - operator care primeşte ca date de intrare mulţimile *A* şi *B* şi atribuie rezultatul *A* **-** *B* variabilei mulţime *C*.

4. ***Uniune****(TipMultime A, TipMultime B, TipMultime C)* - defineşte operatorul uniune adică reuniunea mulţimilor disjuncte. Cu alte cuvinte operatorul atribuie variabilei mulţime *C* valoarea *A* ∪ *B*. *C* nu este definită dacă *A* ∩ *B* ≠ Φ, adică dacă mulţimile *A* şi *B* nu sunt disjuncte.

5. *boolean* ***Apartine****(TipElement x, TipMultime A)* - operator care primeşte ca parametri de intrare elementul *x* al cărui tip este tipul de bază al mulţimii *A* şi mulţimea *A*, şi returnează o valoare booleană – adevărat sau fals - după cum *x* aparţine sau nu mulţimii *A*.

6. ***Vid****(TipMultime A)* - operator care atribuie mulţimii *A*, mulţimea vidă.

7. ***Adauga****(TipElement x, TipMultime A)* - unde *A* este o variabilă de tip mulţime iar *x* un element al cărui tip este identic cu tipul elementelor lui *A*. Operatorul face din *x* un element al lui *A* adică noua valoare a lui A este *A* **∪** *{x}*. Dacă *x* este deja membru al mulţimii *A*, operatorul nu-l modifică pe *A*.

8. ***Suprima****(TipElement x, TipMultime A)* - operator care extrage atomul *x* din *A*, adică *A* este înlocuit cu *A* **-** *{x}*. Dacă *x* nu aparţine mulţimii originale *A*, operatorul nu modifică valoarea lui *A*.

9. ***Atribuie****(TipMultime A, TipMultime B)* - operator care face ca valoarea variabilei mulţime *A* să fie egală cu valoarea variabilei mulţimi *B*, adică îl atribuie pe *B* lui *A* *(A=B)*.

10. *TipElement* ***Min****(TipMultime A)* -operator care returnează cel mai mic element al mulţimii *A*. În mod similar ***Max****(A)* returnează cel mai mare element al mulţimii *A*. Aceste operaţii pot fi aplicate numai mulţimilor ale căror elemente pot fi ordonate liniar printr-o relaţie de precedenţă.

11. *boolean* ***Egal****(TipMultime A, TipMultime B)* - operator care returnează valoarea adevărată dacă şi numai dacă mulţimile *A* şi *B* sunt egale.

12. *TipMultime* ***Caută****(TipElement x)* - operator care operează asupra unei colecţii de mulţimi distincte. ***Cauta****(x)* returnează mulţimea (unică) căreia îi aparţine elementul *x*.

9.3. Implementarea structurii mulţime utilizând structuri de date fundamentale

9.3.1. Implementarea structuri mulţime cu ajutorul vectorilor binari

- O implementare performantă a unei structuri de date abstracte **mulţime** depinde:
  - De **operaţiile** care vor fi realizate asupra structurii.
  - De **dimensiunea mulţimii** (cardinalitatea acesteia).
- Când **mulţimile** cu care se lucrează sunt **submulţimi** ale unei "mulţimi de bază" de mici dimensiuni ale cărei elemente sunt întregii `0,1,2,...,n-1`, cu un `n` precizat, atunci în implementare se poate utiliza **vectorul binar** ("**bit-vector**") sau **tabloul boolean**.
- O astfel de mulţime poate fi reprezentată printr-un **vector binar** pe baza **funcţiei sale caracteristice**.
  - **Funcţia caracteristică** definită pe mulţimea indicilor vectorului cu valori în boolean, precizează că cel de-al `i`-lea bit al vectorului este adevărat (are valoarea 1) dacă `i` este un element al mulţimii [Cr87].
- Referitor la această implementare se pot face următoarele precizări:
  - (1) Operaţiile ***Aparţine*, *Adaugă*** şi ***Suprimă*** pot fi realizate într-un interval constant de timp, adresând direct bitul corespunzător.
  - (2) Operaţiile ***Reuniune*, *Intersecţie***, şi ***Diferenţa*** pot fi realizate într-un timp proporţional cu dimensiunea mulţimii de bază (cardinalitatea acesteia).
  - (3) Dacă această cardinalitate este suficient de redusă, astfel încât vectorul binar se suprapune ca dimensiune peste un cuvânt de calculator, atunci operaţiile ***Reuniune*, *Intersecţie***, şi ***Diferenţa*** pot fi realizate printr-o singură operaţie logică cablată.
- În limbajul PASCAL, anumite mulţimi de dimensiuni reduse pot fi generate cu ajutorul constructorului **SET**. Dimensiunea maximă a unei astfel de mulţimi depinde de compilatorul utilizat.
- În general, pentru aplicaţii care utilizează mulţimi care sunt submulţimi ale unei mulţimi de bază universale `0,1,2,...,n-1`, cu `n` depinzând de aplicaţie, utilizând **vectorul binar** drept suport se poate defini următorul **tip mulţime** [9.3.1.a].

```
/*Definirea unui TipMulţime utilizând vectorul binar - varianta C*/

int NumarElemente=n;

typedef boolean TipMultime[NumarElemente];  /*[9.3.1.a]*/

TipMultime A;
```

```
{Definirea unui TipMulţime utilizând vectorul binar - varianta PASCAL}

type TipMultime=array[0..n-1] of boolean;        [9.3.1.a]

var A: TipMultime;
```

- Dacă A este o variabilă a tipului `TipMultime`, atunci `A[i]` este adevărat dacă şi numai dacă `i` aparţine mulţimii A.

- Operaţia ***Reuniune*** poate fi implementată ca şi în secvenţa [9.3.1.b].

```
/*Implementarea operatorului Reuniune   (Performanţa O(n)) - Varianta pseudocod*/

subprogram Reuniune(TipMultime A, TipMultime B, TipMultime C);

  int i;
  pentru (i=0 la n-1)                              [9.3.1.b]
      C[i]= A[i] || B[i];
```

```
{Implementarea operatorului Reuniune   (Performanţa O(n)) - Varianta PASCAL}

procedure Reuniune(A,B: TipMultime; var C: TipMultime);
  var i: integer;
  begin
    for i:= 0 to n-1 do                             [9.3.1.b]
      C[i]:= A[i] or B[i]
  end;
```

- Operatorii ***Intersectie*** şi ***Diferenta*** rezultă imediat înlocuind operaţia "**or**" ("**||**") cu **"and"** ("&&") respectiv "**and not**" **("&& !").**

- Într-o manieră similară, se pot implementa şi ceilalţi operatori precizaţi în &9.2

  - Operatorii ***Uniune*** şi ***Caută*** nu au sens în acest context.

- **Vectorii binari** pot fi utilizaţi în implementarea mulţimilor şi în situaţia în care elementele acestora **nu** sunt întregi consecutivi.

* În aceast caz trebuie stabilită o **corespondenţă** între **elementele mulţimii** şi o **submulţime convenabilă** a **numerelor întregi**.

    * Pentru acest scop poate fi utilizată o structură de date de tip **asociere** care permite stabilirea corespondenţei în ambele sensuri (Vol.1 &6.8).

    * **Recomandare**: de regulă corespondenţa “`întregi - elemente ale mulţimii`” poate fi cel mai eficient implementată cu ajutorul unui **tablou** `A`, unde `A[i]` este elementul corespunzător întregului `i`.

9.3.2. Implementarea structurii mulţime cu ajutorul listelor înlănţuite

* Structura de date mulţime poate fi implementată şi cu ajutorul unei **liste înlănţuite**, unde nodurile listei sunt elemente ale mulţimii.

* Reprezentarea bazată pe liste are unele **avantaje** faţă de reprezentarea bazată pe vectori binari:

    * (1) Utilizează **spaţiul de memorie** strict necesar mulţimii şi nu spaţiul corespunzător “mulţimii de bază”.

    * (2) Este mai **generală** întrucât poate manipula mulţimi care nu sunt supuse nici unei constrângeri.

* Pentru exemplificare se prezintă implementarea operatorului ***`Intersectie`*** în cazul mulţimilor reprezentate prin **liste înlănţuite simple**.

    * Dacă listele **nu** sunt ordonate, trebuie verificată pentru fiecare element al lui L_1 concordanţa cu fiecare element al lui L_2 parcurgând de fiecare dată integral pe L_2, proces a cărui regie este O *(n^2)*.

    * Dacă mulţimea de bază este **ordonată liniar**, atunci ea poate fi reprezentată printr-o listă ordonată.

        * Avantajul unei astfel de implementări este acela că **nu** este necesară parcurgerea întregii liste pentru a determina dacă un element aparţine sau nu acesteia.

* Un element aparţine **intersecţiei** a două liste L_1 şi L_2 dacă şi numai dacă se găseşte în ambele liste.

* Întrucât cele două liste sunt **ordonate**, pentru fiecare element `e` al lui L_1 se parcurge L_2 până la găsirea:

    * (1) Unui element identic, caz în care elementul se adaugă intersecţiei.

    * (2) Unui element mai mare, caz în care se trece la elementul următor al lui L_1.

* Fiecare dintre cele două liste este parcursă cu ajutorul unui **indicator de poziţie** specific.

* Cu alte cuvinte, **intersecţia a două mulţimi** reprezentate prin listele ordonate L_1 şi L_2 poate fi determinată într-o **singură parcurgere secvenţială** a celor două liste, avansând întotdeauna indicatorul de poziţie corespunzător listei care conţine cel mai mic element curent.

* Procedura care implementează această tehnică apare în secvenţa [9.3.2.b].

- Mulţimile avute în vedere sunt implementate ca şi liste înlănţuite ale căror noduri aparţin tipului nod definit ca şi în secvenţa [9.3.2.a]:

```
------------------------------------------------------------
/*Definirea elementelor mulţimii reperezentate prin liste
înlănţuite ordonate –Structuri de date - varianta C*/

typedef  TipNod * ref_nod;

typedef struct TipNod
             {
               TipElement element;            /*[9.3.2.a]*/
               ref_nod urm;
             } TipNod;

ref_nod incepA, incepB, incepC;
-----------------------------------------------------------
/* Determinarea intersecţiei a două mulţimi - varianta
pseudocod*/

procedure Intersecţie(ref_nod incepA, ref_nod incepB,
ref_nod incepC)

/*Determină intersecţia mulţimilor A şi B reprezentate ca şi
liste înlănţuite ordonate indicate de pointerii incepA şi
incepB. Intersecţia se memorează în lista C indicată de
incepC*/

  ref_nod indicA,indicB,indicC;  /*pointeri la nodurile
  curente în A şi B, respectiv la ultimul nod adăugat în C*/
  incepC=aloca_memorie(TipNod); /*creează lista C*/
  indicA=incepA;
  indicB=incepB;
  indicC=incepC;
  cat timp ((indicA != null) && (indicB != null))
   |/*compară elementele curente ale listelor A şi B*/
   |daca(indicA^.element==indicB^.element)
   |    |{se adaugă elementul intersecţiei}
   |    |indicC^.urm=aloca_memorie(TipNod);
   |    |indicC=indicC^.urm;               /*[9.3.2.b]*/
   |    |indicC^.element=indicA^.element;
   |    |indicA=indicA^.urm;
   |    |indicB=indicB^.urm;
   |    |□
   |  altfel /*elemente diferite*/
   |    daca(indicA^.element<indicB^.element)
   |        indicA=indicA^.urm;
   |      altfel
   |        indicB=indicB^.urm;
   |□ /*cat timp*/
  indicC^.urm=null;
  /*Intersectie*/

------------------------------------------------------------
```

```
{Definirea elementelor mulţimii reperezentate prin liste
înlănţuite ordonate – structuri de date - varianta PASCAL}

type  RefNod=^TipNod;
      TipNod=record
               element: TipElement;              [9.3.2.a]
               urm: RefNod
             end;
-----------------------------------------------------------
/* Determinarea intersecţiei a două mulţimi - varianta
PASCAL*/

procedure Intersecţie(incepA,incepB: RefNod; var incepC:
            RefNod);
{Determină intersecţia mulţimilor A şi B reprezentate ca şi
liste înlănţuite ordonate indicate de pointerii incepA şi
incepB. Intersecţia se memorează în lista C indicată de
incepC}

  var indicA,indicB,indicC: RefNod;
    {pointeri la nodurile curente în A şi B, respectiv la
     ultimul nod adaugat în C}

  begin
    new(incepC); {creează lista C}
    indicA := incepA;
    indicB := incepB;
    indicC := incepC;
    while (indicA <> nil) and (indicB <> nil) do
      begin{compară elementele curente
            ale listelor A şi B}
        if indicA^.element = indicB^.element then
          begin {se adaugă elementul intersecţiei}
            new(indicC^.urm);
            indicC := indicC^.urm;                [9.3.2.b]
            indicC^.element := indicA^.element;
            indicA := indicA^.urm;
            indicB := indicB^.urm
          end
        else {elemente diferite}
          if indicA^.element < indicB^.element then
            indicA := indicA^.urm
          else
            indicB := indicB^.urm
      end;
    indicC^.urm := nil
  end; {Intersectie}
-----------------------------------------------------------
```

- În cadrul procedurii ***Intersecţie***, `A` şi `B` sunt listele corespunzătoare submulţimilor operanzi, iar `C` lista corespunzătoare mulţimii intersecţie care se crează.

- Fiecare listă are un pointer specific: `indicA`, `indicB` respectiv `indicC`.

- Se poziţionează pointerii pe începutul listelor specifice.

- În continuare într-o buclă (instrucţiunea **`while`**), se compară elementele curente ale listelor A şi B.
  - În caz de egalitate, se trece elementul în C şi se avansează ambii pointeri în A şi B.
  - În caz de inegalitate se avansează pointerul din lista care conţine cel mai mic element curent.
- În rutina din secvenţa [9.3.2.b], se presupune că `TipElement` este un tip ordonat, ale cărui constante se pot compara cu operatorul ">".
  - Dacă acest lucru nu este valabil, trebuie implementată o funcţie care stabileşte relaţia de precedenţă dintre două elemente.
- Ca şi exerciţiu se poate realiza implementarea procedurii ***Intersecţie*** utilizând operaţiile primitive definite asupra listelor.
- Operaţiile ***Reuniune*** şi ***Diferenţă*** pot fi implementate prin proceduri foarte apropiate ca formă de procedura ***Intersecţie***.
  - Pentru ***Reuniune*** este necesar ca toate elementele din A şi B să fie trecute în ordine în C.
    - (1) Procedeul este identic cu cel aplicat la ***Intersectie*** pentru elementele egale.
    - (2) Pentru elementele diferite, se trece în lista C cel mai mic element.
    - (3) Când se ajunge la sfârşitul uneia din liste (A sau B) trebuiesc trecute în C restul elementelor aparţinând listei neterminate.
  - Pentru operatorul ***Diferenţa***, **nu** se adaugă la C elemente găsite egale.
    - (1) Se adaugă la C elementul curent al listei A când este găsit mai mic decât elementul curent al listei B.
    - (2) Se adaugă la C elementele care au mai rămas în lista A, când B a fost parcurs integral.
- În secvenţa [9.3.2.c] apare un exemplu de implementare a structurii mulţime cu ajutorul listelor înlănţuite respectiv operatorii ***Adaugare***, ***Afisare***, ***Intersectie, Reuniune*** şi ***Diferenţă***.

```
//Exemplu de implementare a structurii mulţime utilizând
structura listă înlănţuită ordonată

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
//structuri de date
typedef int TipElement;

typedef struct TipNod{

     TipElement element;
     struct TipNod *urm;

} TipNod;

typedef  TipNod *ref_nod;

ref_nod incepA, incepB, incepC, incepD, incepE, incepF;
//pot fi si locale

ref_nod Adauga(ref_nod incep, TipElement elem)
//functie utilizata pentru a adauga cuvinte elemente în
mulţimea încep. Creaza o lista ordonata

{
   ref_nod aux, p;

   p = (TipNod*)malloc(sizeof(TipNod));
   p->element = elem;
   p->urm = NULL;

   if (incep == NULL)  //lista vida
       incep = p;
     else
     {
       aux = incep;
       if (aux->urm == NULL) //lista cu un element
             {
           if (aux->element < p->element) aux->urm = p;
            else
             if (p->element < aux->element)
              {
                p->urm = aux;
                aux->urm = NULL;
                incep = p;
              }
           }
          else //lista cu mai multe elemente
           {
           while ((aux->urm != NULL) && (aux->element <=
                   p->element))
                 aux = aux->urm;
           if (aux->element > p->element) //insertie in fata
                 {
               *p = *aux;
               aux->urm = p;
               aux->element = elem;
                }
               else
                aux->urm = p; //insertie in spate
          }
```

```
    }
    return incep;
}
//Adaugare

void Afisare(ref_nod incep)
{
    ref_nod aux;
    aux = incep;

    while (aux != NULL)
    {
        printf("%d ", aux->element);
        aux = aux->urm;
    }

    printf("\n");
}
//Afisare

ref_nod Intersectie(ref_nod incepA, ref_nod incepB, ref_nod
incepC)
//intersectia multimilor incepA si incepB. Rezultatul in
incepC care este returnat
{
    ref_nod indicA, indicB, indicC;
    incepC = (TipNod*)malloc(sizeof(TipNod));

    indicA = incepA;
    indicB = incepB;
    indicC = incepC;

    while ((indicA != NULL) && (indicB != NULL))
    {
        if (indicA->element == indicB->element)
        {
            indicC->urm =
(TipNod*)malloc(sizeof(TipNod));
            indicC = indicC->urm;
            indicC->element = indicA->element;
            indicA = indicA->urm;
            indicB = indicB->urm;
        }
        else
        if (indicA->element<indicB->element)
            indicA = indicA->urm;
        else
            indicB = indicB->urm;
    }

    indicC->urm = NULL;
    return incepC->urm;
}
//Intersectie
```

```c
ref_nod Reuniune(ref_nod incepA, ref_nod incepB, ref_nod
incepC)
//reuniunea multimilor incepA si incepB. Rezultatul in
incepC care este returnat

{
	ref_nod indicA, indicB, indicC;
	incepC = (TipNod*)malloc(sizeof(TipNod));

	indicA = incepA;
	indicB = incepB;
	indicC = incepC;

	while ((indicA != NULL) && (indicB != NULL))
	{
		indicC->urm = (TipNod*)malloc(sizeof(TipNod));
		indicC = indicC->urm;

		if (indicA->element == indicB->element)
		{
			indicC->element = indicA->element;
			indicA = indicA->urm;
			indicB = indicB->urm;
		}
		else
		if (indicA->element<indicB->element)
		{
			indicC->element = indicA->element;
			indicA = indicA->urm;
		}

		else
		{
			indicC->element = indicB->element;
			indicB = indicB->urm;
		}

	}

	if ((indicA != NULL) || (indicB != NULL))
	{
		indicC->urm = (TipNod*)malloc(sizeof(TipNod));
		indicC = indicC->urm;

		while (indicA != NULL)
		{
			indicC->element = indicA->element;
			indicA = indicA->urm;
		}

		while (indicB != NULL)
		{
			indicC->element = indicB->element;
			indicB = indicB->urm;
		}
	}
```

```
    indicC->urm = NULL;
    return incepC->urm;
}
//Reuniune

ref_nod Diferenta(ref_nod incepA, ref_nod incepB, ref_nod
incepC)
//diferenta multimilor incepA si incepB. Rezultatul in
incepC care este returnat

{
    ref_nod indicA, indicB, indicC;
    incepC = (TipNod*)malloc(sizeof(TipNod));

    indicA = incepA;
    indicB = incepB;
    indicC = incepC;

    while ((indicA != NULL) && (indicB != NULL))
    {
        if (indicA->element == indicB->element)
        {
            indicA = indicA->urm;
            indicB = indicB->urm;
        }
        else
        if (indicA->element<indicB->element)
        {
            indicC->urm =
(TipNod*)malloc(sizeof(TipNod));
            indicC = indicC->urm;
            indicC->element = indicA->element;
            indicA = indicA->urm;
        }
        else
            indicB = indicB->urm;
    }

    while (indicA != NULL)
    {
        indicC->urm = (TipNod*)malloc(sizeof(TipNod));
        indicC = indicC->urm;
        indicC->element = indicA->element;
        indicA = indicA->urm;
    }

    indicC->urm = NULL;
    return incepC->urm;
}
//Diferenta
------------------------------------------------------------
```

- Operaţia ***Atribuie****(A,B)* presupune copierea listei A în lista B.

- Trebuie subliniat faptul că **nu** este suficient ca acest lucru să fie implementat prin simpla modificare a pointerului care-l indică pe `B` dându-i pur şi simplu valoarea pointerului care-l indică pe `A`, ci **trebuie realizată efectiv copierea**.

- Dacă nu se procedează în această manieră, modificări ulterioare ale unei mulţimi se resfrâng în mod nedorit şi asupra celeilalte (conceptul "**aliasing**").

- Operatorul ***Min*** returnează primul element al listei.

- Operatorii ***Suprimă*** şi ***Caută*** pot fi implementaţi prin căutarea nodului implicat în operaţie, după una din tehnicile precizate la studiul listelor, iar în cazul lui ***Suprimă***, ştergând nodul găsit.

- În ceea ce priveşte operaţia ***Adaugă***, aceasta este o aplicaţie derivată din listele ordonate, care a fost precizată în primul volum al lucrării, cap.6 [Cr00].

9.4. Structuri de date derivate din structura mulţime

9.4.1. Structura dicţionar

- În unele aplicaţii care utilizează structuri de date mulţime nu sunt necesare operaţii complexe cum ar fi reuniunea sau intersecţia.

  - De regulă, este necesară păstrarea evidenţei unei mulţimi "curente" de obiecte în care periodic se realizează **adăugiri**, **extrageri** sau teste de **apartenenţă**.

- Se numeşte **dicţionar** o structură de date abstractă derivată din structura mulţime, pe care sunt definiţi operatorii:

  - (1)***Adaugă***.
  - (2)***Suprimă***.
  - (3)***Aparţine***.
  - (4)***Vid*** - care permite iniţializarea unei astfel de structuri de date.

- În continuare vor fi discutate câteva posibilităţi de implementare a **structurii de date dicţionar**.

- Un dicţionar poate fi implementat cu ajutorul unei structuri **listă** atât în varianta **ordonată** cât şi în varianta **neordonată**.

- O altă implementare posibilă a dicţionarului o reprezintă **vectorul binar**.

  - Aceast lucru este realizabil dacă elementele mulţimii sunt numere întregi în domeniul `0,1,2,...,N-1,` cu un `N` precizat, sau pot fi puse în corespondenţă cu un astfel de domeniu de numere întregi.

- În continuare se prezintă mai în detaliu **alte două modalităţi** de implementare a **structurii dicţionar**.

9.4.1.1. Implementarea structurii dicţionar cu ajutorul tablourilor

liniare

- Un dicţionar poate fi implementat cu ajutorul unui **tablou** prevăzut cu un indicator la ultima intrare utilizată.

    - Această implementare este viabilă dacă se presupune ca dicţionarul **nu** va conţine mai multe elemente decât dimensiunea maximă a tabloului.

    - Faţă de listele înlănţuite, această implementare care are **avantajul** simplităţii, are două **dezavantaje**:

        - (1) Dicţionarul nu poate creşte în mod arbitrar.

        - (2) Spaţiul de memorie este utilizat ineficient.

- În secvenţa [9.4.1.1.a] apare un model al acestei reprezentări.

```
/*Implementarea structurii dicţionar cu ajutorul structurii
tablou -varianta C*/

int DimMax = {o valoare potrivita};

typedef TipDictionar
      {
       int ultim;
       TipElement continut[DimMax];
      } TipDictionar;

void Vid(TipDictionar * A) /*Performanţa O(1)*/
{
  (*A).ultim= -1;
} /*Vid*/

boolean Apartine(TipElement x, TipDictionar * A)
{
  int i;                    /*Performanţa O(n)*/
  for (i=0;i<(*A).ultim;i++)
    if ((*A).continut[i]==x) return true;
  return false;
 } /*Apartine*/                          /*[9.4.1.1.a]*/

void Adauga(TipElement x, TipDictionar * A)
{
  if (!Apartine(x,&A))      /*Performanţa O(n)*/
    if ((*A).ultim<(DimMax-1))
        {
         (*A).ultim=(*A).ultim+1;
         (*A).continut[(*A).ultim]=x;
        }
      else
        *eroare('structura este plina');
 } /*Adauga*/

void Suprima(TipElement x, TipDictionar * A)
{
```

```
  int i;                  /*Performanţa O(n)*/
  if ((*A).ultim>=0)
    {
      i= 0;
      while (((*A).continut[i]<>x)&&(i<(*A).ultim))
          i++;
      if ((*A).continut[i]==x)
        {
          (*A).continut[i]=(*A).continut[(*A).ultim];
          (*A).ultim=A.ultim-1;
        } /*if*/
    } /*if*/
 } /*Suprima*/
```

{Implementarea structurii dicţionar cu ajutorul structurii tablou – varianta PASCAL}

```
const DimMax = {o valoare potrivita};

type TipDictionar = record
       ultim: integer;
       continut: array[1..DimMax] of TipElement
     end;

procedure Vid(var A: TipDictionar); {Performanţa O(1)}
  begin
    A.ultim := 0;
  end;{Vid}

function Apartine(x: TipElement; var A: TipDictionar):
boolean;
  var i: integer;               {Performanţa O(n)}
  begin
    for i := 1 to A.ultim do
      if A.continut[i] = x then return(true);
    return(false)
  end;{Apartine}                        [9.4.1.1.a]

procedure Adauga(x: TipElement; var A: TipDictionar);
  begin
    if not Apartine(x,A) then     {Performanţa O(n)}
      if A.ultim < DimMax then
         begin
           A.ultim := A.ultim + 1;
           A.continut[A.ultim] := x
         end
      else
        eroare('structura este plina')
   end;{Adauga}

procedure Suprima(x: TipElement; var A: TipDictionar);
  var i: integer;
  begin                        {Performanţa O(n)}
    if A.ultim > 0 then
      begin
```

```
      i := 1;
      while (A.continut[i] <> x) and (i < A.ultim) do
        i := i + 1;
      if A.continut[i] = x then
        begin
          A.continut[i] = A.continut[A.ultim];
          A.ultim := A.ultim - 1
        end
    end
  end; {Suprima}
```

- Implementarea intersecţiei şi reuniunii cu ajutorul tablourilor liniare este relativ **dificilă** motiv pentru care **nu** a fost abordată problema reprezentării mulţimilor în general cu ajutorul tablourilor.

- Cu toate acestea, întrucât există metode eficiente de sortare a tablourilor, procedurile descrise în secvenţa care se referă la structura dicţionar pot fi considerate drept punct de plecare într-o posibilă implementare a structurilor mulţime cu ajutorul **tablourilor liniare**.

9.4.1.2. Implementarea structurii dicţionar prin tehnica dispersiei

- După cum s-a precizat în paragraful anterior, implementarea **dicţionarului** bazată pe **tablouri liniare** necesită în medie O*(n)* paşi pentru execuţia operaţiilor ***Adaugă***, ***Suprimă*** sau ***Aparţine*** într-un dicţionar cu `n` elemente.

- O regie similară se obţine şi pentru implementarea bazată pe **liste înlănţuite**.

- Implementarea bazată pe **vectori binari**, consumă un interval constant de timp pentru a executa oricare din operaţiile anterior precizate, cu restricţia însă că dimensiunea mulţimii de bază este limitată la o dimensiune impusă de arhitectura hardware.

- O altă tehnică larg utilizată în implementarea structurilor dicţionar este **tehnica dispersiei** (Vol.1 &7.3).

  - **Tehnica dispersiei** necesită în medie un timp constant pe operaţie.

  - Nu impune nici o restricţie referitoare la cardinalitatea sau tipul elementelor mulţimii de bază.

  - În cel mai rău caz, această tehnică necesită O*(n)* paşi adică identic cu implementările bazate pe tablouri sau liste. În practică însă, se ajunge foarte rar în această situaţie.

- După cum s-a precizat în Vol.1, se pot lua în considerare două variante ale tehnicii dispersiei:

  - (1) **Dispersia deschisă** în care situaţiile de coliziune se tratează prin înlănţuire directă şi care permite ca mulţimea să fie memorată într-un spaţiu practic nelimitat, (deci cardinalitatea mulţimii de bază **nu** este limitată).

- (2) **Dispersia închisă** bazată pe metoda adresării deschise liniare sau adresării deschise patratice în care se utilizează un spaţiu fix de memorie şi în consecinţă mulţimea de bază va fi limitată ca dimensiune.

- Întrucât principiile tehnicii dispersiei au fost pe larg discutate în referinţa mai sus precizată, în paragraful de faţă se vor prezenta două **exemple** care materializează cele două variante.

  - În ambele exemple se va utiliza o **funcţie de dispersie** clasică notată cu `h(x)` unde `x` este de tip şir de caractere (string).

- În secvenţa [9.4.1.2.a] se observă structurile de date utilizate în implementarea **dicţionarului** prin tehnica **dispersiei deschise**.

  - **Dicţionarul** este de fapt un **tablou de pointeri**, fiecare indicând o lista înlănţuită cu noduri de tip cuvânt.

  - Fiecare listă cuprinde acele elemente `x` ale dicţionarului pentru care funcţia `h(x)` furnizează aceeaşi valoare (clasă de elemente), situaţia de coliziune fiind rezolvată în acest caz prin **metoda înlănţuirii directe**.

  - Tehnica utilizată în implementarea listelor face necesară tratarea separată a primului cuvânt din listă, element care se observă în cadrul procedurii ***Suprimă***.

```
/*Implementarea structurii dicţionar prin tehnica dispersiei
deschise – varianta C*/

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

#define P=131

typedef char* TipElement; //pointer la string

typedef int TipIndice;

typedef struct TipCuvant
{
    TipElement element;
    struct TipCuvant *urm;
} TipCuvant;

typedef TipCuvant *RefCuvant;

typedef struct TipDictionar
{
    RefCuvant dictionar[P];
} TipDictionar;

TipIndice h(TipElement x)
{
    int i,suma;
```

```
    suma=0;
    int nr=strlen(x);
      for(i=0;i<nr;i++)
        suma=suma+(int)x[i];
    return suma%P;
}

void Vid(TipDictionar *A)
{
    int i;

    for(i=0;i<=P-1;i++)
         A->dictionar[i]=NULL;
}

bool Apartine(TipElement x, TipDictionar *A)
{
    RefCuvant curent;

    curent=A->dictionar[h(x)];
    while(curent!=NULL)
    {
        if(strcmp(curent->element,x)==0) return true;
        else (curent=curent->urm);
    }
    return false;
}

void Adauga(TipElement x, TipDictionar *A)              //[9.4.1.2.a]
{
    int l;
    RefCuvant vechi;

    if((Apartine(x,A))==false)
    {
        l=h(x);
        vechi=A->dictionar[l];
        A->dictionar[l]=(RefCuvant)malloc(sizeof(TipCuvant));
        //alocare memorie pentru campul element (cuvantul propriu-zis)
        A->dictionar[l]->element=
          (char*)malloc((strlen(x)+1)*sizeof(char));
        strcpy(A->dictionar[l]->element,x);
        A->dictionar[l]->urm=vechi;
    }
}

void Suprima(TipElement x, TipDictionar *A){

    RefCuvant curent,aux;
    int l;

    l=h(x);
    if(A->dictionar[l]!=NULL)
    {
        if(strcmp(A->dictionar[l]->element,x)==0)
        {
            aux=A->dictionar[l];
            A->dictionar[l]=A->dictionar[l]->urm;
            free(aux); //eliberare memorie
```

```
        }

        else
        {
            curent=A->dictionar[l];

            while(curent->urm!=NULL)
            {
                if(strcmp(curent->urm->element,x)==0)
                {
                    aux=curent->urm;
                    curent->urm=curent->urm->urm;
                    free(aux); // eliberare memorie
                }
                else curent=curent->urm;
            }

        }
    }
}

void Afisare(TipDictionar *A)
{
  for(i=0; i<P; i++)
    {
      printf("Hash %d: ", i);
      while(A->dictionar[i]!=NULL)
      {
          printf("%s ",A->dictionar[i]->element);
          A->dictionar[i]=A->dictionar[i]->urm;
      }
      printf("\n");
    }
```

{Implementarea structurii dicţionar prin tehnica dispersiei deschise – varianta PASCAL}

```
const P={o valoare convenabilă,de regulă numar prim}

type TipElement = array[1..12] of char;
     TipIndice = 0..(P-1);
     RefCuvant: ^TipCuvant;
     TipCuvant = record
                  element: TipElement;
                  urm: RefCuvant
                 end;
     TipDictionar = array[TipIndice] of RefCuvant;

function h(x: TipElement): TipIndice;
  var i,suma: integer;
  begin                    {funcţia de dispersie}
    suma := 0;
    for i := 1 to 12 do
      suma := suma + ord(x[i]);
    h := suma mod P
  end; {h}
```

```
procedure Vid(var A: TipDictionar);                    [9.4.1.2.a]
  var i: integer;
  begin
    for i := 1 to P-1 do
      A[i] := nil
  end; {Vid}

function Apartine(x: TipElement; var A: TipDictionar):
boolean;
  var curent: RefCuvant;
  begin
    curent :=A[h(x)];
    while curent <> nil do
      if curent^.element = x then
        return(true)
      else
        curent := curent^.urm;
    return(false)
  end; {Apartine}

procedure Adauga(x: TipElement; var A: TipDictionar);
  var l: integer;
      vechi: RefCuvant;
  begin
    if not Apartine(x,A) then
      begin
        l := h(x);
        vechi := A[l];      {inserţie în faţă}
        new(A[l]);
        A[l]^.element := x;
        A[l]^.urm := vechi
      end
  end; {Adauga}

procedure Suprima(x: TipElement; var A: TipDictionar);
  var curent: RefCuvant;
      l: integer;
  begin                                              [9.4.1.2.a]
    l := h(x);
    if A[l] <> nil then
      begin
        if A[l]^.element = x then {x este pe prima
                                   poziţie}
          A[l] := A[l]^.urm {se scoate x din listă}
        else
          begin
            curent := A[l];
            {se aplica tehnica lookahead}
            while curent^.urm <> nil do
              if curent^.urm^.element = x then
                begin {scoate pe x din listă}
                  curent^.urm := curent^.urm^.urm;
                  return
                end
              else {x nu a fost încă găsit}
                curent := curent^.urm
```

```
        end
      end
  end; {Suprima}
```

- În secvenţa [9.4.1.2.b] apare implementarea **dicţionarului** prin tehnica **dispersiei închise**.
    - Situaţiile de coliziune se tratează prin **metoda adresării deschise liniare**.
    - Structurile de date utilizate sunt cele precizate în secvenţa [9.4.1.2.b], iar funcţia `h(x)` este cea utilizată în exemplul anterior.
    - Prin **convenţie**, s-a utilizat un şir de 12 caractere underscore pentru valoarea `liber` şi un şir de 12 asteriscuri pentru valoarea `şters`, presupunându-se că nici un element nu poate lua aceste valori.
- Au fost definite două funcţii de căutare.
    - (1) Funcţia **`Caută`** parcurge tabelul `A` conform **metodei de adresare deschisă liniară** până:
        - (a) Îl găseşte pe `x`.
        - (b) Găseşte o locaţie neocupată (`liber`).
        - (c) A parcurs circular tabloul şi nu l-a găsit pe `x`.
        - (d) În toate cazurile **`Cauta`** returnează **indicele** din tablou la care s-a oprit căutarea indiferent de motiv.
    - (2) Funcţia **`Cauta1`** este asemănătoare lui **`Cauta`** cu singura deosebire că ea extinde procesul de căutare şi la locaţii marcate cu `sters`.
        - **`Cauta1`** se foloseşte numai în operatorul ***Adauga*** pentru a găsi prima locaţie disponibilă.

```
/*Implementarea structurii dicţionar prin tehnica dispersiei
închise – Varianta C*/

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

#define P 131

const char *liber = "____________";
const char *sters=  "************";

typedef char *TipElement;

typedef TipElement TipDictionar[P];

int h(TipElement x)
```

```c
{
    int i,suma;

    suma=0;
    int nr=strlen(x);
    for(i=0;i<nr;i++)
        suma=suma+(int)x[i];
    return suma%P;
}

void Vid(TipDictionar A)
{
    int i;

    for(i=0;i<P;i++)
        A[i]=(char*)malloc(sizeof(char));
    for(i=0;i<P;i++)
        strcpy(A[i],liber);
}

void Adauga(TipElement x, TipDictionar A)
{
    int l;

    if(strcmp(A[Cauta(x,A)],x)!=0) {
        l=Cauta1(x,A);
        if(( strcmp(A[l],liber)==0) || (strcmp(A[l],sters)==0 ))
            strcpy(A[l],x);
        else
            printf("\n Tabela e plina\n");
        }
}

bool Apartine(TipElement x, TipDictionar A)
{

    if(A[Cauta(x,A)==x])
        return true;
    else
        return false;
}

int Cauta( TipElement x, TipDictionar A)
{
    int initial,i;

    initial= h(x);
    i= 0;
    while (((i<P) && strcmp(A[(initial+i) % P],x)!=0)
            &&(strcmp(A[(initial+i) % P],liber)!=0))
        i= i+1;
    return ((initial+i) % P);
}

int Cauta1( TipElement x, TipDictionar A)
{
    //verifica si pozitiile sterse
    int initial,i;
```

```c
    initial= h(x);
    i= 0;
    while (((i<P) && strcmp(A[(initial+i)% P],x)!=0)
           && strcmp(A[(initial+i)% P],liber)!=0 &&
           strcmp(A[(initial+i)% P],sters)!=0)
        i= i+1;

    return ((initial+i) % P);
}

void Suprima(TipElement x, TipDictionar A)
{
    int i;

    i=Cauta(x,A);
    if(strcmp(A[i],x)==0)
        strcpy(A[i],sters);
}

void Afisare(TipDictionar A)
{

    int i;

    for(i=0;i<P;i++)
        printf("Codul Hash :%d -> %s\n",h(A[i]),A[i]);
}
```

{Implementarea structurii dicţionar prin tehnica dispersiei închise – Varianta PASCAL}

```pascal
const P = {o valoare potrivită, de regulă număr prim};
      liber = '            ';  {12 blancuri}
      sters = '************';  {12 asteriscuri}

type TipElement = array[1..12] of char;
     TipDictionar = array[0..P-1] of TipElement;

procedure Vid(var A: TipDictionar);
  var i: integer;
  begin
    for i := 0 to P-1 do
      A[i] := liber
  end; {Vid}

function Cauta(x: TipElement; A: TipDictionar): integer;
  var initial,i: integer;
  begin                                            [9.4.1.2.b]
    initial := h(x);
    i := 0;
    while (i<P) and (A[(initial+i) mod P]<>x) and
          (A[(initial+i) mod P]<>liber) do
      i := i+1;
    Cauta := (initial+i) mod P
  end; {Cauta}
```

```
function Cauta1(x: TipElement; A: TipDictionar): integer;
{similară funcţiei Cauta dar în plus returnează şi o intrare
ştearsă}

procedure Adauga(x: TipElement; var A: TipDictionar);
  var l: integer;
  begin
    if A[Cauta(x,A)] <> x then
      begin
        l := Cauta1(x,A);
        if (A[l] = liber) or (A[l] = sters) then
          A[l] := x
        else
          eroare('tabela este plină')
      end
  end; {Adauga}

function Apartine(x: TipElement; A: TipDictionar): boolean;
  begin
    if A[Cauta(x,A)] = x then
      Apartine := true
    Else                                           [9.4.1.2.b]
      Apartine := false
  end; {Apartine}

procedure Suprima(x: TipElement; var A: TipDictionar);
  var i: integer;
  begin
    i := Cauta(x,A);
    if A[i] = x then A[i]:= sters
  end; {Suprima}
------------------------------------------------------------
```

9.4.2. Structuri de date complexe bazate pe structura mulţime

9.4.2.1. Relaţia bazată pe corespondenţe multiple

- Se consideră o mulţime de studenţi şi o mulţime de concursuri profesionale.

- Un exemplu de relaţie bazată pe **corespondenţe multiple** (“**many–many relationship**” [AH85]) îl reprezintă relaţia dintre studenţi şi concursurile profesionale la care aceştia participă.

    - Aceasta este o relaţie bazată pe corespondenţe multiple deoarece:

        - (1) La un concurs pot participa mai mulţi studenţi.

        - (2) Un student poate participa la mai multe concursuri profesionale.

        - (3) În timp, listele de participanţi se pot modifica, întrucât se pot înscrie noi studenţi, unii se pot retrage sau pot trece la alte discipline.

- Problemele care se pun în legătură cu această relaţie bazată pe corespondenţe multiple se referă la a şti la un moment dat:

- (1) **Care sunt studenţii** care participă la o anumită disciplină de concurs.
- (2) **La ce concurs** participă un anumit student.

9.4.2.2. Implementare bazată pe structura tablou a relaţiei bazate pe corespondenţe multiple

- Structura de date cea mai simplă care acoperă aceste cerinţe este un **tablou** cu două dimensiuni `INSCRIERE`, sugerat de figura 9.4.2.2.a, unde valoarea `adevarat` reprezintă **înscris** (marcat cu `x`) iar 0 valoarea `fals` reprezintă **neînscris**.

	Matematica	Programare	Mecanica
Agache			x
Anghel	x	x	x
Arianu			
Ban		x	
Banu	x		
Berinde	x	x	
Cartu		x	
Cerbu	x		x

INSCRIERE

Fig. 9.4.2.2.a. Exemplu de relaţie bazată pe corespondenţe multiple implementată cu ajutorul unei structuri tablou

- Pentru a **înscrie** un student la un concurs profesional este necesară crearea în prealabil a două **asocieri**.
  - (1) O **asociere** AS, eventual implementată prin tehnica dispersiei, care translatează **numele studentului** într-un **indice** aparţinând unei dimensiuni a tabloului.
  - (2) O a doua **asociere** AC care translatează **numele concursului** într-un **indice** al celei de-a doua dimensiuni a tabloului.
- În această situaţie:
  - (1) **Înscrierea** studentului `s` la concursul `c` se realizează simplu [9.4.2.2.a].

```
------------------------------------------------------
    INSCRIERE [AS(s), AC(c)]= true          [9.4.2.2.a]
------------------------------------------------------
```

- (2) **Retragerea** studentului `s` de la cursul `c` se implementează exact la fel atribuind însă constanta booleană false.
- (3) Pentru a afla **concursurile** la care s-a înscris **un student** cu numele `s` se parcurge linia `AS(s)`a tabloului `INSCRIERE`.

- (4) Pentru a afla **studenţii** înscrişi la **concursul** `c` se parcurge coloana `AC(c)` a tabloului `INSCRIERE`.

- Această abordare conduce la o implementare simplă şi performantă, în schimb consumul de mare de memorie este mare şi gradul de utilizare redus.

- Presupunând că în universitate există în total aproximativ 6000 de studenţi care pot participa la 20 de concursuri diferite, avem nevoie de un tablou care ocupă 120.000 de elemente.

- Deoarece mai puţin de 20% din numărul studenţilor participă de fapt la astfel de concursuri, marea majoritate la o singură disciplină, rezultă că structura tablou este foarte slab utilizată (sub 6%).

- O astfel de matrice se numeşte **rară** ("**sparse**"), parcurgerea ei presupunând un interval considerabil de timp şi în acelaşi timp, memorarea ei, o mare risipă de memorie.

- Ca atare se investighează şi alte posibilităţi de implementare.

9.4.2.3. Implementarea relaţiei bazate pe corespondenţe multiple cu ajutorul mulţimilor. Variante de implementare

- O metodă mai bună de a rezolva această problemă, este aceea de a implementa relaţia bazată pe corespondenţe multiple ca şi o **colecţie de mulţimi**.

- Două dintre aceste mulţimi sunt `S` reprezentând mulţimea tuturor **studenţilor** şi `C` reprezentând mulţimea tuturor **concursurilor**.

  - Fiecare element al lui `S` este o structură `TipStudent1`.

  - Fiecare element al lui `C` este o structură `TipConcurs1`.

```
------------------------------------------------------------
{Implementarea relaţiei bazate pe corespondenţe multiple cu
ajutorul mulţimilor Varianta 1}

type TipStudent1=record
                   marca: integer;
                   numeStudent: string[20]
                 end;

     TipConcurs1=record                              [9.4.2.3.a]
                   dataC: TipData;
                   numeConcurs:  string[15]
                 end;

     TipInscriere1=record
                     student: TipStudent1;
                     concurs: TipConcurs1
                   end;
```

- Pentru a implementa relaţia dorită este necesară o a treia mulţime `I` care implementează **înscrierea**.

    - Elementele mulţimii `I` sunt structuri `TipInscriere1`, care realizează **asocierea** student-concurs.

    - În mulţimea `I` există câte un element pentru fiecare locaţie marcată cu `x` în structura tablou `INSCRIERE` prezentată în figura 9.4.2.2.a.

- În plus pentru a rezolva **problema corespondenţelor multiple** este necesară precizarea unor **mulţimi suplimentare**. Este vorba despre:

    - (1) Mulţimile C_S – câte o mulţime pentru fiecare student `s`, mulţime care include concursurile la care acesta participă.

    - (2) Mulţimile S_C - câte o mulţime pentru fiecare concurs `c`, incluzând mulţimea studenţilor înscrişi la concursul respectiv.

- Astfel de mulţimi ridică însă probleme de implementare din cauza:

    - Numărului mare de mulţimi C_S.

    - Numărului mare de elemente din mulţimea S_C.

    - Naturii diferite a elementelor celor două tipuri de mulţimi: structuri `TipStudent1` respectiv `TipConcurs1`.

- Se pot concepe mai multe **variante de implementare** a mulţimilor C_S respectiv S_C .

- (1) O **primă variantă,** numită **varianta 0**, constă în implementarea mulţimilor C_S şi S_C ca şi **mulţimi de indicatori** la structuri `TipConcurs1` respectiv la structuri `TipStudent1`.

- (2) O altă **variantă** care economiseşte spaţiu şi în acelaşi timp permite precizarea rapidă a relaţiilor studenţi-concursuri este următoarea.

    - Atât mulţimile C_S cât şi mulţimile S_c sunt concepute unitar ca fiind alcătuite din structuri `TipInscriere1`, fiecare structură precizând studentul `s` şi concursul `c` la care acesta s-a înscris [9.4.2.3.a].

    - Formal, definirea acestor mulţimi este cea precizată în [9.4.2.3.b].

```
------------------------------------------------------------
CS ={(s,c) | s s-a înscris la concursul c, s=ct }
                                                 [9.4.2.3.b]
SC ={(s,c) | s s-a înscris la concursul c, c=ct }
------------------------------------------------------------
```

- Referitor la secvenţa [9.4.2.3.b] se face precizarea că `(s,c)` e de fapt un articol de `TipInscriere1`.

- În concluzie, deşi au componente de aceeaşi tip, cele două tipuri de mulţimi se **diferenţiază** prin aceea că într-o mulţime C_S, `s` este constant, iar într-o mulţime S_C, `c` este constant.

- Pentru implementarea acestor mulţimi aferente relaţiilor bazate pe corespondenţe multiple se pot utiliza în mod avantajos **structurile de date multilistă**.

  - Se reaminteşte că o **multilistă** este o colecţie de noduri dintre care unele au mai mult decât un pointer şi pot face parte simultan din mai multe liste.

  - Pentru fiecare tip de nod aparţinând unei structuri multilistă este important să se precizeze cu claritate numele şi semnificaţia pointerilor implicaţi, nodurile putând diferi ca structură.

- Pornind de la această abordare denumită **varianta 1** de implementare, se rafinează **varianta 2** (secvenţa[9.4.2.3.c]) în care:

  - (1) Structura `TipInscriere1` se modifică în `TipInscriere2`, care constă din două câmpuri indicator:

    - Câmpul `concursUrm` indicând elementul următor de tip înscriere din mulţimea C_S căreia îi aparţine.

    - Câmpul `studUrm` indicând elementul următor de tip înscriere din mulţimea S_C corespunzătoare [9.4.2.3.c].

  - (2) Structurii `TipStudent1` i se ataşează un indicator care precizează primul concurs la care s-a înscris studentul în cauză, adică prima structură de tip `TipInscriere2` din mulţimea C_S respectivă şi devine structura `TipStudent2`.

  - (3) Structurii `TipConcurs1` i se ataşează un indicator care precizează primul student care s-a înscris la concursul în cauză, adică prima structură de tip `TipInscriere2` din mulţimea S_C asociată şi devine structura `TipConcurs2`.

```
------------------------------------------------------------
{Implementarea relaţiei bazate pe corespondenţe multiple
           Varianta 2}

type RefTipInscriere2 = ^TipInscriere2;

     TipStudent2=record
                   marca: integer;
                   numeStudent: string[20];
                   concurs: RefTipInscriere2
                 end;

     TipConcurs2=record                             [9.4.2.3.c]
                   dataC: TipData;
                   numeConcurs:  string[15];
                   student: RefTipInscriere2
                 end;
```

```
   TipInscriere2=record
                   concursUrm: RefTipInscriere2;
                   studentUrm: RefTipInscriere2
                 end;
------------------------------------------------------------
```

* Se constată faptul că un articol de `TipInscriere2` aparţine în acelaşi timp la două liste înlănţuite distincte (fig.9.4.2.3.a).

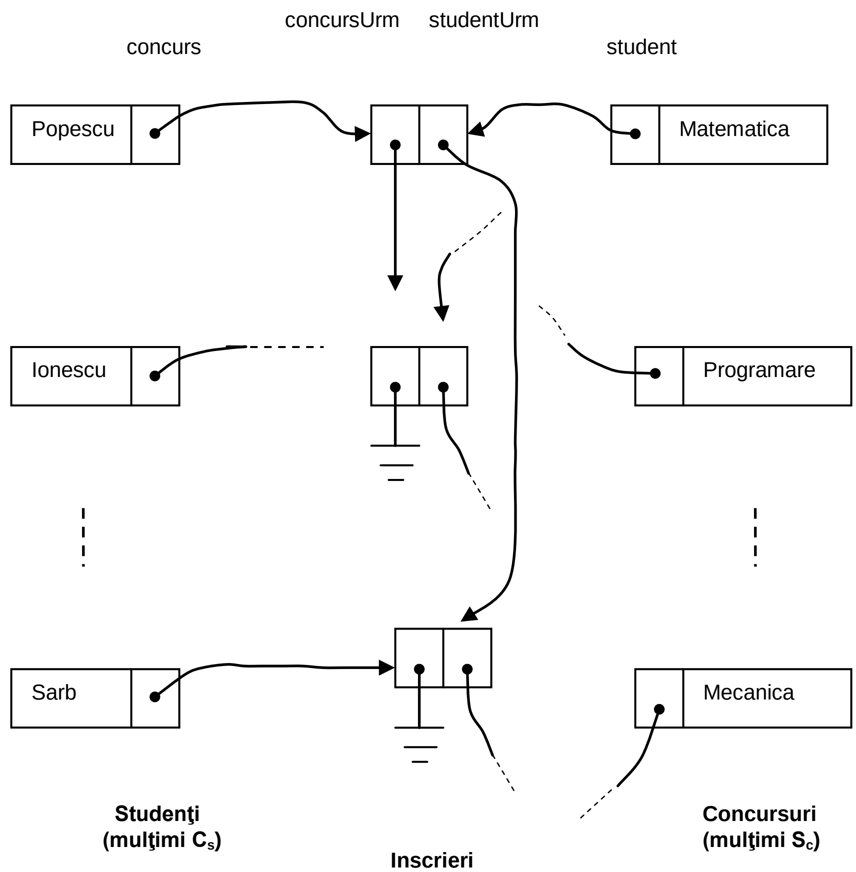

Fig.9.4.2.3.a. Implementarea relaţiei bazate pe corespondenţe multiple, varianta 2

* Fiecare structură `TipStudent2`, este începutul unei liste care materializează mulţimea C_S corespunzătoare, adică mulţimea concursurilor la care s-a înscris studentul în cauză.

* Fiecare structură `TipConcurs2`, este începutul unei liste care materializează mulţimea S_C corespunzătoare, adică mulţimea studenţilor care participă la concursul în cauză.

* Se reaminteşte faptul că, atât mulţimile S_C cât şi mulţimile C_S sunt formate din structuri `Inscriere2`.

- De fapt o structură `TipInscriere2` **nu** indică în mod explicit nici **studentul** nici **concursul** la care se referă.
    - Această informaţie rezultă în mod implicit din **lista** în care este înlănţuită structura respectivă.

- În continuare, structurile `TipStudent2`, respectiv structurile `TipConcurs2` se vor numi **proprietari** ai structurilor `TipInscriere2` care aparţin listelor pe care le iniţiază.

- Astfel pentru a se preciza la ce concursuri participă un anumit student `s`:
    - Trebuie parcurse structurile `TipInscriere2` din mulţimea C_S (pointerul `concursUrm`) pornind de la structura `TipStudent` **proprietar**.
    - Pentru fiecare element parcurs, trebuie determinată structura `TipConcurs` **proprietară**.

- Se observă că structura de date propusă **varianta 2 nu** poate soluţiona simplu această cerinţă.

- Pentru soluţionarea acestei cerinţe se pot adăuga structurii înscriere încă doi **idicatori** unul pentru structura proprietar de `TipStudent`, celălalt pentru structura proprietar de `TipConcurs`, după cum se prezintă în **varianta 3** de implementare, secvenţa [9.4.2.3.d].

```
-------------------------------------------------------------
{Implementarea relaţiei bazate pe corespondenţe multiple
           Varianta 3}

type RefTipInscriere3 = ^TipInscriere3;

     TipStudent3=record
                   marca: integer;
                   numeStudent: string[20];
                   concurs: RefTipInscriere3
                 end;

     TipConcurs3=record
                   dataC: TipData;
                   numeConcurs:  string[15];
                   student: RefTipInscriere3
                 end;                             [9.4.2.3.d]

     TipInscriere3=record
                     concursUrm: RefTipInscriere3;
                     studentUrm: RefTipInscriere3;
                     studentProprietar: RefTipStudent3;
                     concursProprietar: RefTipConcurs3
                   end;
-------------------------------------------------------
```

- Varianta 3 de implementare apare reprezentată grafic în figura 9.4.2.3.b.

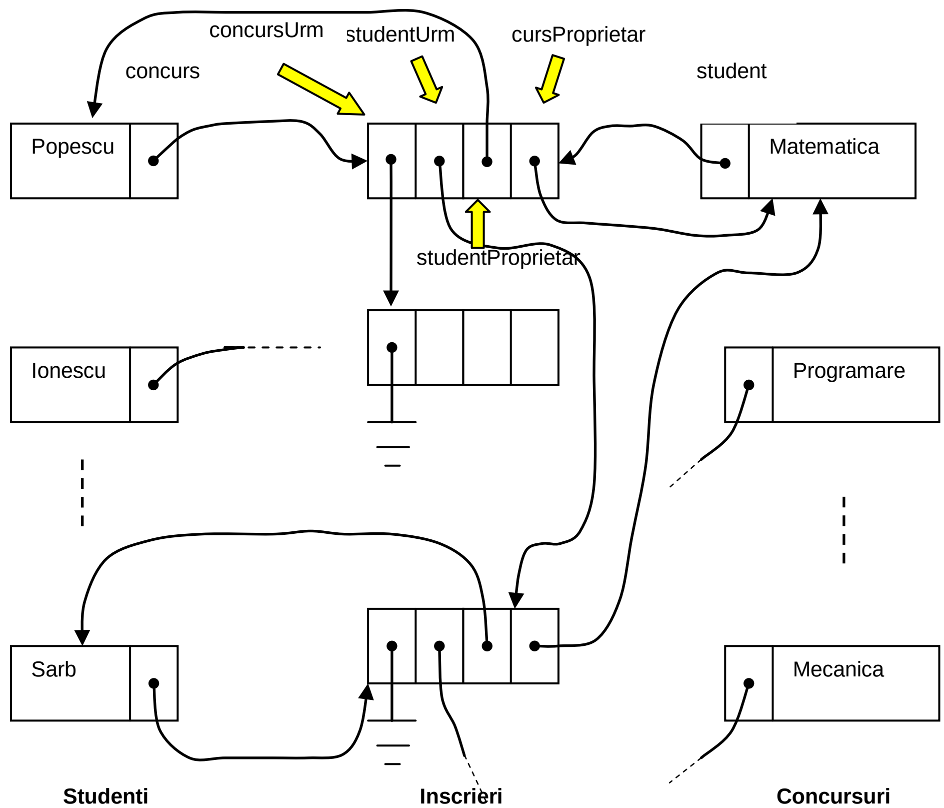

Fig.94.2.3.b. Implementarea relaţiei bazate pe corespondenţe multiple, varianta 3

- O astfel de structură de articol înscriere este **cea mai eficientă** din punct de vedere al performanţei în contextul avut în vedere.
    - Are însă dezavantajul că ocupă multă memorie.

- Se poate salva o cantitate substanţială de memorie, cu preţul sporirii rezonabile a timpului de acces utilizând următoarea metodă:
    - Se elimină ultimii doi pointeri din structura articolului `Inscriere3`.
    - Se plasează la sfârşitul fiecărei liste S_C un pointer la **concursul proprietar**
    - Se plasează la sfârşitul fiecărei liste C_S un pointer la **studentul proprietar**.
    - Astfel fiecare structură de `TipStudent` sau `TipConcurs` devine parte a unei **liste circulare** care include înregistrările a căror proprietar este.

- Acest lucru apare ilustrat în fig.9.4.2.3.c drept **varianta 4** de implementare.

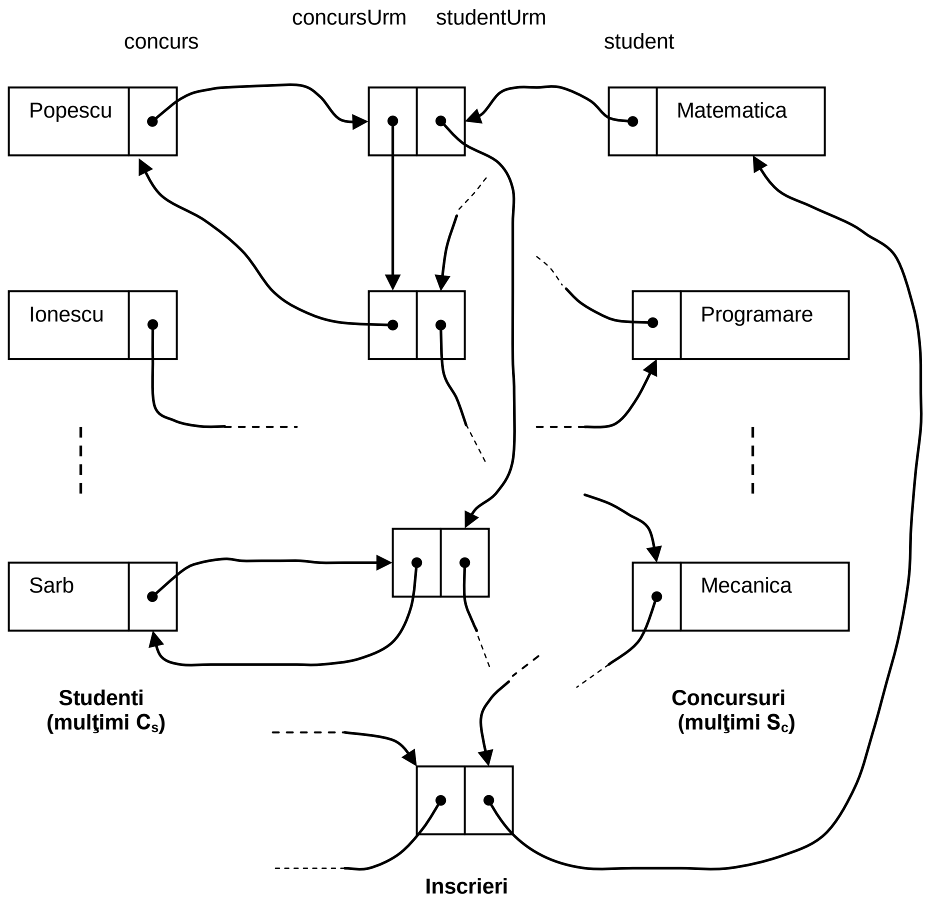

Fig.94.2.3.c. Implementarea relaţiei bazate pe corespondenţe multiple, varianta 4

- Dacă se doreşte spre **exemplu** să se afle mulţimea studenţilor înscrişi la concursul de matematică, se procedează după cum urmează:
  - (1) Se depistează mai întâi structura de `TipConcurs` Matematica a mulţimii concursurilor.
    - Modul în care se realizează acest lucru depinde de maniera de implementare a mulţimii `CONCURSURI`, spre exemplu ca lista înlănţuită sau ca tabelă de dispersie.
  - (2) Pornind de la pointerul cuprins în articolul Matematica se ajunge la prima structură `TipInscriere` din lista sa circulară.
  - (3) Pentru a afla studentul **proprietar** al acestei structuri de `TipInscriere` se urmăreşte câmpul `concursUrm` al structurii până când se găseşte o structură `TipStudent`.
  - (4) Pentru a afla restul studenţilor înscrişi la matematică, se înaintează pe înlănţuirea indicată de câmpul `studentUrm` începând cu prima structură `TipInscriere` şi pentru fiecare structură parcursă se aplică procedeul precizat anterior de determinare a studentului proprietar.

- (5) În final, urmând înlănţuirea indicată de câmpul `studUrm` se ajunge din nou la structura Matematica şi astfel lista urmată s-a închis.

- Operaţiile aferente listării studenţilor participanţi la concursul de Matematica, respectiv determinării studentului proprietar al unui articol de `TipInscriere` sunt descrise formal în secvenţele [9.4.2.3.e] respectiv [9.4.2.3.f].

```
------------------------------------------------------------
{Listare studenţi înscrişi la concursul de matematică}

   pentru fiecare articol de TipInscriere din lista Sc
             indicata de articolul Matematica executa
     begin                                       [9.4.2.3.e]
      *se atribuie lui s numele studentului proprietar
        al articolului de TipInscriere curent;
      afiseaza(s)
     end;
------------------------------------------------------------
{*se atribuie lui s numele studentului proprietar al
articolului de TipInscriere}

     atribuie p <- referinţa la articolul de TipInscriere
               curent;
     repeta                                    [9.4.2.3.f]
       atribuie p <- p^.concursUrm
     pana cand p este o referinta la un articol de
                 TipStudent;
     atribuie s <- numeStudent din articolul de TipStudent
               indicat de p}
------------------------------------------------------------
```

- Pentru a implementa o astfel de structură de tip multilistă în limbajul PASCAL se poate defini un singur tip de **articol cu variante** pentru cazurile `student`, `concurs` şi `înscriere` (secvenţa [9.4.2.3.g]).

    - Acest lucru este necesar deoarece câmpurile `concursUrm` respectiv `studUrm` pot indica articole de tipuri diferite.

- O posibilă implementare PASCAL a listării studenţilor care participă la un concurs precizat, apar în forma procedurii `ListareStudenti(numeC:TipC))` în secvenţa [9.4.2.3.g].

- În aceeaşi secvenţă apar precizate şi structurile de date aferente.

- Se face precizarea că această abordare poate fi utilizată drept model pentru o implementare C.

```
------------------------------------------------------------
{Implementarea relaţiei bazate pe corespondenţe multiple
           Varianta 4}

type TipS = string[20];
     TipC = string[15];
```

```
    TipFel = (student,concurs,inscriere);
    RefArticol = ^TipArticol;

    TipArticol = record
      case fel: TipFel of
        student: (numeStudent: TipS;
                  marca: integer;
                  primulConcurs: RefArticol);
        concurs: (numeConcurs: TipC;
                  dataC: TipData;
                  primulStudent: RefArticol);
        inscriere: (concursUrm,studUrm: RefArticol)
      end;

{Listarea studenţilor participanţi la un concurs precizat}

procedure ListareStudenti(numeC: TipC);          [9.4.2.3.g]
  var c,e,p: RefArticol;
  begin
    c:=pointer la articolul concurs pentru care
         c^.numeConcurs = numeC;  {depinde de implementare}
    e:=c^.primulStudent;
    while e^.fel = inscriere do
      begin
        p:=e;
        repeat
          p:=p^.concursUrm
        until p^.fel = student;
        write(p^.numeStudent);
        e:=e^.studUrm
      end
  end; {ListareStudenti}
------------------------------------------------------------
```

- Exemplul de implementare al unei relaţii bazate pe corespondenţe multiple prezentat în cadrul acestui paragraf, ilustrează faptul că **metoda de dezvoltare graduală pas cu pas** ("**stepwise refinement**") specifică **dezvoltării algoritmilor**, poate fi utilizată cu succes şi în cazul **dezvoltării de structuri de date optimizate**.

- Rezultatul este acela că, de cele mai mult ori structurile rezultate ajung să nu mai semene practic deloc cu modelul real pe care îl abstractizează.

9.4.2.4. Structuri de date combinate

- De cele mai multe ori implementarea reprezentării structurilor abstracte mulţime sau asociere ridică probleme deosebite.

    - Alegând o anumită reprezentare, anumite operaţii se implementează simplu şi performant altele în schimb necesită o regie ridicată.

    - De fapt ca şi în alte domenii, **nu** există soluţie care să aibă numai avantaje, respectiv o structură de date care să implementeze simplu şi în acelaşi timp performant toţi operatorii.

- În acest caz, o posibilitate de rezolvare a problemei constă în utilizarea a **două sau mai multe structuri de date diferite** pentru a reprezenta **o aceeaşi structură de date abstractă**.

- Se presupune spre exemplu că se doreşte menţinerea **unui clasament al jucătorilor de tenis**, în care fiecare jucător are poziţia sa unică.

- **Specificaţia de definire** a **clasamentului** este următoarea:

  - (1) Jucătorii ocupă **poziţiile** în clasament în ordine valorică descrescătoare.

  - (2) Un nou jucător este **adăugat** la sfârşitul clasamentului, deci pe poziţia cu numărul cel mai mare.

  - (3) Un jucător poate **provoca** la joc jucătorul situat pe poziţia anterioară poziţiei sale.

  - (4) Dacă câştigă meciul, cei doi jucători îşi **interschimbă** poziţiile în clasament.

- Această situaţie poate fi reprezentată cu ajutorul unei **structuri de date abstracte** pentru care **modelul** utilizat este o **asociere** între **nume de jucători** (reprezentate ca şiruri de caractere) şi **poziţiile acestora în clasament** (întregi).

  - **Operatorii** pe care îi suportă această structură de date sunt [9.4.2.4.a]:

```
------------------------------------------------------------
{Operatori definiţi pentru structura Clasament}

        1. Adauga(nume) – adaugă numele unui jucător pe
           poziţia cu numărul cel mai mare din clasament.

        2. poziţie Pozitie(nume) - returnează poziţia în
           tablou a jucatorului precizat ca parametru.
                                                [9.4.2.4.a]

        3. nume Provoaca(nume) - funcţie care returnează
           numele jucătorului de pe poziţia i-1, dacă
           poziţia jucătorului precizat ca parametru este i,
           i>1.

        4. Interschimba(pozitie) – interschimbă în clasament
           numele jucătorilor situaţi pe poziţiile i şi i-1,
           dacă poziţia precizată ca parametru este i, i>1.
------------------------------------------------------------
```

- În acest context se poate observa faptul că primii trei operatori au drept parametru un nume de jucător, în timp ce ultimul operator are drept parametru poziţia jucătorului care a lansat provocarea.

- (1) Clasamentul poate fi reprezentat spre exemplu printr-un **tablou** `Clasament`, unde `Clasament[i]` conţine numele jucătorului situat pe poziţia `i` în clasament (fig.9.4.2.4.a).

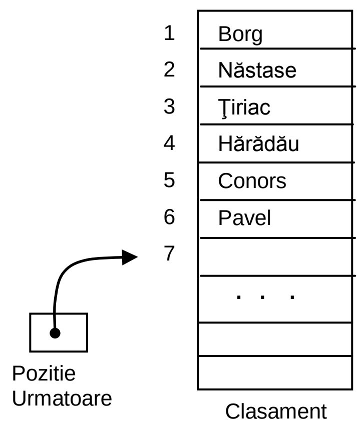

Fig.9.4.2.4.a. Implementarea clasamentului utilizând structura tablou

- Se mai utilizează variabila `PozitieUrmatoare` care precizează prima poziţie neocupată în cadrul clasamentului.
    - Cu ajutorul acestei variabile adăugarea unui nou jucător în clasament se face într-un număr constant şi reduşi de paşi.
- Operatorul ***`Interschimba`***_`(pozitie)`_ se implementează simplu cu performanţa O*(1)*.
- În schimb operatorii ***`Provoaca`***_`(nume)`_ şi ***`Pozitie`***_`(nume)`_ necesită o căutare în tabloul `Clasament` ceea ce necesită un efort de ordinul O*(n)* unde `n` este numărul de jucători participanţi.

- (2) Pe de altă parte, implementarea clasamentului poate fi realizată utilizând o **tabelă de dispersie deschisă** care reprezintă asocierea dintre nume şi poziţii (fig.9.4.2.4.b).

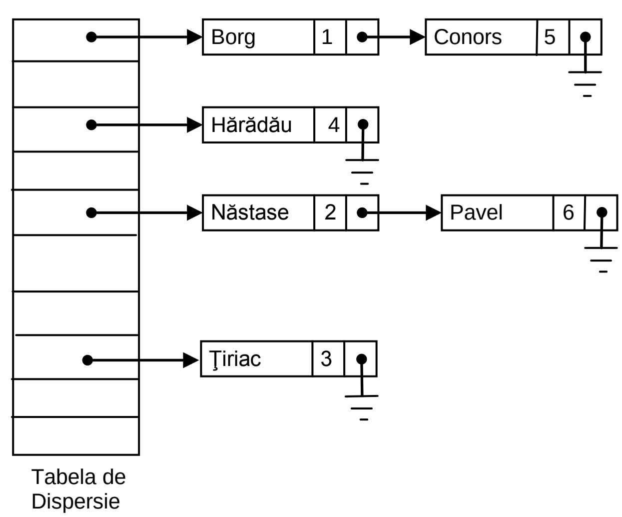

Fig.9.4.2.4.b. Implementarea clasamentului utilizând o tabelă de dispersie

- Se presupune că numărul de intrări în această tabelă este apropiat de numărul de jucători.
- Operatorul ***`Adauga`***_`(nume)`_ consumă în medie O*(1)* unităţi de tip.

- Operatorul ***Poziţie**(nume)* consumă de asemenea în medie O*(1)* unităţi de timp.

- ***Provoaca**(nume)* consumă O*(1)* unităţi de tip pentru căutarea numelui, dar necesită în schimb O*(n)* unităţi de tip pentru a găsi jucătorul situat pe poziţia anterioară, întrucât trebuie parcursă întreaga tabelă de dispersie.

- ***Interschimba**(pozitie)* necesită de asemenea O*(n)* unităţii de timp pentru a găsi poziţiile `i` şi `i-1`.

- (3) Se presupune în continuare că **se combină cele două structuri de date**, după cum rezultă din figura 9.4.2.4.c..

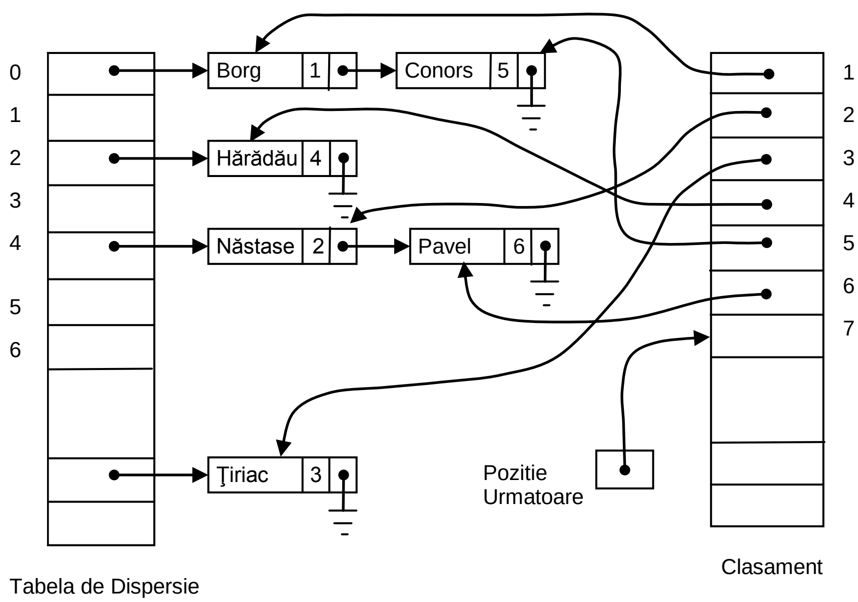

Fig.9.4.2.4.c. Implementarea clasamentului utilizând structuri de date combinate

- Celulele `Tabelei de Dispersie` vor conţine informaţii referitoare la numele jucatorului şi la poziţia sa în tabloul `Clasament`.

- Tabloul `Clasament` va conţine la `Clasament[i]` un pointer la celula corespunzătoare jucătorului din poziţia `i` în `Tabela de Dispersie`.

- În acest nou context :

  - Un nume de jucator nou se adaugă inserându-l în tabela de dispersie în O*(1)* unităţi de timp în medie şi de asemenea plasând pointerul celulei noi create în tabelul `Clasament` în poziţia marcată de cursorul `Pozitie Urmatoare`, într-un interval redus constant de timp (O*(1)*).

  - Operatorul ***Pozitie*** necesită de asemenea un efort de ordinul O*(1)* rezultat din accesul prin tabela de dispersie la jucatorul cu numele precizat şi aflarea poziţiei în clasament memorată în celula aferentă.

  - Pentru a implementa funcţia ***Provoaca***, se caută numele jucătorului în tabela de dispersie, în medie în O*(1)* unităţi de timp, se obţine rangul `i` al jucătorului căutat şi urmând indicaţia pointerului `Clasament[i-1]` se ajunge la celula conţinând numele jucătorului provocat.

- Consultarea lui `Clasament[i-1]` consumă un interval constant de timp, căutarea în tabela de dispersie în medie O*(1)* unităţi de timp, deci în medie ***`Provoaca`*** necesită O*(1)* unităţi de timp.

- ***`Interschimba`***`(`*`pozitie`*`)` consumă un interval constant de timp pentru a găsi celulele jucătorilor în tabloul `Clasament` situaţi pe *`pozitie`* respectiv *`pozitie-l`*, pentru a interschimba poziţiile în celulele corespunzătoare din `Tabela de Dispersie` şi pentru a interschimba pointerii la cele două celule în tabloul `Clasament`.

  - Astfel, şi operatorul ***`Interschimba`*** necesită un interval constant de timp chiar în cazul cel mai defavorabil.

- Acest exemplu, fără a fi deosebit de reprezentativ, are o **valoare intrinsecă deosebită**, el evidenţiind în **mod limpede şi clar** o **metodă conceptuală simplă**, care poate conduce la performanţe spectaculoase în ceea ce priveşte sporirea eficienţei operatorilor care acţionează asupra unei structuri de date, desigur într-un context specific.

- **Dezavantajul** este de asemenea evident: necesitatea multiplicării structurii în două sau mai multe reprezentări diferite care sunt prelucrate simultan.

9.5. Implementarea structuri mulţime cu ajutorul structurilor de date avansate

- În cadrul acestui paragraf vor fi precizate câteva modalităţi mai eficiente de implementare a structurii de date mulţime.

- Aceste modalităţi, mai complexe în principiu, se pretează implementării mulţimilor de mari dimensiuni şi sunt bazate pe diferite categorii de arbori cum ar fi **arborii binari ordonaţi, arborii de căutare** şi **arborii echilibraţi**.

9.5.1. Implementarea structuri mulţime cu ajutorul arborilor binari ordonaţi

- **Arborii binari ordonaţi** pot fi utilizaţi în reprezentarea acelor mulţimi peste care este definită o **relaţie de ordonare** precizată de regulă prin operatorul "<".

  - Această modalitate de reprezentare este utilă în cazul mulţimilor ale căror elemente aparţin unui **univers extins**, care face practic imposibilă utilizarea elementelor mulţimii în calitate de indici direcţi într-un tablou.

  - Un exemplu în acest sens îl constituie mulţimea identificatorilor posibili ai unui program.

- În implementarea bazată pe **arbori binari ordonaţi**, operatorii ***`Insereaza, Suprima, Apartine`*** şi ***`Min`*** pot fi implementaţi fiecare în O$(log_2 n)$ paşi în medie, pentru o mulţime de cardinalitate `n`, element care rezultă din maniera în care este concepută, reprezentată şi exploatată o astfel de structură.

- Structura **arbore binar ordonat** a fost prezentată detaliat în capitolul 8, paragraful &8.3.

    - Se reaminteşte că **proprietatea fundamentală** a acestui tip de arbore este aceea că toate elementele memorate în subarborele stâng al oricărui nod `x` sunt mai mici decât elementul memorat în `x` şi toate elementele memorate în subarborele drept, sunt mai mari ca elementul memorat în `x`.

    - Această proprietate, definitorie pentru **arborii binari ordonaţi** este valabilă pentru orice nod al unui astfel de arbore inclusiv pentru rădăcină.

- Se presupune spre exemplu, că reprezentarea unei mulţimi `dicţionar` se realizează cu ajutorul unei structuri **arbore binar ordonat**, ca cea prezentată în secvenţa [9.5.1.a].

    - După cum se observă tipul **mulţime** se declară ca şi un pointer la un nod. Acest nod este rădăcina arborelui binar ordonat care reprezintă mulţimea.

```
------------------------------------------------------------
/* Reprezentarea mulţimilor cu ajutorul arborilor binari
ordonaţi */

typedef struct tip_nod
          {
               tip_element element;
               struct tip_nod* stang;     /*[9.5.1.a]*/
               struct tip_nod* drept;
          };

typedef tip_nod * ref_tip_nod;

typedef ref_tip_nod tip_multime;
------------------------------------------------------------
```

- În acest context, implementarea operaţiei ***Apartine*** este simplă:

    - Pentru a determina apartenenţa lui `x` la mulţime se execută o căutare în arborele binar asociat bazată pe tehnicile clasice (&8.3.3).

    - O implementare recursivă a unei astfel de tehnici apare în funcţia ***Apartţine*(x,A)** din secvenţa [9.5.1.b].

```
------------------------------------------------------------
/*Implementarea operatorului Apartine*/

boolean Apartine(tip_element x, tip_multime A)

/*caută recursiv în arborele binar A elementul x şi
returnează true daca x apartine lui A sau false in caz
contrar*/

{
    boolean b;
    if (A==NULL)
```

```
        return false;
      else
        if (x==A->element)
            return true;
          else
            if (x<A->element)
                b=Apartine(x,b->stang);  /*[9.5.1.b]*/
              else
                if (x>A->element)
                  b=Apartine(x,A->drept);
 } {Apartine}
-------------------------------------------------------------
```

- Operatorul ***Inserează(x,A)*** care adaugă un element nou mulţimii este de asemenea simplu de implementat aplicând tehnicile de creare a arborilor binari ordonaţi.

    - O implementare posibilă apare prezentată în secvenţa [9.5.1.c]..

```
-------------------------------------------------------------
/*Implementarea operatorului Insereaza - varianta
pseudocod*/

void Insereaza(tip_cheie x, tip_multime A)

/*inserează elementul x în arborele binar ordonat A*/

    daca(A==NULL)
        /*insertie element x*/
        A=aloca_memorie(tip_multime); /*alocare nod nou*/
        A->element=x; A->stang=NULL; A->drept=NULL;
        □
      altfel                        /*[9.5.1.c]*/
        daca(x<A->element)  /*parcurgere arbore binar*/
            Insereaza(x,A->stang);
          daca
              Insereaza(x,A->drept);
            else
              return;  /*daca x==A^.element, x aparţine
                           deja mulţimii şi nu se face nimic*/
/*Insereaza*/
----------------------------------------------------------
```

- După cum se observă, dacă elementul de inserat `x`, aparţine deja mulţimii, procedura ***Inserează*** nu face nimic.

- **Suprimarea** unui nod dintr-o structură arbore binar ordonat ridică unele probleme presupunând tratarea distinctă a cazurilor în care nodul de suprimat are:

    - (1) Un fiu sau niciunul.

    - (2) Doi fii.

- Prima situaţie nu ridică probleme, cea de-a doua însă necesită o prelucrare specială.

- O metodă de rezolvare a acestei situaţii o reprezintă înlocuirea nodului de suprimat cu **predecesorul său direct** la ordonarea în **inordine** şi suprimarea nodului corespunzător precedesorului care nu are fiu drept, metodă prezentată în &8.3.5.

- O metodă similară, presupune înlocuirea nodului de suprimat cu **succesorul său direct** la ordonarea în **inordine** a cheilor şi **suprimarea succesorului** care este un nod fără fiu stâng.

  - Aflarea **succesorului** se reduce de fapt la aflarea celui mai mic nod (cel mai din stânga) al **subarborelui drept** al arborelui care are drept rădăcină nodul de suprimat.

  - În acest scop se dezvoltă funcţia ***SupriMin*** care returnează elementul minim al unui arbore suprimând nodul care-i corespunde (secvenţa [9.5.1.d]).

  - În continuare bazat pe această metodă, se prezintă procedura ***Suprimă*** care suprima un element precizat `x` din arborele `A` (secvenţa [9.5.1.e]).

  - Se observă clar tratarea celor trei situaţii: nici un fiu, un fiu, sau doi fii.

    - În ultima situaţie se înlocuieşte nodul cu succesorul său direct determinat de funcţia `SupriMin(A^.drept)`, adică cu nodul având cheia cea mai mică aparţinând **subarborelui drept** al nodului în cauză.

```
---------------------------------------------------------------
/*Implementarea operatorului SupriMin - varianta pseudocod*/

 tip_element SupriMin(tip_multime A)
 /*returnează şi suprimă cel mai mic element din A*/

    tip_element x;
    daca (A->stang==NULL)
          /*A indică cel mai mic element*/
          x=A->element;                          /*[9.5.1.d]*/
          A=A->drept;    /*suprimare nod*/
          return x;
          □
      altfel
        x=SupriMin(A->stang);
/*SupriMin*/
---------------------------------------------------------------
/*Implementarea operatorului Suprima - varianta pseudocod*/

void Suprima(tip_element x, tip_multime A);

    daca (A!=NULL)
      daca (x<A->element)
          Suprima(x,A->stang);
        altfel
          daca (x>A->element)                    /*[9.5.1.e]*/
              Suprima(x,A->drept);
            altfel  /*pointerul A indică nodul x*/
```

```
                    daca (A->stang==NULL)&&(A->drept==NULL)
                        A=NULL; /*ambii fii lipsesc*/
                      altfel
                        daca (A->stang==NULL)
                            A=A->drept;
                          altfel
                            daca (A->drept==NULL)
                                A=A->stang;
                              altfel /*x are ambii fii*/
                                A->element=SupriMin(A->drept);
/*Suprima*/
------------------------------------------------------------
```

- În figura 9.5.1.a se prezintă un exemplu de arbore binar (a) din care s-a suprimat cheia 9 conform metodei prezentate (b).

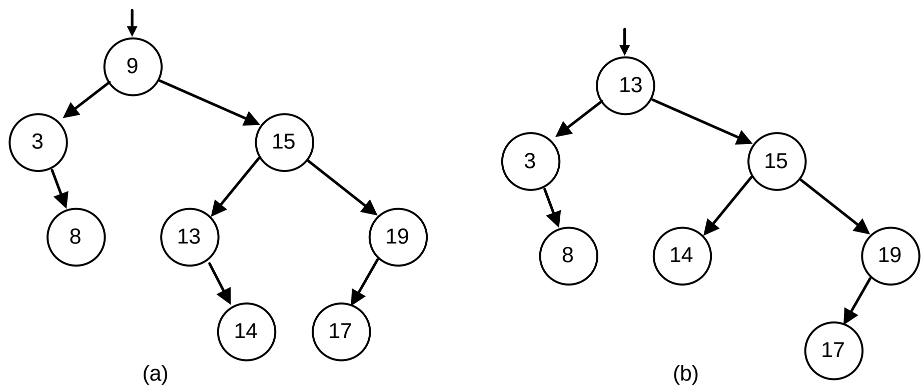

Fig.9.5.1.a. Suprimarea unui nod dintr-o structură de arbore binar ordonat

9.5.1.1. Consideraţii referitoare la performanţele implementării structurii mulţime cu ajutorul arborilor binari ordonaţi

- Este uşor de remarcat că dacă arborele binar cu `n` noduri care reprezintă **structura mulţime** este un **arbore binar complet** (toate nodurile cu excepţia celor de pe ultimul nivel au câte doi fii), atunci operaţiile ***Insereaza***, ***Suprima***, ***Apartine*** şi ***Min*** necesită cel mult $O(log_2\ n)$ paşi, întrucât în cel mai rău caz aceste operaţii necesită parcurgerea tuturor nivelurilor arborelui.

- De regulă, însă, arborii binari ordonaţi a căror formă depinde de ordinea de inserare a cheilor, nu sunt arbori binari compleţi.
  - În cel mai defavorabil caz, un astfel de arbore poate degenera într-o structură **listă liniară** în care operaţiile de mai sus necesită $O(n)$ paşi, pentru un arbore cu `n` noduri.

- Cu alte cuvinte în realitate, operatorii ***Insereaza, Suprima, Apartine*** şi ***Min*** implementaţi cu ajutorul **structurii arbore binar ordonat** necesită un efort de calcul cuprins între $O(log_2 n)$ şi $O(n)$.

- Analiza detaliată a căutării în arborii binari ordonaţi prezentată în &8.3.6 demonstrează însă că în medie, arborii binari ordonaţi reali sunt cu 39% mai înalţi ca şi arborii binari compleţi.

- Concluziile prezentate în cadrul respectivei analize rămân valabile şi pentru situaţia de faţă.

- Astfel se poate concluziona că **testarea apartenenţei** unui element la o mulţime, **inserţia** unui nou element într-o mulţime, **suprimarea** unui element al unei mulţimi şi **căutarea** elementului minim al unei mulţimi necesită în medie O(log_2 n) paşi în implementarea mulţimilor bazată pe **arbori binari ordonaţi.**

- În comparaţie cu implementările bazate pe **alte tipuri de structuri de date** se pot face următoarele observaţii.

- (1) Implementarea bazată pe **tabele de dispersie** a structurii dicţionar necesită în medie un timp constant pentru operaţiile definite.

  - Deşi această performanţă este superioară cele obţinute în implementările bazate pe arbori binari ordonaţi, o **tabelă de dispersie** necesită O(n) paşi pentru implementarea operaţiei ***`Min`***.

  - Astfel, dacă operaţia ***`Min`*** se utilizează des, atunci **arborii binari ordonaţi** sunt mai potriviţi, dacă ***`Min`*** nu este necesară se preferă **tehnica dispersiei**.

- (2) Referitor la implementarea structurii dicţionar, structura **arbore binar ordonat** poate fi comparată şi cu structura **arbore binar parţial ordonat** (ansamblu) utilizat în implementarea cozilor bazate pe prioritate.

  - Într-un **arbore binar parţial ordonat** (ansamblu) cu n elemente sunt necesari O($log_2 n$) paşi pentru operatorii ***`Insereaza`*** şi ***`Extrage`*** (suprimă minimul) nu numai în medie ci chiar în cel mai defavorabil caz, astfel din acest punct de vedere **arborii binari parţial ordonaţi** sunt de preferat.

  - Totuşi **arborii binari ordonaţi** permit implementarea la fel de performantă a operaţiilor ***`Suprima`***, ***`Min`*** cât şi a combinaţiei ***`SupriMin`***, în timp ce **arborii parţial ordonaţi** permit numai ultimele două operaţii.

  - În plus, ***`Aparţine`*** necesită O(n) paşi într-un **arbore binar parţial ordonat** respectiv numai O(log_2 n) paşi în **arborii binari ordonaţi**.

  - Astfel, dacă **arborii binari parţial ordonaţi** sunt foarte potriviţi în implementarea cozilor bazate pe prioritate, ei **nu** pot fi utilizaţi cu aceeaşi eficienţă ca şi **arborii binari ordonaţi** dacă este necesară implementarea unui set extins de operatori (ca şi în cazul mulţimilor).

9.5.2. Implementarea structurii mulţime cu ajutorul arborilor de regăsire

- După cum s-a mai precizat, **arborii de regăsire** sunt structuri de date speciale care pot fi utilizate în repezentarea mulţimilor cu deosebire a acelora ale căror elemente sunt caractere.

- De asemenea cu ajutorul lor pot fi reprezentate tipuri de date care sunt **şiruri de obiecte** de orice tip sau şiruri de numere.

* În literatura de specialitate această categorie de arbori de regăsire sunt cunoscuţi sub denumirea de structuri de tip **trie** cuvânt derivat din cuvântul **"retrieval"** (**regăsire**) (paragraful &8.4).

* Aşa cum s-a precizat în paragraful sus amintit, un **arbore de regăsire** permite implementarea simplă a operatorilor definiţi asupra unei **structuri de date mulţime** ale cărei elemente sunt **şiruri de caractere** (cuvinte).
  * Este vorba în principiu despre operatorii ***`Inserează`***, ***`Suprimă`***, ***`Iniţializează`*** şi ***`Afişează`***.
  * Ultimul operator realizează afişarea tuturor membrilor (cuvintelor) mulţimii.

* Utilizarea arborilor de regăsire este eficientă atunci când există mai multe cuvinte care încep cu aceeaşi secvenţă de litere, adică atunci când **numărul de prefixe distincte** al tuturor cuvintelor din mulţime este mult mai redus decât **numărul total de cuvinte.**

* În paragraful 8.4 se prezintă la nivel de detaliu structura de date `Nod ArboreDeRegasire` utilizată ca suport de reprezentare pentru structura `Arbore DeRegăsire`.
  * În acest context s-au prezentat două modalităţi de implementare una bazată pe **tablouri** şi o a doua bazată pe **liste liniare**.
  * Pentru exemplificare s-a prezentat implementarea operatorului ***`Insereaza`***, iar în finalul paragrafului menţionat s-a realizat o analiză comparată a performaţelor structurii `ArboreDeRegăsire` respectiv a tehnicii dispersiei în contextul utilizării lor la implementarea mulţimilor de cuvinte.
  * Analiza realizată a evidenţiat faptul că **arborii de regăsire** pot fi mai performanţi decât **tablele de dispersie**, ei constituind o posibilitate preferenţială de implementare eficientă a **structurii dicţionar**.

9.5.3. Implementarea structurii mulţime cu ajutorul arborilor binari ordonaţi echilibraţi

* Este cunoscut faptul că performanţa prelucrării unor **structuri de date arbore** este proporţională cu înălţimea acestora.

* În situaţia în care un arbore cu `n` noduri este un **arbore binar plin**, **complet** sau în general **de înălţime minimă**, performanţa prelucrării obţine în cel mai defavorabil caz, valoarea $O(log_2\, n)$.

* Întrucât în procesul de exploatare, înălţimea arborilor evoluează în mod aleator şi ea este în medie cu 39 % mai mare ca şi a arborilor de înălţime minimă (&8.3.6), au fost dezvoltate structuri speciale de arbori a căror înălţime evoluează în mod controlat şi este menţinută în jurul înălţimii minime.

* În această categorie se încadrează **arborii binari ordonaţi echilibraţi** pentru care efortul de prelucrare este în cel mai defavorabil caz $O(log_2\, n)$, în medie mai redus.

* Din păcate complexitatea algoritmilor care prelucrază astfel de arbori **nu** justifică întotodeauna utilizarea lor.

- În această categorie se includ **arborii AVL, arborii binari optimi, arborii B, arborii 2-3** etc.

- Fiecare dintre aceste **structuri arbore** a fost prezentată la nivel de detaliu din punctul de vedere al proprietăţilor specifice, al modalităţilor de implementare şi al performanţelor aferente.

- Ca atare, ori care dintre aceste **structuri arbore** poate fi utilizată pentru a implementa în mod performant, într-un context specific, **structura de date mulţime.**

9.6. Mulţimi pe care sunt definiţi operatorii UNIUNE şi CAUTĂ

- În continuare se propune modelarea următorului **scenariu**:

  - Se presupune că există **o colecţie de obiecte unice** fiecare dintre ele aparţinând unei anumite **mulţimi**.

  - Se **combină** aceste mulţimi într-o ordine oarecare prin operaţii ***`Uniune`*** şi din timp în timp se cere să se precizeze **căreia dintre mulţimi** îi aparţine un **obiect specificat**.

- Aceast scenariu poate fi soluţionat utilizând mulţimi pe care sunt definite operaţiile ***`Uniune`*** şi ***`Cauta`***.

- Se reaminteşte definirea celor doi operatori:

  - Operatorul ***`Uniune`***`(A,B,C)` atribuie lui `C` mulţimea rezultată din reuniunea mulţimilor `A` şi `B` care sunt **mulţimi disjuncte** (nu au elemente în comun) (vezi &9.2).

    - Operatorul ***`Uniune`*** **nu** este definit dacă `A` şi `B` **nu** sunt disjuncte, element care diferenţiază această operaţie de reuniune normală a mulţimilor.

  - ***`Cauta`***`(x)` este o funcţie care returnează numele (unic) al mulţimii căreia îi aparţine obiectul `x`.

    - Dacă `x` apare în mai multe mulţimi sau în niciuna, operatorul ***`Cauta`*** nu este definit (vezi &9.2)..

9.6.1. Implementarea bazată pe tablouri a mulţimii pe care sunt definiţi operatorii UNIUNE şi CAUTA

- Se consideră o structură abstractă de date `MulţimeDeSubmulţimi` constând dintr-**o mulţime de submulţimi disjuncte** care se vor numi **componente**, peste care sunt definiţi următorii **operatori** [9.6.1.a]:

----

1. ***Uniune***(*TipNumeSubmulţime A, TipNumeSubmulţime B*) - realizează uniunea componentelor *A* şi *B* denumind rezultatul uniunii *A* sau *B*, în mod arbitrar. [9.6.1.a]

2. *TipNumeSubmulţime* ***Caută***(*TipElementSubmulţime x*) - este o funcţie care returnează numele componentei căreia îi aparţine elementul *x*.

3. ***Intializeaza***(*TipNumeSubmulţime A, TipElementSubmulţime x*) - creează o submulţime componentă numită *A*, care conţine numai elementul *x*.

---

* Pentru a realiza o implementare rezonabilă a structurii `MultimeDeSubmultimi` este necesar să se observe că tipul **mulţime de submulţimi** include alte două tipuri:

    * (1) Tipul corespunzător **numelui submulţimilor componente** - `TipNumeSubmultime`.

    * (2) Tipul corespunzător **elementelor submulţimilor** - `TipElementSubmultime`.

* În cele ce urmează, se vor utiliza **întregi** pentru a preciza atât **numele submulţimilor componente** cât şi **elementele** acestora.

    * Dacă `n` este numărul total de elemente, atunci elementele submulţimilor aparţin **domeniului** `0..n-1`.

    * Pentru implementare, este important ca tipul elementelor să fie un tip **subdomeniu**, deoarece el va fi utilizat ca indice într-un **tablou** care materializează corespondenţa element-submulţime.

        * Elementele acestui tablou sunt **nume de submulţimi**.

        * Astfel dacă `M` este o variabilă încadrată în tipul `MulţimeDeSubmulţimi`, atunci `M[i]=j` precizează că **elementul** `i` aparţine **submulţimii cu numele** `j`.

    * Tipul corespunzător numelor submulţimilor componente în principiu **nu** este supus restricţiilor, întrucât constantele sale sunt elemente de tablou şi nu indici.

    * Trebuie subliniat faptul că dacă tipul elementelor submulţimilor este altul decât tipul subdomeniu, trebuie definită o **asociere**, eventual printr-o tabelă de dispersie, care să realizeze o corespondenţă de genul element-indice de tablou.

* Cunoscând în avans numărul total de elemente, se poate defini drept suport al reprezentării mulţimilor pe care sunt definiţi operatorii ***Uniune*** şi ***Cauta*** structura de date din secvenţa [9.6.1.a] sau varianta generalizată a acesteia din secvenţa [9.6.1.b]

```
/*Mulţimi pe care sunt definiţi operatorii UNIUNE şi CAUTA
```

```
    Implementare bazata pe tablouri varianta C*/

#define N {număr total de elemente};              [9.6.1.a]

typedef int Multime_De_Submultimi[N];

Multime_De_Submultimi M;
------------------------------------------------------------
{Mulţimi pe care sunt definiţi operatorii UNIUNE şi CAUTA
      Implementare bazata pe tablouri – varianta PASCAL}

CONST n = {număr total de elemente};               [9.6.1.a]

type MultimeDeSubmultimi = array[1..n] OF INTEGER;
------------------------------------------------------------
{Mulţimi pe care sunt definiţi operatorii UNIUNE şi CAUTA
   Implementare bazata pe tablouri   Varianta generalizată
PASCAL}

type MultimeDeSubmultimi = array[SubdomeniuElemente] OF
      TipNumeSubmultime;
                                                     [9.6.1.b]
var M: MultimeDeSubmultimi;
------------------------------------------------------------
```

- În continuare, se presupune că se declară variabila `M` de tip `MulţimeDeSubmulţimi`, cu precizarea că `M[x]` memorează numele submulţimii căreia îi aparţine la momentul considerat elementul `x`.

- Operatorii ***Uniune,Cauta*** şi ***Initializeaza*** sunt simplu de implementat.

    - Spre exemplu implementarea operatorul ***Uniune*** apare în secvenţa [9.6.1.c].

```
------------------------------------------------------------
/*Implementarea operatorului Uniune*/

subprogram Uniune(TipNumeSubmultime A, TipNumeSubmultime B,
                   MultimeDeSubmultimi M)

  int x;
  pentru (x=0 la n-1)
    daca (M[x]==B)                                  [9.6.1.c]
       M[x]= A;
/*Uniune*/
------------------------------------------------------------
```

    - Într-o manieră similară, ***Initializeaza****(A,x)* atribuie lui `M[x]` valoarea `A`.

    - ***Cauta****(x)* returnează valoarea lui `M[x]` .

- Performanţa raportată la timp a acestei implementări poate fi simplu apreciată. Astfel:

* Execuţia lui ***`Uniune`*** consumă O(n) unităţi de timp.

* ***`Cauta`***`(x)` şi ***`Initializeaza`***`(A,x)` consumă intervale constante de timp de execuţie (O(1)).

9.6.2. Implementarea bazată pe liste înlănţuite a mulţimii pe care sunt definiţi operatorii UNIUNE şi CAUTA

* Se porneşte de la observaţia că în cel mai defavorabil caz pentru a introduce `n` elemente definite ca mai sus într-o singură submulţime, sunt necesare `n-1` execuţii ale operatorului ***`Uniune`***, primul element al fiecărei mulţimi fiind introdus printr-o operaţie ***`Initializeaza`***.

    * Utilizând algoritmul din secvenţa anterioară [9.6.1.c], cele `n-1` execuţii vor consuma O(n^2) unităţi de timp.

* O cale de creştere a vitezei de execuţie a operatorului ***`Uniune`*** este aceea de a lega într-o **listă înlănţuită** toţi membrii unei submulţimi.

    * Astfel, în loc de a parcurge toţi membrii ambelor submulţimi la realizarea uniunii `A` cu `B`, se va parcurge numai lista elementelor lui `B` care vor fi introduse în mulţimea `A`.

* În medie această soluţie salvează timp dar şi aici poate apare o situaţie defavorabilă.

    * Dacă se presupune că la cea de-a `i`-a execuţie a operatorului se execută ***`Uniune`***`(A,B)` unde `A` este o submulţime de dimensiune `1`, `B` este o submulţime de dimensiune `i`, iar rezultatul va fi mulţimea `A`, această operaţie presupune O(i) unităţi de timp.

    * Cel mai defavorabil caz constă dintr-o secvenţă de `n-1` astfel de uniuni, în care de fiecare dată se adaugă unei submulţimi cu **un element**, o altă submulţime care după fiecare execuţie creşte cu un element.

    * Timpul necesar execuţiei unei astfel de secvenţe cel precizat în [9.6.2.a] adică un efort de ordinul O(n^2).

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \qquad \texttt{[9.6.2.a]}$$

----

* Pentru a evita această situaţie defavorabilă, implementarea operatorului ***`Uniune`*** trebuie să realizeze întotodeauna mutarea elementelor submulţimii cu un număr mai mic de elemente în cea cu un număr mai mare.

- Prin urmare, de fiecare dată când un element este adăugat unei submulţimi prin operaţia ***Uniune***, el va aparţine unei submulţimi de cel puţin două ori mai mari ca şi cea din care provine.

- Astfel, dacă există iniţial `n` submulţimi fiecare cu câte un element, nici unul din cele `n` elemente nu îşi poate schimba submulţimea căreia îi aparţine de mai mult de $1+\log_2 n$ ori.

- Întrucât timpul necesar execuţiei noii versiuni a operaţiei ***Uniune*** este proporţional cu numărul de elemente care îşi schimba numele submulţimii căreia îi aparţin, rezultă că **numărul total** de astfel de schimbări este cel mult $n*(1+\log_2 n)$.

- Drept consecinţă, toate operaţiile de tip ***Uniune*** necesare pentru crearea unei mulţimi cu `n` elemente, vor necesita maximum $O(n \log_2 n)$ unităţi de timp.

- Pentru implementarea acestei soluţii se propun următoarele **structuri de date**:

  - (1) Tabloul `inceputuri` care conţine câte o intrare pentru fiecare submulţime. Fiecare element al acestui tablou este un articol având următoarea structură:

    1. Câmpul `contor` care precizează numărul de elemente ale submulţimii.

    2. Câmpul `primulElement` de tip index în tabloul `nume`, câmp care indică primul element al listei membrilor submulţimii.

  - (2) Tabloul `nume` păstrează membrii submulţimilor ca liste înlănţuite prin intermediul cursorilor. Fiecărui element aparţinînd unei submulţimi îi corespunde o intrare în tablou cu următoarea structură:

    1. Câmpul `numeSubmulţime` care păstrează numele submulţimii căreia îi aparţine elementul respectiv.

    2. Câmpul de înlănţuire `urm` de tip index în tabloul `nume`, care precizează următorul membru al submulţimii respective. Indexul corespunzător lui `NULL` se notează cu `0` (zero).

- În cazul special în care atât numele submulţimilor cât şi elementelor lor aparţin subdomeniului `1..n`, pentru implementarea mai sus descrisă se pot defini structurile de date din secvenţa [9.6.2.a].

```
/*Mulţimi pe care sunt definiţi operatorii Uniune şi Cauta
  Implementare bazată pe liste înlănţuite implementate cu
  ajutorul cursorilor – structuri de date - varianta C*/

#define N {număr total de elemente};

typedef struct tip_multime
    {
      int contor;/*numărul curent de elemente al mulţimii*/
```

```
      int primulElem; /*cursor la primul element în
    } multime;          tabloul nume*/

typedef struct tip_element
    {
      int numeSubmultime;  /*numele submulţimii cărei îi
                             aparţine elementul*/
      int urm; /*cursor la elementul următor în tabloul
                 nume*/
    } element;                                     [9.6.2.a]

typedef struct MultimeDeSubmultimi
    {
       tip_multime inceputuri[N]; /*tablou conţinând
       începuturile listelor corespunzătoare submulţimilor*/
       tip_element nume[N]; /*tablou care precizează pentru
         fiecare element submulţimea căreia îi aparţine;
         elementele aparţinând aceleiaşi submulţimi sunt
         înlănţuite*/
    } multime_de_submultimi;
------------------------------------------------------------
{Mulţimi pe care sunt definiţi operatorii Uniune şi Cauta
 Implementare bazată pe liste înlănţuite implementate cu
 ajutorul cursorilor – structuri de date - varianta PASCAL}

type TipNumeSubmultime = 1..n;
     TipElementSubmultime = 1..n;

     MultimeDeSubmultimi = record {mulţime de submulţimi}
       inceputuri: array[1..n] OF
         {tablou conţinând începuturile listelor
          corespunzătoare submulţimilor}
         record {început de listă}
           contor: 0..n; {număr curent de elemente}
           primulElem: 0..n {cursor în tabloul nume}
         end;
       nume: array[1..n] OF                        [9.6.2.a]
         {tablou care precizează pentru fiecare element
          submulţimea căreia îi aparţine; elementele
          aparţinând aceleiaşi submulţimi sunt
          înlănţuite}
         record {componenta tablou}
           numeSubmultime: TipNumeSubmultime;
           urm: 0..n  {cursor în tabloul Nume}
         end
     end; {MultimeDeSubmultimi}
------------------------------------------------------------
```

- În figura 9.6.2.a. apare prezentat un exemplu de structură de date de tip **mulţime de submulţimi** unde submulţimea `1` este `{1,3,4}`, submulţimea `2` este `{2}` iar submulţimea `4` este `{5,6}`, exemplu bazat pe modelul de structură de date propus.

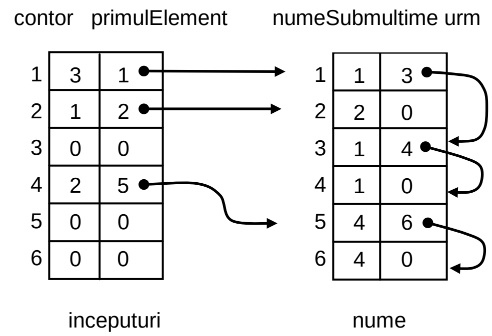

Fig.9.6.2.a. Implementarea structurii mulţime de submulţimi cu ajutorul listelor înlănţuite bazate pe cursori

* Procedurile ***Initializare, Uniune*** şi ***Cauta*** apar în secvenţa [9.6.2.b].

```
------------------------------------------------------------
/*Mulţimi pe care sunt definiţi operatorii Uniune şi Cauta
  Implementarea operatorilor Initializare, Uniune şi Cauta
  - varianta C*/

typedef int Tip_Nume_Submultime;
typedef int Tip_Element_Submultime;

void Initializare(Tip_Nume_Submultime A,
     Tip_Element_Submultime x, MultimeDeSubmultimi * M)
/*iniţializează pe A ca o submulţime care-l conţine numai pe
  x*/
{
    (*M).nume[x].numeSubmultime=A;
    (*M).nume[x].urm=0;  /*sfârşitul listei elementelor
                              lui A*/
    (*M).inceputuri[A].contor=1;
    (*M).inceputuri[A].primulElem=x;
 } /*Initializeaza*/

void Uniune (Tip_Nume_Submultime A,
     Tip_Nume_Submultime B, MultimeDeSubmultimi * M)

/*realizează uniunea lui A şi B denumind submulţimea
        rezultată în mod arbitrar A sau B*/
 {
   int i; /*utilizat în găsirea sfârşitului celei mai
                scurte liste*/

   if ((*M).inceputuri[A].contor>
                  (*M).inceputuri[B].contor)
      { /*A este cea mai mare mulţime, deci se adaugă
             B la A*/
        i=(*M).inceputuri[B].primulElem;
        do /*se parcurge mulţimea B modificând pentru
                fiecare element al său, numele submulţimii
                din B în A*/
          {
```

```
            (*M).nume[i].numeSubmultime=A;
            i=(*M).nume[i].urm;             /*[9.6.2.b]*/
          } while ((*M).nume[i].urm==0);
        /*înlănţuie lista A la sfârşitul lui B şi modifică
           numele listei B în A. Se şterge lista B*/
        (*M).nume[i].numeSubmultime=A; /*i indică ultimul
                                        element al lui B*/
        (*M).nume[i].urm=(*M).inceputuri[A].primulElem;
        (*M).inceputuri[A].primulElem=
                (*M).inceputuri[B].primulElem;
        (*M).inceputuri[A].contor=(*M).inceputuri[A].contor
                           +(*M).inceputuri[B].contor;
        (*M).inceputuri[B].contor=0;
        (*M).inceputuri[B].primulElem=0 /*sterge mulţimea
                                          B*/
      } /*if*/
    else /*B este cel puţin tot aşa de mare ca A, se
          adaugă A la B*/
      {
        *secvenţă similară cu cea de mai sus,
           interschimbând însă pe B cu A;
      } /*else*/
  } /*Uniune*/

Tip_Nume_Submultime Cauta(Tip_Element_Submultime x,
                MultimeDeSubmultimi * M)
  /*returneaza numele submulţimii căreia îi aparţine x*/
  {
    return((*M).nume[x].numeSubmultime)
  } /*Cauta*/
------------------------------------------------------------
```

- Referitor la implementarea operatorului ***Uniune*** se fac următoarele precizări.

  - Iniţial se verifică care dintre mulţimi conţine mai puţine elemente şi se adaugă această submulţime celeilalte.

  - Prima parte a algoritmului tratează situaţia în care B are mai puţine elemente ca şi A iar partea a doua situaţia inversă.

    - În cazul în care A are mai multe elemente decât B, tuturor elementelor mulţimii B li se schimbă numele în tabloul `nume`, în A (bucla **`do - while`**).

    - Se înlănţuie lista A la sfârşitul listei B modificate.

    - Se modifică începutul listei A astfel încât să indice începutul listei B modificate, iar contorul corespunzător să indice suma contoarelor celor două liste.

    - În final se şterge lista B din tabloul `inceputuri`.

- Cazul următor se tratează identic cu deosebirea că se interschimbă A cu B.

9.6.3. Implementare bazată pe arbori a mulţimii pe care sunt definiţi operatorii UNIUNE şi CAUTĂ

- O altă metodă de implementare a mulţimilor peste care sunt definiţi operatorii ***Uniune*** şi ***Cauta*** se bazează pe **structura arbore** implementată în varianta **indicator spre părinte** (vezi &8.4.1.4).

- Descrierea acestei metode se va realiza numai la **nivel de principiu**.

    - Astfel se presupune că nodurile structurii arbore conţin elementele unei mulţimi sau eventual referiri la aceste elemente prin intermediul unei asocieri.

    - Fiecare nod, cu excepţia rădăcinii conţine un **indicator spre părintele** său.

    - Rădăcina păstrează **numele mulţimii**.

- Asocierea dintre numele mulţimilor şi rădăcinile arborilor care memorează mulţimile, permite accesul simplu la o anumită mulţime în cazul realizării operaţiei ***Uniune***.

    - În fig. 9.6.3.a sunt prezentate mulţimile `A={1,2,3,4,}`, `B={5,6}` şi `C={7}` reprezentate în acest mod.

    - Dreptunghiurile care conţin numele mulţimilor nu sunt considerate noduri separate, ci se presupune că fac parte din structura nodului rădăcină.

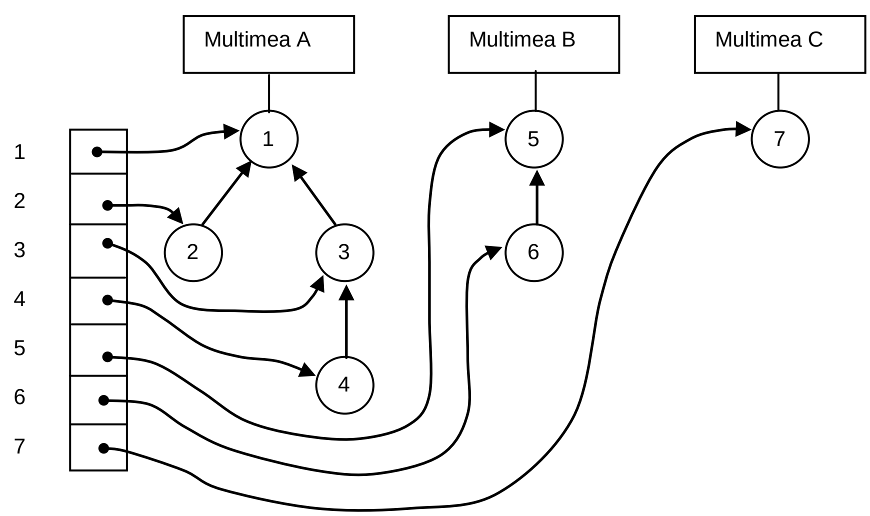

Fig.9.6.3.a. Mulţime de submulţimi reprezentată ca şi o colecţie de arbori

- Pentru a afla **numele mulţimii** care conţine un anumit element `x`:

- (1) Pe baza asocierii, se determină nodul din arbore care îl conţine pe x.

- (2) Urmând drumul de la nod spre rădăcină se află numele mulţimii care conţine elementul.

- Pentru a realiza **uniunea a două mulţimi** este suficient că rădăcina arborelui corespunzător uneia dintre mulţimi să devină fiul rădăcinii arborelui corespunzător celeilalte.

  - Astfel unind mulţimea A şi B din fig. 9.6.3.a se obţine mulţimea din fig. 9.6.3.b.

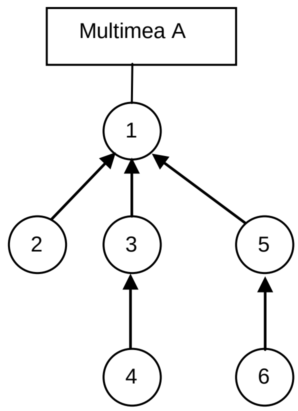

Fig.9.6.3.b. Uniunea mulţimilor A şi B

- În cazul cel mai **defavorabil** rezultă un arbore degenerat într-o listă liniară cu n noduri.

  - În această situaţie se ajunge executând în mod repetat operaţia ***Uniune*** între o mulţime conţinând un singur element şi o altă submulţime generată în aceeaşi manieră.

  - Execuţia operaţiei ***Cauta*** asupra tuturor elementelor necesită în acest caz O(n^2) unităţi de timp.

  - Întrucât operaţia ***Uniune*** se execută în O*(1)* unităţi de timp, performanţa de ansamblu a acestei implementări depinde direct de numărul de operaţii de căutare efectuate.

- O cale de **creştere a acestei performanţe** este aceea de a face cât mai eficientă operaţia de căutare.

  - În acest scop fiecărei rădăcini i se adaugă un **câmp suplimentar** care precizează **numărul de elemente ale mulţimii**.

  - Când se realizează uniunea a două mulţimi se va proceda astfel încât rădăcina arborelui cu număr mai mic de elemente să devină fiul rădăcinii aborelui cu mai multe elemente.

* Astfel de fiecare dată când un nod este mutat printr-o operaţie ***Uniune*** într-un nou arbore se întâmplă două lucruri:

  (1) Distanţa de la nod la rădăcină creşte cu o unitate

  (2) Un nod al mulţimii cu mai puţine elemente devine membru al unei mulţimi care este cel puţin de două ori mai mare decât cea din care provine.

* Prin urmare, dacă numărul total de elemente este `n`, nici un nod nu poate fi mutat de mai mult de $\log_2 n$ ori.

  * Rezultă că distanţa de la un nod al structurii la rădăcină nu poate depăşi, niciodată $\log_2 n$.

  * În consecinţă, performanţa operaţiei de căutare a unui nod devine O$(log_2 n)$, iar performanţa totală corespunzătoare căutării tuturor nodurilor O$(n \log_2 n)$.

9.6.3.1 Comprimarea drumului.

* O altă idee care poate contribui la creşterea performanţei implementării mulţimilor pe care sunt definiţi operatorii ***Uniune*** şi ***Cauta*** este **comprimarea drumului** ("**path compression**") [AH85].

  * Conform acestei metode în timpul execuţiei operaţiei de căutare, când se parcurge drumul de la nod spre rădăcină, fiecare nod întâlnit de-a lungul acestui parcurs se face fiu al rădăcinii.

  * Acest lucru se poate realiza în două treceri:

    * (1) La prima trecere se află rădăcina.

    * (2) La a doua trecere se reparcurge drumul făcând din fiecare nod un fiu al rădăcinii.

  * În figura 9.6.3.1.a (b) se prezintă structura modificată a arborelui (a), obţinută în urma execuţiei operaţiei ***Cauta*** pentru elementul `9`.

  * După cum se observă nodurile `1` şi `2` nu sunt afectate deoarece nodul `1` este chiar rădăcina iar `2` este fiul acesteia.

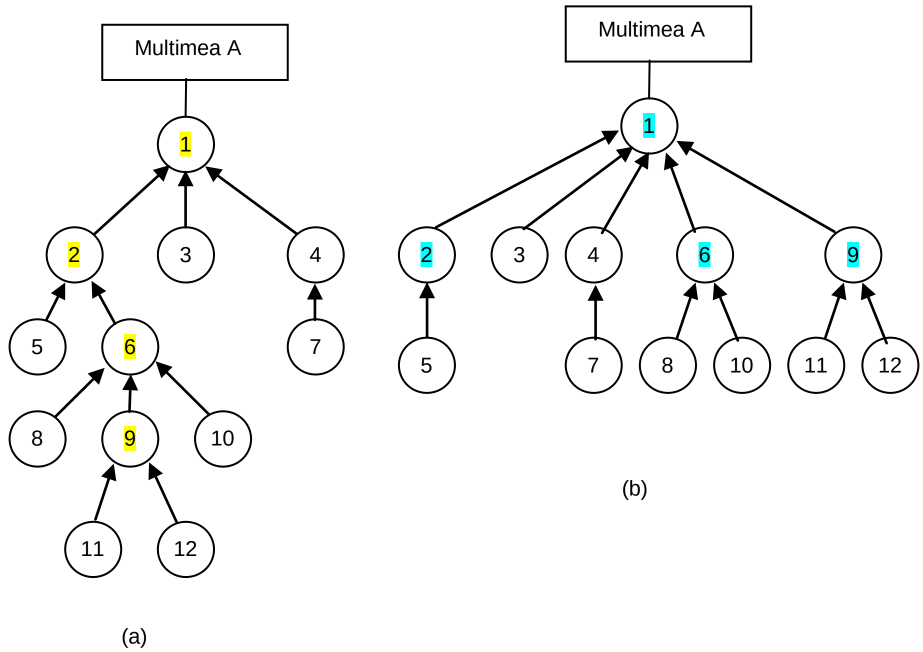

Fig.9.6.3.1.a. Exemplu de comprimare a drumului

- Comprimarea drumului **nu** afectează performanţa operatorului ***Uniune*** în schimb îmbunătăţeşte progresiv performanţa operatorului ***Cauta***, prin scurtarea drumului parcurs, cu un efort relativ redus.

- Analiza performanţei medii a operaţiei de căutare, atunci când se utilizează comprimarea drumului este o operaţie foarte dificilă.

  - În cazul în care arborele este degenerat în lista liniară ***Cauta*** poate necesita O(n) unităţi de timp, însă comprimarea drumului poate modifica în timp structura arborelui astfel încât căutarea tuturor elementelor mulţimii, respectiv toate cele n operaţii ***Cauta*** să necesite în total un efort de calcul de ordinul O(n).

- Algoritmul care utilizează atât comprimarea drumului, cât şi adăugarea arborelui mai mic celui mai mare, este în mod asimptotic, **cea mai eficientă metodă** cunoscută de implementare a mulţimilor peste care sunt definite operaţiile ***Uniune*** şi ***Cauta***.

9.7. Mulţimi pe care sunt definiţi operatorii UNIUNE, CAUTĂ şi PARTIŢIONARE

- Fie M o mulţime ale cărei elemente sunt ordonate de o relaţie "<".

- Peste această mulţime se consideră definiţi operatorii ***Uniune*** şi ***Cauta*** precizaţi anterior precum şi operatorul ***Partitionare***.

- Operatorul ***Partitionare*** (M,M_1,M_2,x) îl divide pe M în două mulţimi $M_1=\{e|e\in M$ **şi** $e<x\}$ respectiv $M_2=\{e|e\in M$ **şi** $e\geq x\}$.

  - Valoarea lui M după diviziune este **nedefinită**, mai puţin în cazul în care ea este una din mulţimile M_1 sau M_2.

- Cu alte cuvinte operatorul ***Partitionare*** separă mulţimea M în două submulţimi:

    - (1) Prima mulţime M_1 conţine toate elementele lui M **mai mici** decât elementul x furnizat ca parametru

    - (2) A doua mulţime M_2 conţine toate elementele lui M cu valori **mai mari sau egale** cu x.

9.7.1. Problema celei mai lungi subsecvenţe comune (LCS)

- În cele mai multe situaţii **operaţia de partiţionare a unei mulţimi** rezidă în compararea fiecărui element al mulţimii cu o valoare fixă x.

    - În continuare se va aborda o astfel de problemă.

- Se numeşte **subsecvenţa** a unei secvenţe x, un şir format din 0 sau mai multe elemente, **nu neapărat contigue**, extras din x.

    - Fiind date două secvenţe x şi y se numeşte **cea mai lungă subsecvenţă comună** a celor două secvenţe (**"longest common subsequence" – LCS**), cea mai lungă secvenţă care este în acelaşi timp subsecvenţa atât pentru x cât şi pentru y.

- Spre exemplu, un LCS al secvenţelor a,b,c,b,d,a,b şi b,d,c,a,b,a este subsecvenţa b,c,b,a obţinută după cum rezultă din figura 9.7.1.a.

    - Mai există şi alte LCS-uri, de lungime 4, spre exemplu b,d,a,b dar nu există nici o subsecvenţă comună de lungime 5.

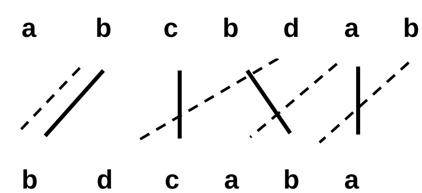

Fig.9.7.1.a. Exemplu de cea mai lungă subsecvenţă comună.

- În **UNIX** există comanda DIFF care compară fişierele linie cu linie şi află cea mai lungă subsecvenţă comună, unde o **linie a fişierului** este considerată drept un element al subsecvenţei.

    - Premiza de la care porneşte comanda DIFF este următoarea: se presupune că liniile care nu apar în LCS sunt linii inserate, şterese sau modificate care diferenţiază cele două fişiere.

- Dacă spre exemplu cele două fişiere sunt două versiuni ale unui acelaşi program, DIFF va găsi cu mare probabilitate modificările efectuate.

- Există mai multe soluţii cu caracter de generalitate care rezolvă **problema LCS** de regulă în $O(n^2)$ paşi pentru o subsecvenţă de lungime n.

- Comanda DIFF utilizează o **strategie** diferită care este foarte eficientă când fişierele **nu** au prea multe repetări ale aceleeaşi linii.

- Algoritmul utilizat de DIFF face uz de o implementare eficientă a mulţimilor pe care sunt definiţi operatorii ***Uniune***, ***Cauta*** şi ***Partitionare*** şi se execută în $O(p \log_2 n)$ unităţi de timp unde:

  - n este numărul maxim de linii ale fişierului.

  - p este numărul de perechi de poziţii, câte una din fiecare fişier, care conţin aceeaşi linie.

- Spre exemplu p în figura 9.7.1.a are valoarea 4+6+1+1=12 respectiv:

  - Cei doi de a din fiecare şir contribuie cu 2x2=4 perechi

  - b contribuie cu 3x2=6 perechi

  - c şi d cu câte 1x1=1 pereche.

- În cel mai rău caz p poate fi n^2 iar algoritmul va consuma $O(n^2 \log_2 n)$ unităţi de timp.

  - Acest lucru se întâmplă atunci când A este identic cu B şi toate elementele sale sunt identice.

  - În practică însă p este apropiat de n rezultând o eficienţă de ordinul $O(n \log_2 n)$.

9.7.2. Determinarea LCS utilizând mulţimi pe care sunt definiţi operatorii UNIUNE, CAUTĂ şi PARTIŢIONARE

- Fie $A=a_1a_2...a_n$ şi $B=b_1b_2...b_m$ două secvenţe pentru care se doreşte găsirea secvenţei LCS.

- Algoritmul pentru determinarea LCS constă în **trei** paşi.

- (1) **Pasul 1.** Pentru fiecare simbol a aparţinând secvenţei A, se identifică poziţia sa în cadrul secvenţei.

  - În acest scop se defineşte structura $Poziţii(a)=\{i \mid a_i=a\}$.

- Se face precizarea că această corespondenţă **nu** este biunivocă, un acelaşi element `a` poate să apară pe mai multe poziţii în cadrul secvenţei `A`.

- După cum se observă, `Poziţii` constă din mai multe **mulţimi de indici**, câte una pentru fiecare element distinct al secvenţei A.

- Aceste mulţimi conţin poziţiile în cadrul secvenţei ale respectivului element exprimate ca indici.

- Înregistrarea mulţimii poziţiilor se poate realiza în două moduri:

    a) Prin intermediul unei **asocieri** între simboluri şi începuturile listelor care conţin poziţiile;

    b) Prin intermediul unei **tabele de dispersie** deschise.

- În ambele situaţii `Pozitii` se poate determina în medie cu un efort de calcul de ordinul O*(n)* **paşi**, unde prin "pas" se înţelege timpul necesar prelucrării unui simbol (prin dispersie, asociere sau prin comparaţie cu un altul).

    - Acest timp poate fi o **constantă** dacă simbolurile sunt caractere sau întregi dar tot atât de bine în situaţia în care simbolurile lui A şi B sunt **linii de text**, atunci timpul necesar prelucrării unui simbol depinde de lungimea medie a liniilor textului.

- (2) **Pasul 2**. După ce s-au determinat mulţimile de indici ale structurii `Poziţii`, pe rând, pentru fiecare simbol (element) care apare în secvenţa A, se poate trece la determinarea LCS în raport cu secvenţa B.

- În cele ce urmează se va preciza modul în care se găseşte **lungimea lui LCS**, aflarea secvenţei propriu-zise fiind recomandată ca exerciţiu.

    - Se consideră secvenţa curentă A ca fiind delimitată de elementele $a_1...a_i$.

    - Pasul 2 al algoritmului ia în considerare secvenţe $b_1...b_j$ ale secvenţei B pentru $j=1,2...,m$.

    - Considerând o secvenţă $b_1...b_j$, este necesar să fie determinată pentru fiecare indice i cuprins între 0 şi n, LCS-ul pentru secvenţele $a_1...a_i$ şi $b_1...b_j$.

    - Valorile indicilor i procesaţi se grupează în mulţimile M_k, pentru $k=0,1,2,...,n$, unde o mulţime M_k constă din toate valorile indicelui i pentru care LCS-ul secvenţelor $a_1...a_i$ şi $b_1...b_j$ are lungimea k.

        - Astfel o mulţime M_k va fi formată întotdeauna dintr-un set de indici întregi **consecutivi**, iar indicii mulţimii M_{k+1} sunt **mai mari** decât cei ai lui M_k pentru toţi k.

- Spre **exemplu**, considerând secvenţele din figura 9.7.1.a, în figura 9.7.2.a se prezintă modul de determinare al mulţimilor M_k pentru j=5.

  - **Ideea de bază** este aceea conform căreia, se iau pe rând elementele secvenţei A, începând cu secvenţa vidă şi se adaugă pe rând elementul următor al secvenţei A.

  - În fiecare astfel de pas se determină numărul de coincidenţe dintre secvenţa A şi secvenţa B cu j=5.

  - Pentru început se încearcă găsirea coincidenţelor între 0 elemente ale secvenţei A (i=0) şi cele 5 elementele b,d,c,a,b ale secvenţei B.

    - Se găseşte evident un LCS de lungime 0, ca atare indicele 0 este adăugat mulţimii M_0 (fig.9.7.2.a.(a)).

  - Considerând în continuare pe i=1, deci primul element al secvenţei A şi cele 5 elemente ale secvenţei B se găseşte un LCS de lungime 1 (fig.9.7.2.a.(b)), deci indicele 1 aparţine mulţimii M_1.

  - Trecând la elementul următor al lui A şi luând în calcul primele două elemente ale lui A, în acelaşi context se găseşte un LCS de lungime 2 (fig.9.7.2.a.(c)).

  - Procedând în acelaşi mod se obţin în final următoarele mulţimi de indici: M_0={0}, M_1={1}, M_2={2,3}, M_3={4,5,6}, M_4={7} (fig.9.7.2.a. (a)… (h)).

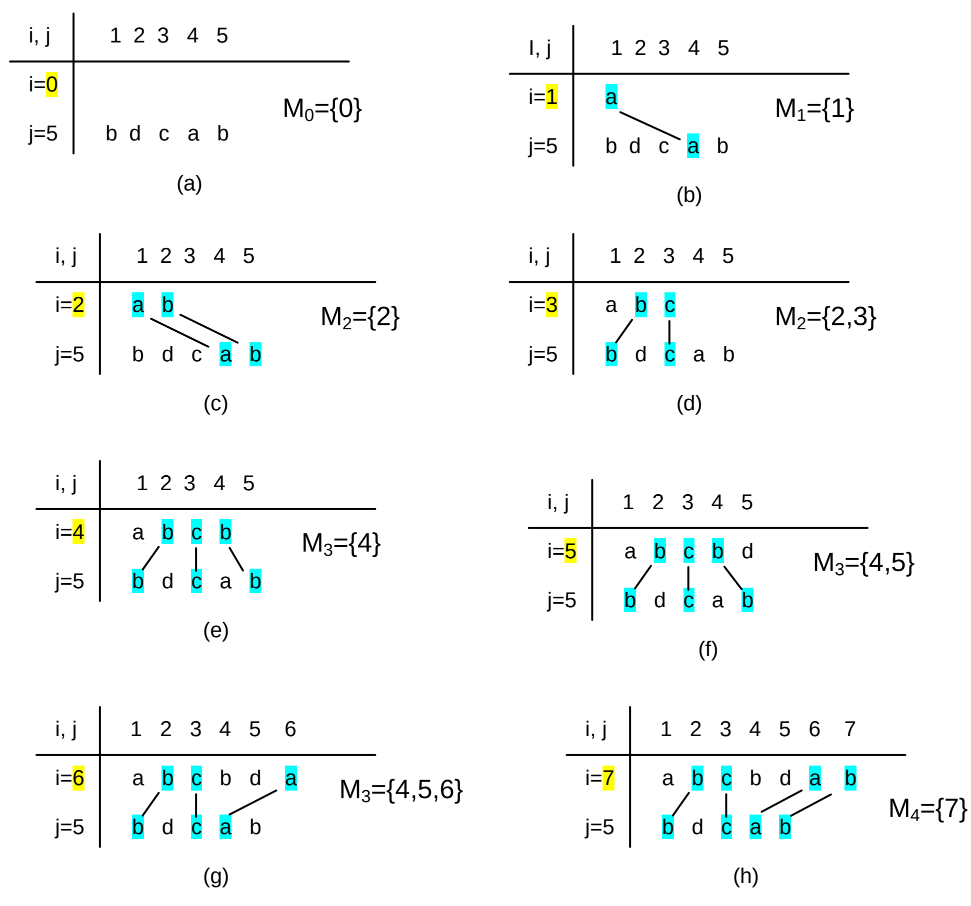

Fig.9.7.2.a. Exemplu de determinare a mulţimilor M_k

- (3) În **pasul 3** al algoritmului:

  - Se presupune că s-au determinat mulţimile M_k pentru poziţia `j-1` a celei de-a doua secvenţe, cu alte cuvinte pentru o secvenţă `B` de lungime `j-1`.

  - Se investighează în continuare maniera în care se modifică conţinutul mulţimilor M_k atunci când **se adaugă** secvenţei `B` elementul situat pe poziţia următoare `j`.

  - În acest scop se consideră `Pozitii(`b_j`),` adică poziţiile în care elementul adăugat b_j apare în secvenţa `A`.

  - Pentru fiecare indice `r` aparţinând lui `Pozitii(`b_j`)` se verifică dacă se poate îmbunătăţi lungimea vreunui LCS existent, adăugând potrivirea dintre a_r şi b_j LCS-ului deja determinat pentru secvenţele $a_1...a_{r-1}$ şi $b_1...b_{j-1}$.

  - Dacă atât `r-1` (`r`-ul anterior) cât şi `r` (`r`-ul curent) aparţin unei aceleiaşi mulţimi M_k, atunci toate elementele $s \geq r$ din M_k, aparţin de fapt mulţimii M_{k+1} pentru b_j considerat.

    - Acest lucru este valabil pe baza observaţiei că celor `k` potriviri existente între elementele secvenţelor $a_1...a_{r-1}$ şi $b_1...b_{j-1}$ li se mai adaugă o potrivire şi anume cea dintre a_r şi b_j.

- **În consecinţă**, trecând de la elementul b_{j-1} la elementul b_j, adică adăugând secvenţei `B` elementul următor, mulţimile M_k şi M_{k+1} se pot modifica pe baza următorului algoritm:

  (1) Se determină mulţimea de indicii `r` aparţinând lui `Pozitii(`b_j`)`. Pentru fiecare indice `r`:

  (2) Se execută operatorul ***Cauta***`(r)` pentru a determina mulţimea M_k căreia îi apartine `r`.

  (3) Se execută operatorul ***Cauta***`(r-1)`.

  - Dacă ***Cauta***`(r-1)`nu este mulţimea M_k, potrivirea dintre b_j şi a_r nu conduce la nici un beneficiu. În consecinţă se sar paşii 4 şi 5 ai algoritmului, iar M_{k+1} nu se modifică.

  (4) Dacă ***Cauta***`(r-1)`este M_k, se execută operatorul ***Partitionare***`(`M_k`,`M_k`,`M'_k`,r)` pentru a determina acele elemente ale lui M_k care sunt mai mari sau egale cu `r` şi pentru a construi cu ele mulţimea M'_k.

  (5) Se execută operatorul ***Uniune***`(`M'_k`,`M_{k+1}`,`M_{k+1}`)`prin care se mută elementele din mulţime M'_k în mulţimea M_{k+1}.

(6) Ciclul se reia de la (2) pentru fiecare valoare a lui `r` din mulţimea `Poziţii(bj)`.

- **OBS:** Se subliniază faptul că indicii `r` ai lui `Pozitii(bj)` trebuiesc procesaţi începând cu **cea mai mare valoare**, în **ordine descrescătoare**.

    - Pentru a motiva acest lucru se consideră spre exemplu situaţia în care elementele `7` şi `9` aparţin mulţimii `Pozitii(bj)` şi înaintea extinderii secvenţei `B` cu elementul b_j, mulţimea M_3=`{6,7,8,9}` şi mulţimea M_4=`{10,11}`.

    - Dacă se tratează elementele `7` şi `9` în această ordine, (Cazul 1, fig.9.7.2.b.(a)) procesând mai întîi elementul `7`, se partiţionează M_3 în M_3=`{6}` şi M'_3=`{7,8,9}` ceea ce conduce la M_4=`{7,8,9,10,11}`.

    - În continuare procesând elementul `9`, M_4 se partiţionează în M_4=`{7,8}`şi M'_4=`{9,10,11}`, după care `9,10` şi `11` se unesc în M_5.

    - Astfel în **mod paradoxal** indicele `9` a fost mutat din M_3 în M_5, adăugând un singur element celei de-a doua secvenţe, lucru care este imposibil.

    - Eroarea provine din faptul că s-au considerat în mod eronat potrivirile dintre b_j şi a_7 respectiv dintre b_j şi a_9 în această ordine, creându-se astfel un LCS imaginar de lungime `5`.

    - Pentru a evita această situaţie verificările trebuiesc realizate considerând întîi potrivirea b_j şi a_9 şi apoi potrivirea b_j şi a_7 (Cazul 2 fig.9.7.2.b.(b)).

M_3 = {6, 7, 8, 9}; M_4 = {10, 11};
Poziţii (b_j) = {7, 9};

Cazul 1. Se prelucrează întâi 7 apoi 9

I. Cauta (7) = M_3
Cauta (6) = M_3

II. Partitionare (M_3, M_3, M'_3, 7)
⇨ M_3 = {6}; M'_3 = {7, 8, 9};

III. Uniune (M'_3, M_4, M_4)
⇨ M_4 = {7, 8, 9, 10, 11};
r-1 r

IV. Cauta (9) = M_4
Cauta (8) = M_4

V. Partitionare (M_4, M_4, M'_4, 9)
⇨ M_4 = {7, 8}; M'_4= {9,10,11};

VI. Uniune (M'_4, M_5, M_5)
⇨ M_5 = {9, 10, 11};
absurd

(a)

M_3 = {6, 7, 8, 9}; M_4 = {10, 11};
Poziţii (b_j) = {7, 9};

Cazul 2. Se prelucrează întâi 9 apoi 7

I. Cauta (9) = M_3
Cauta (8) = M_3

II. Partitionare (M_3, M_3, M'_3, 9)
⇨ M_3 = {6, 7, 8}; M'_3 = {9};

III. Uniune (M'_3, M_4, M_4)
⇨ M_4 = {9, 10, 11}

IV. Cauta (7) = M_3
Cauta (6) = M_3

V. Partitionare (M_3, M_3, M'_3, 7)
⇨ M_3 = {6}; M'_3= {7, 8};

VI. Uniune (M'_3, M_4, M_4)
⇨ M_4 = {7, 8, 9, 10, 11};
corect

(b)

Fig.9.7.2.b. Exemple de execuţie incorectă (a), respectiv corectă (b) a algoritmului propus

- În secvenţa [9.7.2.a] apare o schiţă a algoritmulul care calculează şi actualizează mulţimile M_k în timpul parcurgerii celei de-a doua secvenţe.

  - Se presupune că structura `Pozitii` corespunzătoare tuturor elementelor secvenţei `A` a fost construită în prealabil.

- Pentru a determina lungimea LCS, este suficient ca la sfârşitul prelucrării să se execute o operaţie ***Cauta***`(n)`, `n` fiind indicele ultimului caracter al secvenţei `A`.

- După cum s-a mai precizat, determinarea efectivă a LCS-ului este sugerată ca exerciţiu.

```
/*Determinarea lungimii LCS*/
                                              /*[9.7.2.a]*/
*iniţializează M0={0,1,2,...,n};
*iniţializează Mi pe multimea vidă [i=1,n];
pentru (j=1 la m)  /*determină Mk pentru poziţia j*/
  pentru (r in Pozitii(bj), începând cu cea mai mare
             valoare)
    | k=Cauta(r);
    | daca (k==Cauta(r-1))
    |    | /*r nu este cel mai mic din Mk*/
    |    | Partitionare(Mk,Mk,Mk',r);
    |    | Uniune(Mk',Mk+1,Mk+1);
    |    | □ /*daca*/
    | □ /*pentru*/
/*LCS*/
```

9.7.3. Analiza performanţei algoritmului LCS

- După cum s-a precizat, algoritmul din secvenţa [9.7.2.a] este performant şi util dacă **nu** există prea multe potriviri între cele două secvenţe.

- Măsura **numărului de potriviri** este `p` cu menţiunea că structura `Pozitii` este calculată pentru secvenţa `A` ([9.7.3.a]).

$$p = \sum_{j=1}^{m} card(Pozitii(b_j)) \qquad [9.7.3.a]$$

- În formula [9.7.3.a] card(POZITII(b_j)) este cardinalitatea mulţimii Pozitii(b_j) iar m este lungimea secvenţei B.

  - Cu alte cuvinte p este **suma numărului poziţiilor** din secvenţa A care se potrivesc cu b_j, pentru toţi j=1,m.

  - Este de aşteptat ca valoarea medie a lui p să fie de acelaşi ordin cu m şi n care sunt lungimile celor două secvenţe (fişiere).

- O structură de date eficientă pentru implementarea mulţimilor M_k este **arborele 2-3** (&8.9.4).

  - Utilizând o astfel de structură, **iniţializarea** mulţimilor M_k (secvenţa [9.7.2.a], liniile [1] şi [2]) necesită O*(n)* paşi.

  - Operaţia ***Cauta*** necesită un **tablou suplimentar** care păstrează **asocierea** dintre elemente şi nodurile terminale corespunzătoare lor, precum şi completarea structurii arbore 2-3 cu **pointeri** de tipul "indicator spre părinte".

    - **Numele mulţimii** (k pentru M_k) poate fi păstrat în rădăcină, astfel încât operaţia ***Cauta*** se execută urmând drumul de la nod spre rădăcină în $O(log_2 n)$ paşi.

    - În consecinţă, toate execuţiile liniilor [5] şi [6] din cadrul algoritmului necesită un efort de ordinul $O(p \log_2 n)$, deoarece fiecare linie este executată exact odată pentru fiecare potrivire.

- Operatorul ***Uniune*** din linia [8] are proprietatea specială că fiecare membru al lui M'_k este mai mic decât orice membru al lui M_{k+1}, proprietate avantajoasă pentru implementarea bazată pe arbori 2-3.

  - Operaţia ***Uniune*** începe prin plasarea arborelui corespunzător mulţimii M'_k la stânga celui corespunzător mulţimii M_{k+1}. Pot apare trei situaţii:

    (1) Dacă ambii arbori sunt de aceeaşi înălţime, se creează o nouă rădăcină care are drept fii rădăcinile celor doi arbori.

    (2) Dacă M'_k este mai scurt, se inserează rădăcina acestui arbore ca şi cel mai din stânga fiu al celui mai din stânga nod al lui M_{k+1} situat la nivelul corespunzător. Dacă acest nod are patru fii se procedează la restructurare ca şi în cazul inserţiei în arbori 2-3 (fig.9.7.3.a (a), (b)).

    (3) Dacă M_{k+1} este mai scurt, rădăcina lui se face cel mai din dreapta fiu al celui mai din dreapta nod al lui M'_k situat la nivelul corespunzător după care, dacă este necesar se restructurează şi se redenumeşte arborele.

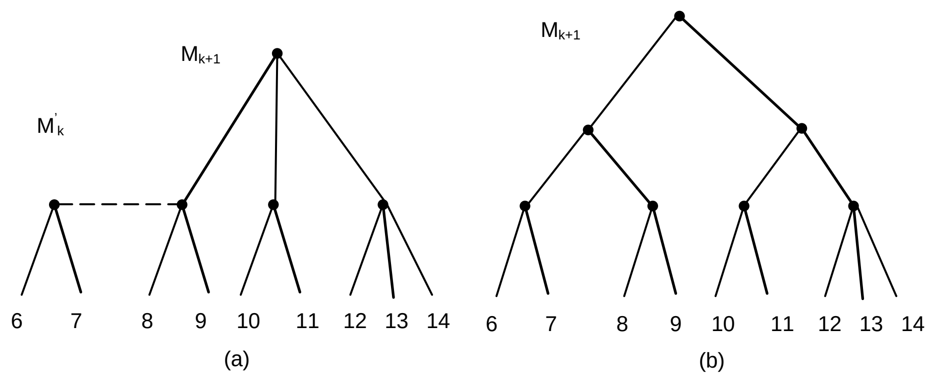

Fig.9.7.3.a. Exemplu de execuţie a operatorului ***Uniune***

- Operatorul ***Partitionare*** pentru `r`, (linia [7]) necesită parcurgerea drumului de la nodul terminal `r` spre rădăcină, duplicând fiecare nod interior situat de-a lungul drumului şi ataşând câte o copie a fiecărui nod celor doi arbori rezultaţi.

    - Nodurile fără urmaşi sunt eliminate.

    - Nodurile cu un singur fiu sunt suprimate, nodul fiu fiind inserat după caz în arborele respectiv la nivelul corespunzător.

    - Dacă este necesar se procedează la restructurarea arborelui.

    - Această situaţie se poate urmări în figura 9.7.3.b, unde s-a realizat partiţionarea în raport cu valoarea `9` a arborelui din figura 9.7.3.a (b).

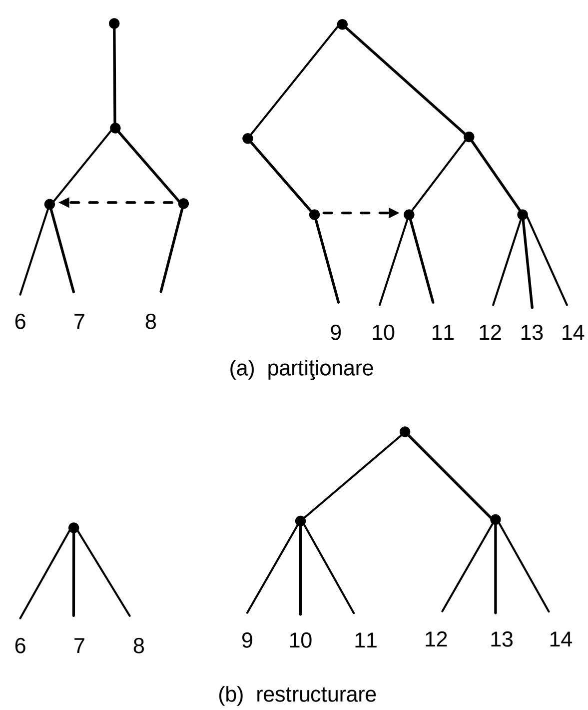

Fig.9.7.3.b. Exemplu de execuţie a operatorului ***Partitionare***

- Analizând un număr mare de cazuri, s-a observat că partiţionarea şi reorganizarea arborilor 2-3 de "jos în sus" necesită un efort de calcul de ordinul O$(\log_2 n)$.

    - Astfel, timpul total necesar execuţiei liniilor [7] şi [8] din secvenţa [9.7.2.a] este proporţional cu O$(p \log_2 n)$ şi în consecinţă întregul algoritm necesită în medie O$(p \log_2 n)$ paşi.

- Mai trebuie adăugat timpul necesar calculului operatorului `Pozitii(a)`.

    - După cum s-a mai precizat, dacă elementele lui `a` sunt de mari dimensiuni, acest calcul poate dura mai mult decât oricare altă parte a algoritmului.

    - Dacă elementele pot fi comparate într-un singur pas, atunci sortarea şirului $a_1, a_2, ..., a_n$, de fapt sortarea obiectelor (i, a_i) după câmpul a_i , se poate realiza cu un efort proporţional cu O$(n \log_2 n)$, după care `Pozitii(a)` poate fi completat pornind de la lista sortată, în O(n) paşi.

- Astfel lungimea secvenţei LCS poate fi calculată cu un efort de ordinul O$((\max (n,p)) \log_2 n)$ şi întrucât de regulă $p \geq n$, rezultă O$(p \log_2 n)$.