

## 6. Liste

### 6.1. Structura de date listă

- În cadrul structurilor de date avansate, **structura listă** ocupă un loc important.
- O **listă**, este o **structură dinamică**, care se definește pornind de la noțiunea de **vector**.
  - **Elementele** unei liste sunt toate de **același** tip, ca atare o listă este o **structură de date omogenă**.
  - **Toate elementele** unei liste sunt înregistrate **în memoria centrală** a sistemului de calcul.
- Spre deosebire de **structura statică tablou** la care se impune ca numărul componentelor să fie **constant**, în cazul **listelor** acest număr poate fi **variabil**, chiar **nul**.
  - **Listele** sunt structuri **flexibile** particulare, care funcție de necesități pot **crește** sau **descrește** și ale căror **elemente** pot fi **referite, inserate** sau **șterse** în orice poziție din cadrul listei.
  - Două sau mai multe **liste** pot fi **concatenate** sau **scindate** în subliste.
- În practica programării listele apar în mod obișnuit în aplicații referitoare la regăsirea informației, implementarea translațiilor de programe, simulare etc.

### 6.2. TDA Listă

- Din punct de vedere **matematic**, o **listă** este o **secvență** de zero sau mai multe **elemente** numite **noduri** aparținând unui anumit **tip** numit **tip de bază**.
- Formal, o listă se reprezintă de regulă ca și în [6.2.a]:

$$\text{-----}$$
$$a_1, a_2, \dots, a_n \qquad \qquad \qquad [6.2.a]$$
$$\text{-----}$$

- unde  $n \geq 0$  și fiecare  $a_i$  aparține **tipului de bază**.
- Numărul  $n$  al nodurilor se numește **lungimea listei**.

- Presupunând că  $n \geq 1$ , se spune că  $a_1$  este **primul** nod al **listei** iar  $a_n$  este **ultimul nod**.
  - Dacă  $n = 0$  avem de-a face cu o **listă vidă**.
- O proprietate importantă a unei **liste** este aceea că nodurile sale pot fi considerate **ordonate liniar** funcție de **poziția** lor în cadrul listei.
  - De regulă, se spune că nodul  $a_i$  se află pe poziția  $i$ .
  - Se spune că  $a_i$  **precede** pe  $a_{i+1}$  pentru  $i = 1, 2, \dots, n-1$ .
  - Se spune că  $a_i$  **succede** (urmează) lui  $a_{i-1}$  pentru  $i = 2, 3, 4, \dots, n$ .
- Este de asemenea convenabil să se postuleze existența **poziției** următoare **ultimului element** al **listei**.
  - În această idee se introduce funcția **FIN**( $L$ ):`TipPozitie` care returnează poziția următoare poziției  $n$  în lista  $L$  având  $n$  elemente.
  - Se observă că **FIN**( $L$ ) are o **distanță** variabilă față de începutul listei, funcție de faptul că lista crește sau se reduce, în timp ce alte poziții au o distanță fixă față de începutul listei.
- Pentru a defini un **tip de date abstract**, în cazul de față **TDA Listă**, este necesară:
  - (1) Definirea din punct de vedere matematic a **modelului asociat**, definire precizată mai sus.
  - (2) Definirea unui **set de operatori** aplicabili obiectelor de **tip listă**.
- Din păcate, pe de o parte este relativ greu de definit un set de operatori valabil în toate aplicațiile, iar pe de altă parte natura setului depinde esențial atât de **maniera de implementare** cât și de cea de **utilizare** a **listelor**.
- În continuare se prezintă **două seturi reprezentative de operatori** care acționează asupra listelor, unul restrâns și altul extins.

### 6.2.1. TDA Listă 1. Set de operatori restrâns

- Pentru a defini **setul restrâns** de operatori:
  - Se notează cu  $L$  o **listă** ale cărei noduri aparțin tipului de bază `TipNod`.
  - $x$ :`TipNod` este un **obiect** al acestui tip (deci un nod al listei).
  - $p$  este o variabilă de `TipPozitie`. Tipul poziție este dependent de implementare (indice, cursor, pointer, etc.) [AH85].

- În termenii formalismului utilizat în cadrul acestui manual, **TDA Listă** - varianta restrânsă apare în [6.2.1.a].
- **Modelul suport** al abordării îl constituie **structura tablou**.

---

### TDA Listă 1

#### (Set de operatori restrâns)

**Modelul matematic:** o secvență formată din zero sau mai multe elemente numite **noduri** toate încadrate într-un anumit tip numit **tip de bază**.

#### Notății:

*L*: *TipLista*;

*p*: *TipPozitie*;

*x*: *TipNod*.

[6.2.1.a]

#### Operatori:

1. **Fin**(*L*:*TipLista*):*TipPozitie*; - operator care returnează poziția următoare ultimului nod al listei, adică poziția următoare sfârșitului ei. În cazul listei vide **Fin**(*L*)=0;
2. **Insereaza**(*L*:*TipLista*,*x*:*TipNod*,*p*:*TipPozitie*); - inserează în lista *L*, nodul *x* în poziția *p*. Toate nodurile care urmează acestei poziții se mută cu un pas spre pozițiile superioare, astfel încât  $a_1, a_2, \dots, a_n$  devine  $a_1, a_2, \dots, a_{p-1}, x, a_p, \dots, a_n$ . Dacă *p* este **Fin**(*L*) atunci lista devine  $a_1, a_2, \dots, a_n, x$ . Dacă  $p > \text{Fin}(L)$  rezultatul este nedefinit;
3. **Cauta**(*x*:*TipNod*,*L*:*TipLista*):*TipPozitie*; - caută nodul *x* în lista *L* și returnează poziția nodului. Dacă *x* apare de mai multe ori, se furnizează poziția primei apariții. Dacă *x* nu apare de loc se returnează valoarea **Fin**(*L*);
4. **Furnizeaza**(*p*:*TipPozitie*,*L*:*TipLista*):*TipNod*; - operator care returnează nodul situat pe poziția *p* în lista *L*. Rezultatul este nedefinit dacă  $p = \text{Fin}(L)$  sau dacă în *L* nu există poziția *p*. Se precizează că tipul operatorului **Furnizeaza** trebuie să fie identic cu tipul de bază al listei;
5. **Suprima**(*p*:*TipPozitie*,*L*:*TipLista*); - suprimă elementul aflat pe poziția *p* în lista *L*. Dacă *L* este  $a_1, a_2, \dots, a_n$  atunci după suprimare *L* devine  $a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n$ . Rezultatul este nedefinit dacă *L* nu are poziția *p* sau dacă  $p = \text{Fin}(L)$ ;
6. **Urmator**(*p*:*TipPozitie*,*L*:*TipLista*):*TipPozitie*;

operator care returnează poziția următoare poziției  $p$  în cadrul listei. Dacă  $p$  este ultima poziție în  $L$  atunci **Urmator**( $p, l$ )=**Fin**( $L$ ). **Urmator** nu este definit pentru  $p=\mathbf{Fin}(L)$ ;

7.**Anterior**( $p:TipPozitie, L:TipLista$ ): $TipPozitie$ ; - operatori care returnează poziția anterioară poziției  $p$  în cadrul listei. Dacă  $p$  este prima poziție în  $L$  atunci **Anterior** nu este definit;

8.**Initializeaza**( $L:TipLista$ ): $TipPozitie$ ; - operator care face lista  $L$  vidă și returnează poziția **Fin**( $L$ )=0;

9.**Primul**( $L:TipLista$ ): $TipPozitie$ ; - returnează valoarea primei poziții în lista  $L$ . Dacă  $L$  este vidă, poziția returnată este **Fin**( $L$ )=0;

10.**TraverseazaLista**( $L:TipLista, ProcesareNod(...):PROCEDURE$ ); - parcurge nodurile listei  $L$  în ordinea în care apar ele în listă și aplică fiecăruia procedura *ProcesareNod*.

- 
- **Exemplul 6.2.1.** Pentru a ilustra utilitatea acestui set de operatori se consideră un exemplu tipic de aplicație.

- Fiind dată o **listă de adrese de persoane**, se cere să se elimine **duplicatele**.
  - Conceptual acest lucru este simplu: pentru fiecare nod al listei se elimină nodurile echivalente care-i urmează.
  - Definind o structură de date specifică, în termenii operatorilor anterior definiți, algoritmul de eliminare a adreselor duble din listă poate fi formulat astfel [6.2.1.b].

---

{Exemplu - Eliminarea nodurilor duplicat din cadrul unei liste}

```

TYPE TipNod=RECORD
    NrCurent: integer;
    Nume: string[20];
    Adresa: string[40]
END;
[6.2.1.b]

PROCEDURE Elimina(VAR L: TipLista);
    {procedura suprimă duplicatele nodurilor din listă}
VAR p,q: TipPozitie; {p indică poziția curentă}
                    {q este utilizat în căutare}
BEGIN
    p:= Primul(L);
    WHILE p<>Fin(L) DO
        BEGIN
            q:= Urmator(p,L);

```

```

        WHILE q<>Fin(L) DO
            IF Furnizeaza(p,L)=Furnizeaza(q,L) THEN
                Suprima(q,L)
            ELSE
                q:= Urmator(q,L);
            p:= Urmator(p,L)
        END;
    END; {Elimina}
-----
/* Exemplu - Eliminarea nodurilor duplicat din cadrul unei
liste */

typedef struct{
    int nr_curent;
    char* nume;
    char* adresa;
}tip_nod;                                /*[6.2.1.b]*/

void elimina(tip_lista *l)
    /*procedura suprimă duplicatele nodurilor din listă*/
{
    tip_pozitie p,q; /*p indică poziția curentă*/
                        /*q este utilizat în căutare*/
    p= primul(*l);
    while (p!=fin(*l)){
        q= urmator(p,*l);
        while (q!=fin(*l))
            if (furnizeaza(p,*l)==furnizeaza(q,*l))
                suprima(q,*l);
            else
                q= urmator(q,*l);
        p= urmator(p,*l);
    }
} /*elimina*/
/*-----*/

```

- În legătură cu cea de-a doua buclă **WHILE**, se poate face o **observație importantă** referitoare la variabila q.
  - Dacă se suprimă din listă elementul situat pe poziția q, elementele aflate pe pozițiile q+1, q+2, etc, retrogradează cu o poziție în listă.
  - Dacă în mod întâmplător q este ultimul element al listei, atunci valoarea sa devine **Fin(L)**.
  - Dacă în continuare s-ar executa instrucția q:=**Urmator**(q,L), lucru dictat de logica algoritmului, s-ar obține o valoare nedeterminată pentru q.
  - Din acest motiv, s-a prevăzut ca trecerea la elementul următor să se facă numai după o **nouă verificare a condiției**, respectiv dacă condiția instrucției **IF** este adevărată se execută **numai** suprimarea iar în caz contrar **numai** avansul.

---

## 6.2.2. TDA Listă 2. Set de operatori extins

- În același context, în continuare se prezintă un al **doilea set de operatori** referitori la liste având o complexitate mai ridicată [6.2.2.a].
- **Modelul suport** al reprezentării listei avute în vedere se bazează pe **înlănțuiri** [SH90].

---

### TDA Listă 2

#### (Set de operatori extins)

[6.2.2.a]

**Modelul matematic:** o secvență finită de noduri. Toate nodurile aparțin unui același tip numit **tip de bază**. Fiecare nod constă din două părți: o parte de informații și o a doua parte conținând legătura la nodul următor. O variabilă specială indică primul nod al listei.

#### Notății:

- *TipNod* - tipul de bază;
- *TipLista* - tip indicator la tipul de bază;
- *TipInfo* - partea de informații a lui *TipNod*;
- *TipIndicatorNod* - tip indicator la tipul de bază (identic cu *TipLista*);
- *incLista*: *TipLista* - variabilă care indică începutul listei;
- *curent*, *p*, *pnod*: *TipIndicatorNod* - indică noduri în listă;
- *element*: *TipNod*;
- *info*: *TipInfo*; parte de informații a unui nod;
- *b* - valoare booleană;
- **nil** - indicatorul vid.

#### Operatori:

1. **CreazaListaVida**(*incLista*: *TipLista*); - variabila *incLista* devine **nil**.
2. **ListaVida**(*incLista*: *TipLista*): *boolean*; - operator care returnează **TRUE** dacă lista este vidă respectiv **FALSE** altfel.
3. **Primul**(*incLista*: *TipLista*, *curent*: *TipIndicatorNod*); - operator care face ca variabila *curent* să indice primul nod al listei precizată de *incLista*.
4. **Ultimul**(*curent*: *TipIndicatorNod*): *boolean*; -

operator care returnează **TRUE** dacă *curent* indică ultimul element al listei.

5. **InserInceput**(*incLista: TipLista, pnod: TipIndicatorNod*); - inserează la începutul listei *incLista* nodul indicat de *pnod*.
6. **InserDupa**(*curent, pnod: TipIndicatorNod*); - inserează nodul indicat de *pnod* după nodul indicat de *curent*. Se presupune că *curent* indică un nod din listă.
7. **InserInFatza**(*curent, pnod: TipIndicatorNod*); - insertie în fața nodului *curent*.
8. **SuprimaPrimul**(*incLista: TipLista*); - suprimă primul nod al listei *incLista*.
9. **SuprimaUrm**(*curent: TipIndicatorNod*); - suprimă nodul următor celui indicat de *curent*.
10. **SuprimaCurent**(*curent: TipIndicatorNod*); - suprimă nodul indicat de *curent*.
11. **Urmatorul**(*curent: TipIndicatorNod*); - *curent* se poziționează pe următorul nod al listei. Dacă *curent* indică ultimul nod al listei el va deveni **nil**.
12. **Anterior**(*curent: TipIndicatorNod*); - *curent* se poziționează pe nodul anterior celui *curent*.
13. **MemoreazaInfo**(*curent: TipIndicatorNod, info: TipInfo*); - atribuie nodului indicat de *curent* informația *info*.
14. **MemoreazaLeg**(*curent, p: TipIndicatorNod*); - atribuie câmpului *urm* (de legătură) al nodului indicat de *curent* valoarea *p*.
15. **FurnizeazaInfo**(*curent: TipIndicatorNod*):*TipInfo*; - returnează partea de informație a nodului indicat de *curent*.
16. **FurnizeazaUrm**(*curent: TipIndicatorNod*):*TipIndicator Nod*; - returnează legătura nodului *curent* (valoarea câmpului *urm*).
17. **TraverseazaLista**(*incLista: TipLista, ProcesareNod (...): PROCEDURE*); - parcurge nodurile listei *L* în ordinea în care apar ele în listă și aplică fiecăruia procedura *ProcesareNod*.

-----

### 6.3. Tehnici de implementare a listelor

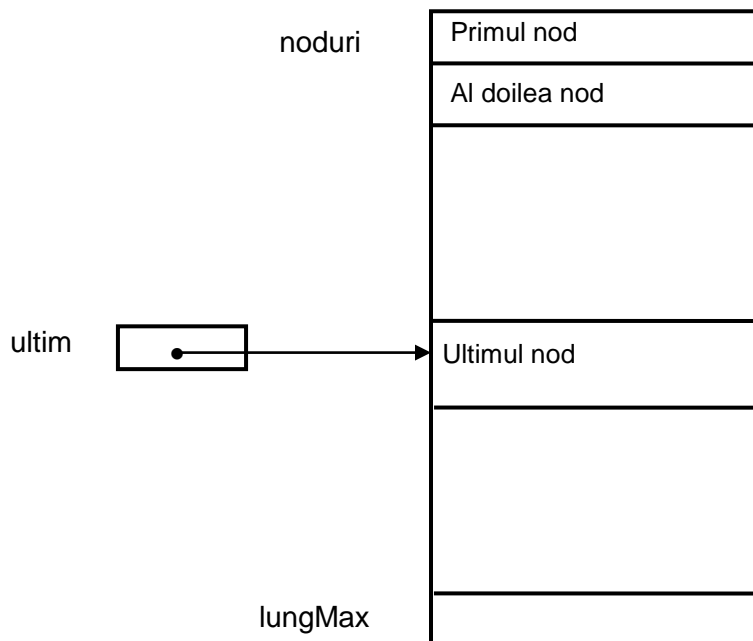
- De regulă pentru structurile de date **fundamentale** există **construcții de limbaj** care le reprezintă, construcții care își găsesc un anumit corespondent în particularitățile hardware ale sistemelor care le implementează.
  - Pentru **structurile de date avansate** însă, care se caracterizează printr-un nivel mai înalt de abstractizare, acest lucru **nu** mai este valabil.
- De regulă, **reprezentarea structurilor de date avansate** se realizează cu ajutorul **structurilor de date fundamentale**, observație valabilă și pentru structura listă.
- Din acest motiv, în cadrul acestui paragraf :
  - Vor fi prezentate câteva dintre **structurile de date fundamentale** care pot fi utilizate în reprezentare **listelor**.
  - **Procedurile și funcțiile** care implementează **operatorii specifici** prelucrării listelor vor fi descriși în termenii acestor structuri.

#### 6.3.1. Implementarea listelor cu ajutorul structurii tablou

- În cazul **implementării listelor** cu ajutorul **structurii tablou**:
  - O **listă** se asimilează cu un **tablou**.
  - **Nodurile listei** sunt **memorate** într-o zonă contiguă **în locații succesive de memorie**.
- În această reprezentare:
  - O listă poate fi ușor **traversată**.
  - Noile noduri pot fi **adăugate** în mod simplu la **sfârșitul listei**.
  - **Insertia** unui nod în mijlocul listei presupune însă deplasarea tuturor nodurilor următoare cu o poziție spre sfârșitul listei pentru a face loc noului nod.
  - **Suprimarea** oricărui nod cu excepția ultimului, presupune de asemenea deplasarea tuturor celorlalte în vederea eliminării spațiului creat.
  - Insertia și suprimarea unui nod necesită un **efort** de execuție  $O(n)$ .



- În **implementarea bazată pe tablouri**, `TipLista` se definește ca o **structură (articol)** cu două câmpuri.
  - (1) Primul câmp este un **tablou** numit `noduri`, cu elemente de `TipNod`.
    - Lungimea acestui tablou este astfel aleasă de către programator încât să fie suficientă pentru a putea păstra **cea mai mare dimensiune de listă** ce poate apare în respectiva aplicație.
  - (2) Cel de-al doilea câmp este un **indicator** (`ultim`) care indică poziția în tablou a **ultimului nod** al listei.
- Cel de-al  $i$ -lea nod al listei se găsește în cel de-al  $i$ -lea element al tabloului, pentru  $1 \leq i \leq \text{ultim}$  (fig.6.3.1.a).
- **Poziția** în cadrul listei se reprezintă prin valori întregi, respectiv cea de-a  $i$ -a poziție prin valoarea  $i$ .
- Funcția ***Fin***(`L`) returnează valoarea `ultim+1`.



**Fig.6.3.1.a.** Implementarea listelor cu ajutorul structurii tablou.

- O variantă a unei astfel de implementări apare în secvența [6.3.1.a].

---

{Implementarea listelor cu ajutorul structurii tablou -  
varianta Pascal}

CONST lungMax=...;

```

TYPE TipLista=RECORD [6.3.1.a]
    noduri:ARRAY[1..lungMax] OF TipNod;
    ultim:TipIndice
END;

    TipPozitie=TipIndice;
-----
/* Implementarea listelor cu ajutorul structurii tablou -
varianța C*/

#define lung_max = 100

typedef struct { /*[6.3.1.a]*/
    tip_nod noduri[lung_max];
    tip_indice ultim;
} tip_lista;

typedef tip_indice tip_pozitie;
/*-----*/

```

- Secvența de program [6.3.1.b] prezintă modul în care se pot implementa operațiile specifice setului **restrâns** de operatori: **Fin**, **Inseereaza**, **Suprima** și **Cauta** utilizând implementarea bazată pe tablouri a listelor.
- Se fac următoarele precizări:
  - Dacă se încearcă inserția unui nod într-o listă care deja a utilizat în întregime tabloul asociat se semnalează un mesaj de **eroare**.
  - Dacă în cursul procesului de căutare **nu** se găsește elementul căutat, **Cauta** returnează poziția ultim+1.
  - S-a prevăzut parametrul boolean er, care în caz de **eroare** se returnează cu valoarea adevărat și care poate fi utilizat pentru tratarea erorii sau pentru întreruperea execuției programului.

```

-----
{Implementarea setului restrâns de operatori referitori la
liste: Fin, Inseereaza, Suprima, Cauta cu ajutorul structurii
tablou - varianța Pascal}

```

```

FUNCTION Fin(VAR L: TipLista): TipPozitie;
BEGIN
    Fin:= L.ultim+1    {performanța O(1)}
END; {Fin}

```

```

PROCEDURE Inseereaza(VAR L: TipLista; x: TipNod;
    p: TipPozitie; VAR er: boolean);
    {plasează pe x în poziția p a listei; performanța O(n)}
    VAR q: TipPozitie;
BEGIN [6.3.1.b]
    er:= false;

```

```

IF L.ultim>=lungMax THEN
    BEGIN
        er:= true; mesaj('lista este plină')
    END
ELSE
    IF (p>L.ultim+1) OR (p<1) THEN
        BEGIN
            er:= true; Write('poziția nu există')
        END
    ELSE
        BEGIN
            FOR q:= L.ultim DOWNTO p DO
                L.noduri[q+1]:= L.noduri[q];
            L.ultim:= L.ultim+1;
            L.noduri[p]:= x
        END
    END; {Insereaza}

```

```

PROCEDURE Suprima(p: TipPozitie; VAR L: TipLista,
    er: boolean);
    {extrage elementul situat pe poziția p a listei}
VAR q: TipPozitie;
BEGIN                                     {performanța O(n)}
    er:= false;
    IF (p>L.ultim) OR (p<1) THEN
        BEGIN
            er:= true; mesaj('poziția nu există')
        END
    ELSE
        BEGIN
            L.ultim:= L.ultim-1;
            FOR q:=p TO l.ultim DO
                L.noduri[q]:= L.noduri[q+1]
        END
    END; {Suprima}

```

```

FUNCTION Cauta(x: TipNod; L: TipLista): TipPozitie;
    {returnează poziția lui x în listă}
VAR q: TipPozitie; gasit: boolean;
BEGIN
    q:= 1; gasit:= false;                 {performanța O(n)}
    REPEAT
        IF L.noduri[q]=x THEN
            BEGIN
                Cauta:= q; gasit:= true
            END;
        q:= q+1
    UNTIL gasit OR (q=L.ultim+1);
    IF NOT gasit THEN Cauta:= L.ultim+1
END; {Cauta}

```

---

```
/* Implementarea setului restrâns de operatori referitori la  
liste: Fin, Insereaza, Suprima, Cauta cu ajutorul structurii  
tablou - varianta C */
```

```
#include <stdlib.h>
```

```
#define lung_max 100
```

```
#define n 30
```

```
typedef int tip_indice;
```

```
typedef struct{  
    int nr_curent;  
    char* nume;  
    char* adresa;  
}  
tipnod;
```

```
typedef struct{  
    tip_nod noduri[lung_max];  
    tip_indice ultim;  
}  
tiplista;  
typedef tip_indice tip_pozitie;
```

```
typedef unsigned boolean;
```

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

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

```
boolean reccmp(tip_nod, tip_nod); /*comparare conținut  
noduri*/
```

```
tip_pozitie fin(tip_lista* l){  
    tip_pozitie fin_result;  
    fin_result= l->ultim+1; /*performata O(1)*/  
    return fin_result;  
} /*fin*/
```

```
void insereaza(tip_lista* l, tip_nod x,  
               tip_pozitie p, boolean* er)  
/*plasează pe x în poziția p a listei; performanța O(n)*/  
{  
    tip_pozitie q; /*[6.3.1.b]*/  
    *er= false;  
    if (l->ultim>=lung_max){  
        *er= true;  
        printf("lista este plina");  
    }  
    else  
        if ((p>l->ultim+1) || (p<1)){  
            *er= true;  
            printf("poziția nu există");  
        }  
        else{  
            for( q= l->ultim; q >= p; q --)  
                l->noduri[q]= l->noduri[q-1];  
            l->ultim= l->ultim+1;
```

```

        l->noduri[p-1]= x;
    }
} /*insereaza*/

void suprima(tip_pozitie p, tip_lista* l, boolean er)
/*extrage elementul din poziția p a listei*/
{
    tip_pozitie q; /*performanța O(n)*/
    er= false;
    if ((p>l->ultim) || (p<1)){
        er= true;
        printf("pozitia nu exista");
    }
    else{
        l->ultim= l->ultim-1;
        for( q=p; q <= l->ultim; q ++){
            l->noduri[q-1]= l->noduri[q];
        }
    }
} /*suprima*/

tip_pozitie cauta(tip_nod x, tip_lista l)
/*returnează poziția lui x în listă*/
{
    tip_pozitie q;
    boolean gasit;
    tip_pozitie cauta_result;
    q= 1; gasit= false; /*performanța O(n)*/
    do {
        if (reccmp(l.noduri[q-1], x) == 0){
            cauta_result= q;
            gasit= true;
        }
        q= q+1;
    } while (!(gasit||(q==l.ultim+1)));
    if (!gasit) cauta_result= l.ultim+1;
    return cauta_result;
} /*cauta*/

boolean reccmp(tip_nod x, tip_nod y)
/*comparare conținut noduri*/
{
    if ((x.nr_curent == y.nr_curent) &&
        !(strcmp(x.num, y.num, n)) &&
        !(strcmp(x.adresa, y.adresa, n)))
        return true;
    else
        return false;
}
/*-----*/

```

• În acest context, implementarea celorlalți operatori **nu** ridică probleme deosebite:

- Operatorul **Primul** returnează întotdeauna valoarea 1.
- Operatorul **Urmator** returnează valoarea argumentului incrementată cu 1.

- Operatorul **Anterior** returnează valoarea argumentului diminuată cu 1 după ce în prealabil s-au făcut verificările de limite.
- Operatorul **Initializare** face pe `L.ultim` egal cu 0.
- La prima vedere pare tendențioasă redactarea unor proceduri care să guverneze **toate accesese** la o anumită **structură de date**.
- Cu toate acestea acest lucru are o importanță cu totul remarcabilă, fiind legat de utilizarea conceptului de "**obiect**" în exploatarea structurilor de date.
  - Dacă programatorul va redacta programele în **termenii operatorilor** care manipulează **tipurile abstracte de date** în loc de a face în **mod direct** uz de **detaaliile lor de implementare**:
    - (1) Pe de-o parte crește **eleganța** și **siguranța** în funcționare a programului.
    - (2) Pe de altă parte **modificarea** programului sau a structurii de date propriu-zise se poate realiza facil, **doar** prin modificarea procedurilor care reglementează accesese la ea, fără a mai fi necesară căutarea și modificarea în program a locurilor din care se fac accese la respectiva structură.
    - (3) Această **flexibilitate** poate să joace de asemenea un rol esențial în cazul efortului necesar dezvoltării unor produse software de mari dimensiuni.

### 6.3.2. Implementarea listelor cu ajutorul pointerilor

- **Listele liniare** se pot implementa și cu ajutorul **tipului de date pointer**.
- Deoarece o **listă liniară** este o **structură dinamică** ea poate fi definită în **termeni recursivi** după cum urmează [6.3.2.a]:

-----  
 {Implementarea listelor cu ajutorul pointerilor -  
 implementarea ca structură de date recursivă - varianta  
 Pascal}

**TYPE** TipPointerNod = ^TipNod; [6.3.2.a]

```

TipNod = RECORD
    cheie: integer;
    urm: TipPointerNod;
    info: TipInfo
END;
```

```

TipLista = TipPointerNod;
```

-----

```

/* Implementarea listelor cu ajutorul pointerilor -
implementarea ca structură de date recursivă - varianta C */

typedef int tip_info;

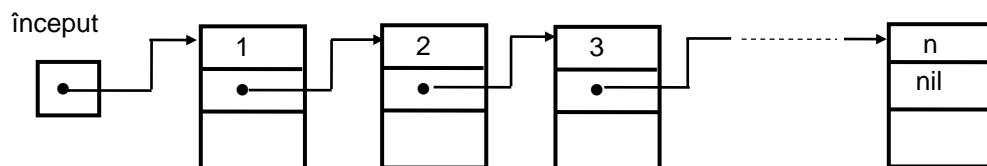
typedef struct tip_nod* tip_pointer_nod;          /*[6.3.2.a]*/

typedef struct {
    int cheie;
    tip_pointer_nod urm;
    tip_info info;
} tip_nod;

typedef tip_pointer_nod tip_lista;
/*-----*/

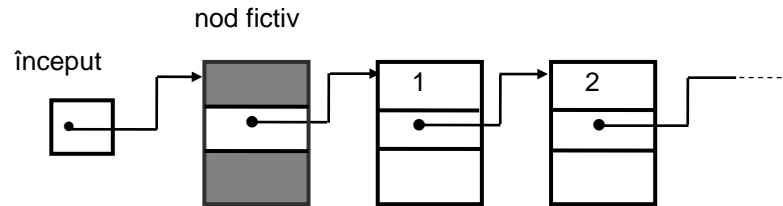
```

- După cum se observă, în cazul definirii unui **nod al structurii listă înlănțuită** s-au pus în evidență trei câmpuri:
  - O **cheie** care servește la identificarea nodului.
  - Un **pointer** de înlănțuire la nodul următor.
  - Un **câmp** `info` conținând informația utilă.
- În figura 6.3.2.a apare reprezentarea unei astfel de **liste liniare** împreună cu o variabilă pointer `inceput` care indică primul nod.
  - **Lista liniară** din figură are **particularitatea** că valoarea cheii fiecărui nod este egală cu numărul de ordine al nodului.
- În secvența [6.3.2.a] se observă că o listă liniară poate fi definită ca și o **structură recursivă** având o componentă de tip **identic** cu cel al structurii complete.
  - Caracteristica esențială a unei astfel de structuri rezidă în prezența **unei singure înlănțuiri**.



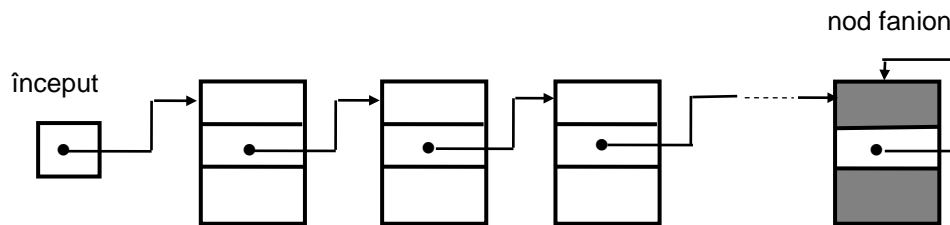
**Fig.6.3.2.a.** Exemplu de listă liniară

- În continuare, în cadrul acestui paragraf se prezintă câteva **tehnici de implementare a listelor liniare** ca și **structuri recursive**.
- (1) Există posibilitatea ca **pointerul** început care indică începutul listei, să indice o **componentă** de tipNod având câmpurile cheie și info neasignate, iar câmpul urm al acesteia, să indice **primul nod** efectiv al listei.
  - Utilizarea acestui nod de început, cunoscută sub denumirea de **tehnica nodului fictiv** simplifică în multe situații prelucrarea listelor înlanțuite (fig.6.3.2.b).



**Fig.6.3.2.b.** Implementarea listelor prin tehnica nodului fictiv

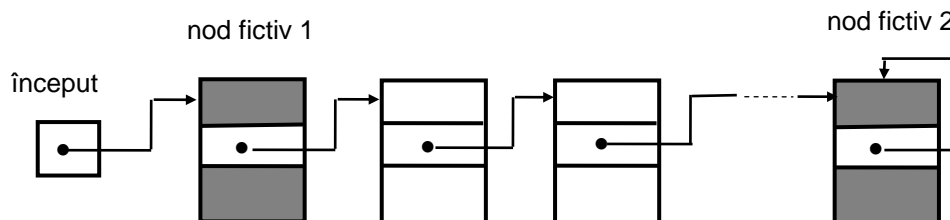
- (2) Există de asemenea posibilitatea utilizării unui **nod fictiv final** pe post de **fanion** având înlanțuirea **nil** sau care se înlanțuie cu el însuși [Se88].
  - Această tehnică de implementare este cunoscută sub denumirea de **tehnica nodului fanion** (fig.6.3.2.c).



**Fig.6.3.2.c.** Implementarea listelor prin tehnica nodului fanion

- (3) O altă posibilitate de implementare o reprezintă utilizarea a **două noduri fictive**, unul inițial și un altul final - **tehnica celor două noduri fictive** (fig.6.3.2.d).





**Fig.6.3.2.d.** Implementarea listelor cu ajutorul tehnicii celor două noduri fictive

- Fiecare dintre modalitățile de implementare prezentate au **avantaje specifice** care vor fi evidențiate pe parcursul capitolului.

### 6.3.2.1. Tehnici de inserție a nodurilor și de creare a listelor înlănțuite

- Presupunând că este dată o **structură de date listă**, în continuare se prezintă o secvență de cod pentru inserția unui nod nou în listă.
- Inițial, **inserția** se execută **la începutul** listei.
  - Se consideră că **inceput** este o **variabilă pointer** care indică **primul** nod al listei.
  - Variabila **auxiliar** este o variabilă pointer ajutătoare [6.3.2.1.a].

---

#### {Inserție la începutul listei - varianta Pascal}

```
[1] new(auxiliar);
[2] auxiliar^.urm:= inceput;      {performanța O(1)}
[3] inceput:= auxiliar;
[4] inceput^.info:= ...;          [6.3.2.1.a]
```

---

#### /\*inserție la începutul listei - varianta C \*/

```
[1] auxiliar= (tip_nod*)malloc(sizeof(tip_nod));
[2] auxiliar->urm= inceput;      /*performanța O(1)*/
[3] inceput= auxiliar;
[4] inceput->info= ...;          /*[6.3.2.1.a]*/
/*-----*/
```

- În figura 6.3.2.1.a se prezintă grafic maniera în care se desfășoară o astfel de inserție.
- Pe baza acestui fragment de program se prezintă în continuare, **crearea unei liste înlănțuite**.
  - Se pornește cu o **listă vidă** în care se inserează pe rând câte un nod la începutul listei până când numărul nodurilor devine egal cu un număr dat n.

- În secvență s-a omis asignarea câmpurilor de informație [6.3.2.1.b].
- Datorită faptului că inserția noului nod are loc de fiecare dată la **începutul** listei, secvența creează lista în **ordinea inversă** a furnizării cheilor.

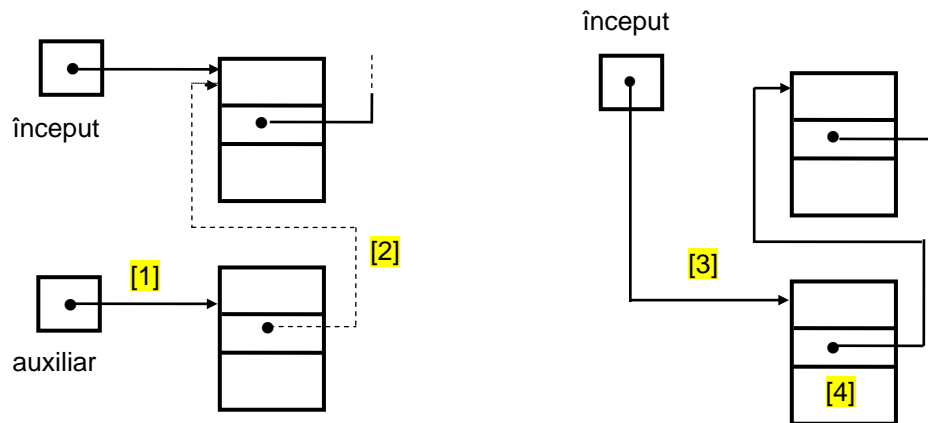


Fig.6.3.2.1.a. Inserția unui nod la începutul unei liste înlănțuite

---

**{Crearea unei liste înlănțuite}**

```

inceput:= nil; {se pornește cu lista vidă}
WHILE n>0 DO
  BEGIN
    new(auxiliar);
    auxiliar^.urm:= inceput;
    inceput:= auxiliar;
    auxiliar^.cheie:= n;
    n:= n-1
  END;

```

---

**/\*crearea unei liste înlănțuite\*/**

```

inceput= NULL; /*se pornește cu lista vidă*/
while (n>0){
  auxiliar = (tip_nod*)malloc(sizeof(tip_nod));
  auxiliar->urm= inceput;
  inceput= auxiliar;
  auxiliar->cheie= n;
  n=n-1;
}
/*-----*/

```

- Dacă se dorește crearea listei în **ordine naturală**, atunci este nevoie de o secvență care inserează un nod **la sfârșitul** unei liste.

- Această secvență de program se redactează mai simplu dacă se cunoaște locația **ultimului** nod al listei.
  - Teoretic lucrul acesta **nu** prezintă nici o dificultate, deoarece se poate parcurge lista de la începutul ei (indicat prin `inceput`) până la detectarea nodului care are câmpul `urm = nil`.
  - În practică această soluție **nu** este convenabilă, deoarece parcurgerea de fiecare dată a întregii liste este **ineficientă**.
  - Se preferă să se lucreze cu o **variabilă pointer ajutătoare** `ultim` care indică mereu **ultimul nod al listei**, după cum `inceput` indică mereu **primul nod**.
- În prezența lui `ultim`, secvența de program care inserează un nod la sfârșitul unei liste liniare și concomitent îl actualizează pe `ultim` este următoarea [6.3.2.1.c]:

-----  
**{Insertie la sfârșitul unei liste înlănțuite}**

[6.3.2.1.c]

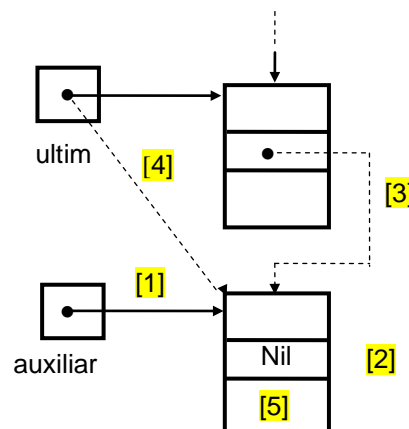
```
[1] new(auxiliar);
[2] auxiliar^.urm:= nil;                               {performanța O(1)}
[3] ultim^.urm:= auxiliar;
[4] ultim:= auxiliar;
[5] ultim^.info:= ...;
```

-----  
**/\*insertie la sfârșitul unei liste înlănțuite\*/**

/\*[6.3.2.1.c]\*/

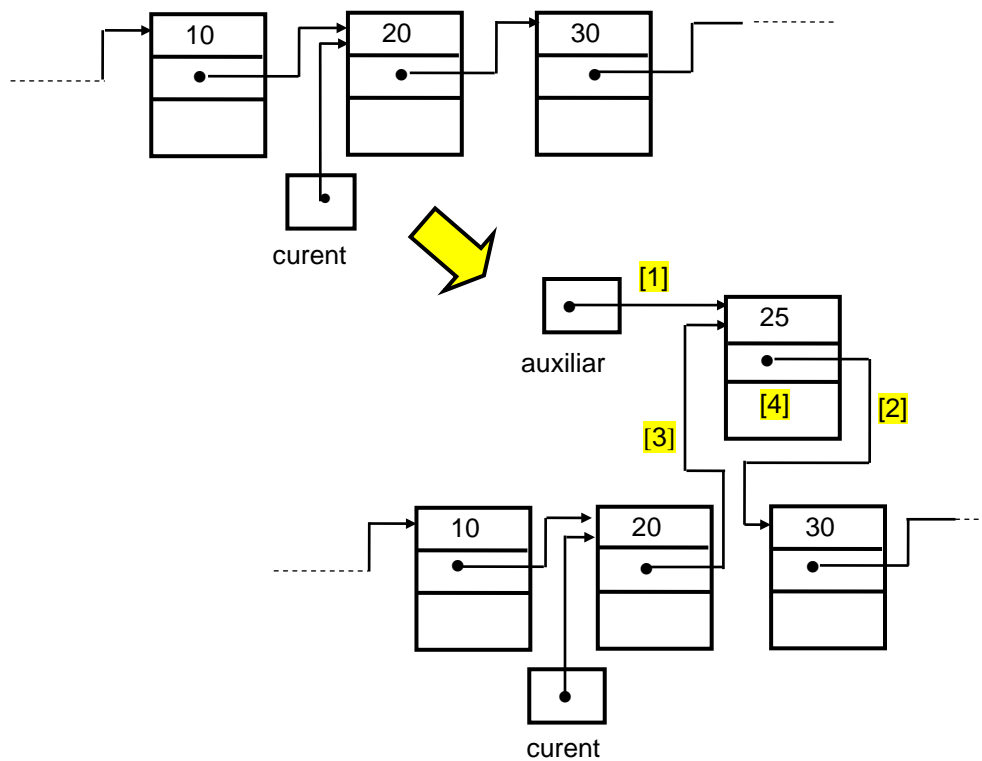
```
[1] auxiliar= (tip_nod*)malloc(sizeof(tip_nod));
[2] auxiliar->urm= NULL;                               /*performanța O(1)*/
[3] ultim->urm= auxiliar;
[4] ultim= auxiliar;
[5] ultim->info= ...;
/*-----*/
```

- Reprezentarea grafică a acestei inserții apare în figura 6.3.2.1.b.



**Fig.6.3.2.1.b.** Inserția unui nod la sfârșitul unei liste înlănțuite

- Referitor la secvența [6.3.2.1.c] se atrage atenția că ea **nu** poate **insera un nod** într-o **listă vidă**.
  - Acest lucru se observă imediat întrucât în acest caz `ultim^urm` **nu există**.
- Există mai multe posibilități de a rezolva această problemă:
  - (1) Primul nod trebuie inserat printr-un alt procedeu spre exemplu prin **inserție la începutul listei**.
    - În continuare nodurile se pot adăuga conform secvenței precizate.
  - (2) O altă posibilitate de a rezolva această problemă o constituie utilizarea unei liste implementate cu ajutorul **tehnicii nodului fictiv**.
    - În acest caz, primul nod al listei există întotdeauna și ca atare `ultim^urm` există chiar și în cazul unei liste vide.
  - (3) O a treia posibilitate este aceea de a utiliza o listă implementată cu ajutorul **tehnicii nodului fanion**.
    - În acest caz nodul de inserat se introduce peste nodul fanion și se creează un nou nod fanion.
- În continuare se descrie inserția unui nod nou **într-un loc oarecare** al unei **liste**.
  - Fie `curent` un pointer care indică un nod listei,
  - Fie `auxiliar` o variabilă pointer ajutătoare.
- În aceste condiții **inserția unui nod nou după nodul indicat** de `curent` se realizează conform figurii 6.3.2.1.c în care nodul nou inserat are cheia 25.



**Fig.6.3.2.1.c.** Inserția unui nod nou după un nod precizat (curent)

- Secvența de cod care realizează această inserție apare în [6.3.2.1.d].

**{Inserția unui nod nou după un nod precizat de indicatorul  
curent}**

[6.3.2.1.d]

```
[1] new(auxiliar);
[2] auxiliar^.urm:= curent^.urm;    {performanța O(1)}
[3] curent^.urm:= auxiliar;
[4] auxiliar^.info:= ...;
```

**/\*inserția unui nod nou după un nod precizat de indicatorul  
curent\*/**

/\*[6.3.2.1.d]\*/

```
[1] auxiliar= (tip_nod*)malloc(sizeof(tip_nod));
[2] auxiliar->urm= curent->urm;    /*performanța O(1)*/
[3] curent->urm= auxiliar;
[4] auxiliar->info= ...;
/*-----*/
```

- Dacă se dorește însă inserția noului nod în lista liniară **înaintea unui nod indicat** de pointerul **curent**, apare o complicație generată de imposibilitatea practică de a afla simplu, **adresa predecesorului** nodului indicat de **curent**.
  - După cum s-a precizat deja, în practică **nu** se admite parcurgerea de la început a listei până la detectarea nodului respectiv.
- Această problemă se poate însă rezolva simplu cu ajutorul următoarei **tehnici**:

- (1) Se crează un nod nou.
  - (2) Se asignează integral acest nod cu conținutul nodului indicat de curent.
  - (3) Se inserează noul nod după nodul indicat de pointerul curent.
  - (4) Se creează câmpurile cheie și info pentru noul nod și se asignează cu ele câmpurile corespunzătoare ale vechiului nod indicat de pointerul curent.
- Secvența de cod care implementează această tehnică apare în [6.3.2.1.e] iar reprezentarea sa grafică a în figura 6.3.2.1.d.

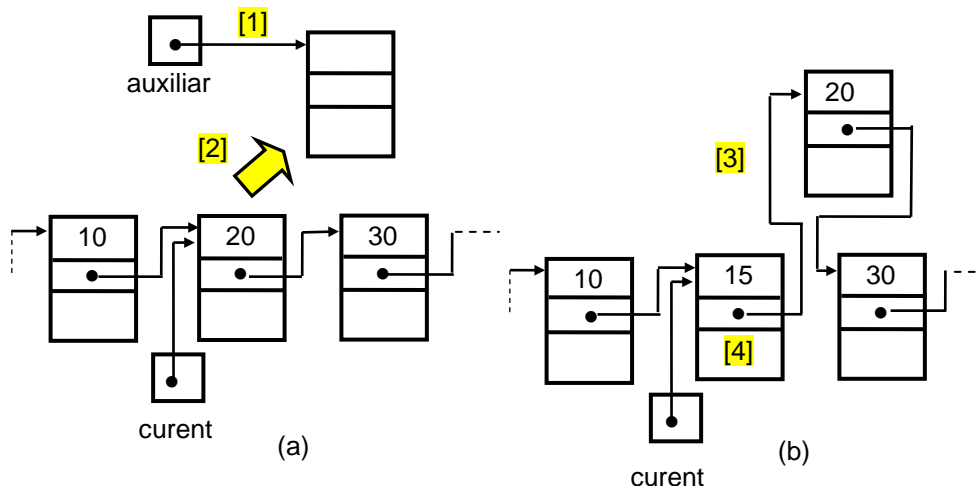
---

**{Insertia unui nod nou înaintea unui nod precizat de indicatorul curent}**

[6.3.2.1.e]

```
[1] new(auxiliar);
[2] auxiliar^:= curent^;           {performanța O(1)}
[3] curent^.urm:= auxiliar;
[4] curent^.info:= ...;
```

---



**Fig.6.3.2.1.d.** Inserția unui nod nou în fața unui nod indicat

```
/*-----*/
/*insertia unui nod nou înaintea unui nod precizat de
indicatorul curent*/
/*[6.3.2.1.e]*/
[1] auxiliar = (tip_nod*)malloc(sizeof(tip_nod));
[2] *auxiliar= *curent;           /*performanța O(1)*/
[3] curent->urm= auxiliar;
[4] curent->info= ...;
/*-----*/
```

### 6.3.2.2. Tehnici de suprimare a nodurilor

- Se consideră următoarea **problemă**:
  - Se dă un pointer `curent` care indică un nod al unei liste liniare înlănțuite și se cere să **se suprimă succesorul nodului indicat** de `curent`.
  - Această suprimare se poate realiza prin intermediul fragmentului de cod [6.3.2.2.a] în care `auxiliar` este o variabilă pointer ajutătoare.

---

**{Suprimarea succesorului nodului precizat de indicatorul `curent` (varianta 1)}**

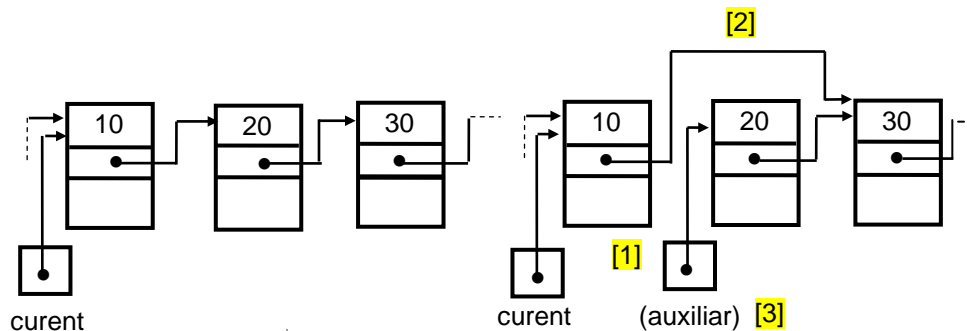
```
[1] auxiliar:= curent^.urm;           {performanța O(1)}
[2] curent^.urm:= auxiliar^.urm       [6.3.2.2.a]
[3] Dispose(auxiliar);
```

---

**/\*suprimarea succesorului nodului precizat de indicatorul `curent` (varianta 1)\*/**

```
[1] auxiliar= curent->urm;           /*performanța O(1)*/
[2] curent->urm= auxiliar->urm;       /*[6.3.2.2.a]*/
[3] free(auxiliar);
/*-----*/
```

- Efectul execuției aceste secvențe de cod se poate urmări în figura 6.3.2.2.a.



**Fig.6.3.2.2.a.** Tehnica suprimării succesorului nodului indicat de `curent`

- Se observă că secvența de cod de mai sus se poate **înlocui** cu următoarea secvență în care nu mai este necesar pointerul `auxiliar` [6.3.2.2.b]:

---

**{Suprimarea succesorului nodului precizat de indicatorul `curent` (varianta 2)}**

{performanța O(1)}

```
curent^.urm:= curent^.urm^.urm;      [6.3.2.2.b]
```

---

**/\*suprimarea succesorului nodului precizat de indicatorul `curent` (varianta 2)\*/**

/\*performanța O(1)\*/

```

        curent->urm= curent->urm->urm;          /*[6.3.2.2.b]*/
/*-----*/

```

- Utilizarea pointerului auxiliar are însă avantajul că prin intermediul lui, programatorul poate avea acces ulterior la nodul suprimat din listă în vederea disponibilizării zonei de memorie alocate lui, zonă care în condițiile execuției secvenței [6.3.2.2.b] se pierde.
- Revenind la problema suprimării unui nod, se consideră cazul în care se dorește **suprimarea** nodului **indicat** de curent.
  - Aici apare aceeași dificultate semnalată în paragraful anterior, generată de imposibilitatea aflării simple a **adresei predecesorului nodului** indicat de pointerul curent.
- Soluția se bazează pe aceeași **tehnică**:
  - (1) Se copiază conținutul nodului succesor integral în nodul indicat de curent.
  - (2) Se suprimă nodul succesor.
- Aceasta se poate realiza printr-o singură instrucție și anume [6.3.2.2.c]:

```

-----
{Suprimarea nodului precizat de indicatorul curent
 (varianta 1)}
                                {performanța O(1)}
        curent^:= curent^.urm^;          [6.3.2.2.c]
-----
/*suprimarea nodului precizat de indicatorul curent
 (varianta 1)*/
                                /*performanța O(1)*/

        *curent= *curent->urm;          /*[6.3.2.2.c]*/
/*-----*/

```

- Ca și înainte, această soluție prezintă **dezavantajul** că pierde iremediabil zona de memorie ocupată inițial de succesorul nodului indicat de pointerul curent.
- O soluție care evită acest dezavantaj este cea prezentată în secvența [6.3.2.2.d]:

```

-----
{Suprimarea nodului precizat de indicatorul curent
 (varianta 2)}

        auxiliar:= curent^.urm;          {performanța O(1)}
        curent^:= auxiliar^;
        Dispose(auxiliar);              [6.3.2.2.d]
-----
/*suprimarea nodului precizat de indicatorul curent
 (varianta 2)*/

```



```

    auxiliar= curent->urm;           /*performanța O(1)*/
    *curent= *auxiliar;
    free(auxiliar);                 /*[6.3.2.2.d]*/
/*-----*/

```

- Se remarcă însă faptul că ambele tehnici de suprimare se pot aplica **numai** dacă nodul indicat de curent **nu este ultimul nod al listei**, respectiv numai dacă  $curent^.urm \neq nil$ .
- Pentru a evita acest neajuns se pot utiliza alte modalități de implementare a listei înlănțuite, spre exemplu **tehnica nodului fanion**.

### 6.3.2.3. Traversarea unei liste înlănțuite. Căutarea unui nod într-o listă înlănțuită

- Prin **traversarea** unei liste se înțelege executarea în manieră ordonată a unei anumite operații asupra tuturor nodurilor listei.
  - Fie pointerul p care indică **primul nod al listei** și fie curent o variabilă pointer auxiliară.
  - Dacă curent este un nod oarecare al listei se notează cu **Prelucrare(curent)** operația amintită, a cărei natură nu se precizează.
- În aceste condiții fragmentul de program [6.3.2.3.a] reprezintă **traversarea în sens direct** a listei înlănțuite
- Fragmentul [6.3.2.3.b] reprezintă **traversarea** unei liste înlănțuite în **sens invers**.

---

**{Traversarea unei liste înlănțuite}**

[6.3.2.3.a]

```

curent:= p;                               {performanța O(n)}
WHILE curent<>nil DO
    BEGIN
        Prelucrare(curent^);
        curent:= curent^.urm
    END;
/*traversarea unei liste înlănțuite*/
/*[6.3.2.3.a]*/

curent= inceput;
while (curent!=NULL){                    /*performanța O(n)*/
    prelucrare(*curent);
    curent= curent->urm;
}
/*-----*/

```

**{Traversarea unei liste înlănțuite în sens invers  
(variantă recursivă)}**

```
PROCEDURE TraversareInversa(curent: TipLista);
BEGIN
  IF curent<>nil THEN [6.3.2.3.b]
  BEGIN
    TraversareInversa(curent^.urm); {performanța O(n)}
    Prelucrare(curent^)
  END
END; {TraversareInversa}
```

-----  
**/\*traversarea unei liste înlănțuite în sens invers  
(varianta recursivă\*/**

```
void traversare_inversa(tip_lista curent)
/*performanța O(n)*/
{
  if (curent!=NULL) /*[6.3.2.3.b]*/
  {
    traversare_inversa(curent->urm);
    prelucrare(*curent);
  }
} /*traversare_inversa*/
/*-----*/
```

- O operație care apare frecvent în practică, este **căutarea** adică **depistarea** unui nod care are cheie egală cu o valoare dată x [6.3.2.3.c].
  - **Căutarea** este de fapt o **traversare cu caracter special** a unei liste.

-----  
**{Căutarea unui nod cu o cheie precizată x (varianta 1)}**  
[6.3.2.3.c]

```
curent:= inceput;
WHILE (curent<>nil) AND (^cheie<>x) DO
  curent:= curent^.urm;
IF curent<>nil THEN {nodul căutat este indicat de
                    curent}
-----
/*cautarea unui nod cu o cheie precizată x (varianta 1)*/
/*[6.3.2.3.c]*/
curent= inceput;
while ((curent!=NULL) && (curent->cheie!=x))
  curent= curent->urm;
if (curent!=NULL) /*nodul căutat este indicat de
                  curent*/
/*-----*/
```

- Dacă acest fragment se termină cu **curent = nil**, atunci **nu** s-a găsit nici un nod cu cheia x, altfel nodul indicat de curent este primul nod având această cheie.
- În legătură cu acest fragment de program trebuie subliniat faptul că există suspiciunea ca în unele compilatoare să fie considerat **incorect**.

- Într-adevăr la evaluarea expresiei booleene din cadrul instrucțiunii **WHILE**, dacă lista **nu** conține nici un nod cu cheia **x**, atunci în momentul în care **curent** devine **nil**, nodul indicat de **curent** **nu** există.
- În consecință, funcție de implementare, se semnalează eroare, deși expresia booleană completă este perfect determinată, ea fiind falsă din cauza primei subexpresii.
- Varianta propusă de secvența [6.3.2.3.d] a operației de căutare este **corectă** în toate cazurile, ea utilizând o **variabilă booleană ajutătoare** notată cu **gasit**.

---

**{Căutarea unui nod cu o cheie precizată x (varianta 2)}**

```
gasit:= false;
curent:= inceput;
WHILE (curent<>nil) AND NOT gasit DO
  IF curent^.cheie=x THEN
    gasit:= true
  ELSE                                     [6.3.2.3.d]
    curent:= curent^.urm;
IF gasit=true THEN {nodul căutat este indicat de
                    curent}
```

---

**/\*cautarea unui nod cu o cheie precizată x (varianta 2)\*/**

```
gasit= false;
curent= inceput;
while ((curent!=NULL)&&~gasit)
  if (curent->cheie==x)
    gasit= true;
  else                                     /*[6.3.2.3.d]*/
    curent= curent->urm;
if (gasit==true) ; /*nodul căutat este indicat de
                    curent*/
/*-----*/
```

- Dacă la terminarea acestui fragment de program **gasit=true** atunci **curent** indică nodul căutat. În caz contrar nu există un astfel de nod și **curent=nil**.
- Pornind de la cele prezentate în acest subparagraf, se pot concepe cu ușurință funcțiile și procedurile care materializează **operatorii** aplicabili listelor implementate cu ajutorul pointerilor atât în varianta restrânsă cât și în varianta extinsă.

### 6.3.3. Implementarea listelor cu ajutorul cursorilor. Gestionarea dinamică a memoriei

- În anumite limbaje de programare ca și FORTRAN sau ALGOL **nu** există definit tipul pointer.

- De asemenea, în anumite situații este mai avantajos pentru programator din punctul de vedere al **performanței codului implementat** să **evite** utilizarea **pointerilor**.
- În astfel de cazuri, **pointerii** pot fi simulați cu ajutorul **cursorilor** care sunt valori întregi ce indică **poziții** în **tablouri**.
- În cadrul acestui paragraf va fi abordată **implementarea listelor înlănțuite** cu ajutorul **cursorilor**, scop în care se definesc structurile de date [6.3.3.a].
- În accepțiunea acestei implementări:
  - (1) Pentru toate listele ale căror noduri sunt de TipNod, se crează un tablou (Zona) având elementele de TipElement.
  - (2) Fiecare element al tabloului conține un câmp `nodLista:TipNod` și un câmp `urm:TipCursor` definit ca **subdomeniu** al tipului întreg.
  - (3) `TipLista` este în aceste condiții identic cu `TipCursor`, orice listă fiind **precizată** de fapt de un **cursor** în tabloul Zona.

-----  
**{Implementarea listelor cu ajutorul cursorilor - structuri de date}**

```

TYPE TipCursor = 0..LungMax;
      TipNod=...
      TipElement = RECORD
        nodLista: TipNod;
        urm: TipCursor
      END;

```

[6.3.3.a]

```

      TipLista = TipCursor;

```

```

VAR Zona: ARRAY[1..LungMax] OF TipElement;
      L,M,disponibil: TipLista;

```

-----  
**/\*implementarea listelor cu ajutorul cursorilor\*/**

```

enum { lung_max = 10};
typedef unsigned char tip_cursor;

```

```

typedef unsigned boolean;
#define true (1)
#define false (0)

```

```

typedef int tip_nod;

```

```

typedef struct tip_element {
    tip_nod nod_lista;
    tip_cursor urm;
} tip_element;

```

/\*[6.3.3.a]\*/

```

typedef tip_cursor tip_lista;

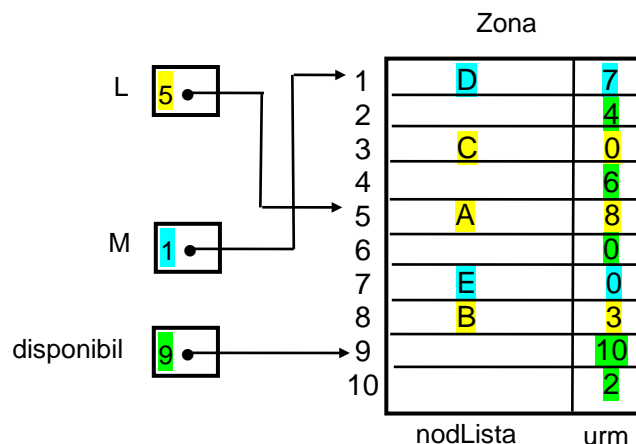
```

```

tip_element zona[lung_max];
tip_lista l,m,disponibil;
/*-----*/

```

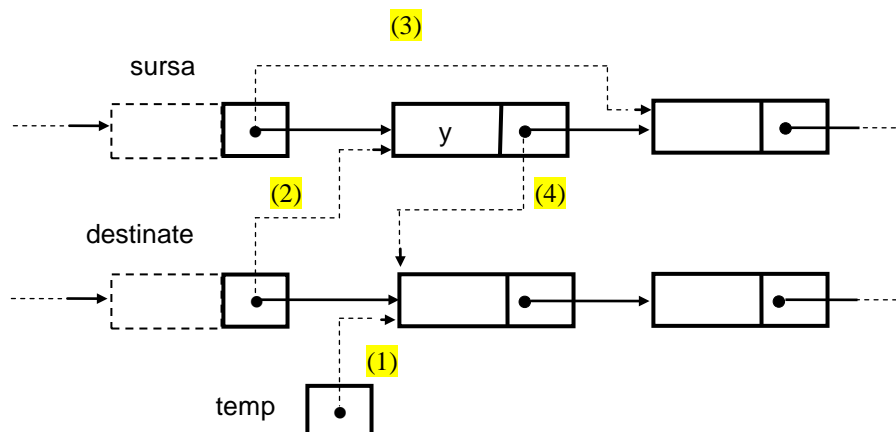
- Spre exemplu dacă  $L:TipLista$  este un cursor care indică începutul unei liste atunci:
  - Valoarea lui  $Zona[L].nodLista$  reprezintă **primul** nod al listei  $L$ .
  - $Zona[L].urm$  este indexul (cursorul) care indică cel de-al doilea element al listei  $L$ , ș.a.m.d.
  - Valoarea zero a unui cursor, semnifică legătura vidă, adică faptul că nu urmează nici un element.
- În figura 6.3.3.a s-au reprezentat două liste  $L = A, B, C$  și  $M = D, E$  care partajează tabloul  $Zona$  cu lungimea maximă 10.
- Se observă că:
  - Acele locații ale tabloului care **nu** apar în nici una din cele două liste sunt înălțuite într-o altă listă numită *disponibil*.
  - Lista *disponibil* este utilizată fie:
    - Pentru a **furniza o nouă locație** în vederea realizării unei **inserții**.
    - Pentru a **depozita o locație** a tabloului rezultată din **suprimarea** unui nod în vederea unei reutilizări ulterioare.



**Fig.6.3.3.a.** Implementarea listelor înălțuite cu ajutorul cursorilor

- Avem de fapt de-a face cu o **gestionare dinamică a zonei de memorie** alocată listelor (tabloul *Zona*) realizată de către programator.

- Astfel, pentru a **insera** un nod  $x$  în lista  $L$ :
  - (1) Se suprimă prima locație a listei disponibil.
  - (2) Se inserează această locație în poziția dorită a listei  $L$ , actualizând valorile cursorilor implicați.
  - (3) Se asignează câmpul `nodLista` al acestei locații cu valoarea lui  $x$ .
- **Suprimarea** unui element  $x$  din lista  $L$  presupune:
  - (1) Suprimarea locației care îl conține pe  $x$  din lista  $L$ .
  - (2) Inserția acestei locații în capul listei disponibil.
- Atât **inserția** cât și **suprimarea** sunt de fapt cazuri speciale ale următoarei **situații**:
  - Fie două liste precizate prin cursorii *sursa* și *destinație*.
  - Fie  $y$  prima locație a listei indicate de *sursa*.
  - Se **suprimă**  $y$  din lista indicată de *sursa* și se **înlănțuie** pe prima poziție a listei indicate de *destinație*. Acest lucru se realizează astfel:
    - (1) Se salvează valoarea cursorului *destinație* în locația auxiliară *temp*.
    - (2) Se atribuie valoarea cursorului *sursa* cursorului *destinație* care astfel îl va indica pe  $y$ .
    - (3) Se atribuie lui *sursa* valoarea legăturii lui  $y$ .
    - (4) Legătura lui  $y$  se asignează cu fosta valoare a lui *destinație*.
  - Reprezentarea grafică a acestei acțiuni apare în fig.6.3.3.b unde sunt prezentate legăturile înainte (linie continuă) și după (linie întreruptă) desfășurarea ei.



**Fig.6.3.3.b.** Mutarea unui nod dintr-o listă înlănțuită (sursă) în altă listă înlănțuită (destinație)

- În [6.3.3.b] apare codul Pascal respectiv C, care implementează funcția Muta.

-----  
{Liste înlănțuite implementate cu ajutorul cursorilor -  
Operatorul de gestionare dinamică a memoriei}

```
FUNCTION Muta(VAR sursa,destinatie:TipCursor):boolean;
    {mută elementul indicat de sursă în fața celui
     indicat de destinație}
VAR temp: TipCursor;
BEGIN
    IF sursa=0 THEN                                {performanța O(1)}
        BEGIN
            mesaj('locația nu există');
            Muta:= false
        END
    ELSE                                            [6.3.3.b]
        BEGIN
            temp:= destinatie;
            destinatie:= sursa;
            sursa:= zona[destinatie].urm;
            zona[destinatie].urm:= temp;
            Muta:= true
        END
    END; {Muta}
```

-----  
/\* Liste înlănțuite implementate cu ajutorul cursorilor -  
Operatorul de gestionare dinamică a memoriei \*/

```
boolean muta(tip_cursor* sursa,tip_cursor* destinatie)
    /*mută elementul indicat de sursă în fața celui
     indicat de destinație*/
    {
        tip_cursor temp;
        boolean muta_result;
        if (*sursa==0)                                /*performanța O(1)*/
            {
                printf("locația nu există");
                muta_result= false;
            }
        else                                            /*[6.3.3.b]*/
            {
                temp= *destinatie;
                *destinatie=*sursa;
                *sursa= zona[*destinatie-1].urm;
                zona[*destinatie-1].urm= temp;
                muta_result= true;
            }
        return muta_result;
    }    /*muta*/
/*-----*/
```

- Pentru exemplificarea utilizării **operatorului de gestionare dinamică a memoriei** *Muta*, în secvența [6.3.3.c] se prezintă:
  - Implementarea operatorilor *Insereaza* și *Suprima*.
  - Operatorul *Init* care înlănțuie inițial toate elementele tabloului Zona în lista indicată de disponibil.
    - Procedurile **omit** verificarea erorilor.
    - Se presupune existența funcției *Muta*.
    - Se precizează că variabila inceput indică **începutul listei curente**.

---

*{Liste înlănțuite implementate cu ajutorul cursorilor -  
Operatorii Insereaza, Suprima și Init}*

```
PROCEDURE Insereaza(x: TipNod; p: TipPozitie;
                    VAR inceput: TipPozitie);
BEGIN
    {performanța O(1)}
    IF p=0 THEN
        BEGIN {se inserează pe prima poziție}
            IF Muta(disponibil,inceput) THEN
                zona[inceput].nodLista:= x      [6.3.3.c]
            END
        ELSE {se inserează într-o poziție diferită de
              prima}
            IF Muta(disponibil,zona[p].urm) THEN
                {locația pentru x va fi indicată de către
                 zona[p].urm}
                zona[zona[p].urm].nodLista:= x
            END; {Insereaza}
```

```
PROCEDURE Suprima(p:TipPozitie;VAR inceput:TipPozitie);
BEGIN
    IF p=0 THEN {performanța O(1)}
        Muta(inceput,disponibil)
    ELSE
        Muta(zona[p].urm,disponibil)
    END; {Suprima}
```

```
PROCEDURE Init;
    {înlănțuie inițial toate elementele zonei în lista de
     disponibili}
    VAR i: TipCursor;
    BEGIN {performanța O(n)}
        FOR i:=lungMax-1 DOWNTO 1 DO
            zona[i].urm:= i+1;
            disponibil:= 1; zona[lungMax].urm:= 0
        END; {Init}
```

---

*/\* Liste înlănțuite implementate cu ajutorul cursorilor -  
Operatorii Insereaza, Suprima și Init \*/*



```

void insereaza(tip_nod x, tip_cursor p,
               tip_cursor* inceput)
{
    /*performanța O(1)*/
    if (p==0){ /*se inserează pe prima poziție*/
        if (muta(&disponibil,inceput))
            zona[*inceput-1].nodlista= x; /*[6.3.3.c]*/
    }
    else /*se inserează într-o poziție diferită de
          prima*/
        if (muta(&disponibil,&zona[p-1].urm))
            /*locația pentru x va fi indicată de către
              zona[p].urm*/
            zona[zona[p-1].urm-1].nodlista= x;
    } /*insereaza*/

void suprima(tip_cursor p,tip_cursor* inceput)
{
    if (p==0) /*performanța O(1)*/
        muta(inceput,&disponibil);
    else
        muta(&zona[p-1].urm,&disponibil);
    } /*suprima*/

void init()
    /*inițializează elementele zonei înlănțuindu-le în
      lista de disponibili*/
    {
        tip_cursor i; /*performanța O(n)*/
        for( i=lung_max-1; i >= 1; i --)
            zona[i-1].urm= i+1;
        disponibil= 1; zona[lung_max-1].urm= 0;
    } /*init*/
/*-----*/

```

#### 6.3.4. Implementarea listelor cu ajutorul referințelor

- În limbajele orientate pe obiecte care **nu** definesc **tipul pointer**, implementarea structurilor de date recursive în general și a listelor înlănțuite în mod special se poate realiza foarte elegant cu ajutorul **referințelor**.
- Astfel, pentru a implementa o listă înlănțuită în limbajul JAVA, ca punct de pornire se poate defini **clasa** **Nod**, care specifică formatul obiectelor asociate nodurilor listei [6.3.4.a].

```

-----
class Nod {
    private Object element; /*elementul memorat în nodul
                             curent
    private Nod urm; /*referința la nodul următor al listei

    // constructori
    Nod() {

```

[6.3.4.a]

```

        //crează un nod cu un element nul și cu o referință
        nulă
        this(null,null);
    }

    public Nod(Object e, Nod n) {
        //crează un nod cu un anumit element e și o anumită
        referință urm
        element = e;
        urm = n;
    }

    //metode de actualizare
    void setElement(Object elemNou) {
        element = elemNou;
    }

    void setUrm(Nod urmNou) {
        urm = urmNou
    }

    //metode de acces
    Object getElement() {
        return element;
    }
    Node getUrm() {
        return urm;
    }
}

```

---

- În continuare pornind de la clasa Nod se poate defini o **clasă** ListaInlantuita care:
    - Păstrează o referință la nodul de început al listei.
    - În mod opțional pot păstra și alte informații referitoare la listă cum ar fi o referință la **ultimul** element al listei și/sau **numărul** de noduri.
  - În secvența [6.3.4.b] apare un fragment dintr-o astfel de clasă în care se prezintă structura de date listă ca atare precum și unele dintre metodele care implementează operatorii specifici.
- 

```

public class ListaInlantuita implements Lista {
    private Nod inceput;    //referință la începutul listei
    private int dimensiune; //numărul de elemente ale listei

    public ListaInlantuita() {    //Inițializează lista
        inceput = null;
        dimensiune = 0;
    }

    public int dimensiune() {    //Returnează dimensiunea
                                curentă

```

```

        return dimensiune;
    }

    public boolean listaVida() { //Returnează true dacă
                                lista este vidă

        if (inceput == null)
            return true;
        return false;
    }

    public void InserInceput(Object elem){ //Insertie la
                                           începutul listei

        Nod n = new Nod();
        n.setElement(elem);
        n.setUrm(inceput);
        inceput = n;
        dimensiune++;
    }

    public Object SuprimaPrimul() { //Suprimarea
                                    elementului de la începutul listei

        Object obj;
        if (inceput.listaVida())
            //se tratează excepția în mod specific;
        obj = inceput.getElement();
        inceput = inceput.getUrm();
        dimensiune--;
        return obj;
    }

    //alte metode ...
}
-----

```

### 6.3.5. Comparație între metodele de implementare a listelor

- Este greu de precizat care dintre metodele de implementare a listelor este mai bună, deoarece răspunsul depinde de:
  - (1) Limbajul de programare utilizat.
  - (2) Operațiile care se doresc a fi realizate.
  - (3) Frecvența cu care sunt invocați operatorii.
  - (4) Constrângerile de timp de acces, de memorie, de performanță, ș.a.
- În orice caz se pot formula următoarele **observații**:
  - (1) **Implementarea bazată pe tablouri sau pe cursori** necesită specificarea **dimensiunii maxime** a listei în momentul compilării.

- Dacă **nu** se poate determina o astfel de **limită superioară a dimensiunii listei** se recomandă **implementarea bazată pe pointeri sau referințe**.
- (2) **Aciași operatori** pot avea performanțe diferite în implementări diferite.
  - Spre exemplu inserția sau suprimarea unui nod precizat au o durată constantă la **implementarea înlănțuită** ( $O(1)$ ), dar necesită o perioadă de timp proporțională cu numărul de noduri care urmează nodului în cauză în **implementarea bazată pe tablouri** ( $O(n)$ ).
  - În schimb operatorul **Anterior** necesită un timp constant în **implementarea prin tablouri** ( $O(1)$ ) și un timp care depinde de lungimea totală, respectiv de poziția nodului în **implementarea bazată pe pointeri, referințe sau cursori** ( $O(n)$ ).
- (3) **Implementarea bazată pe tablouri sau pe cursori** poate fi ineficientă din punctul de vedere al **utilizării memoriei**, deoarece ea ocupă tot timpul spațiul maxim solicitat, indiferent de dimensiunea reală a listei la un moment dat.
- (4) **Implementarea înlănțuită** utilizează în fiecare moment spațiul de memorie strict necesar lungimii curente a listei, dar necesită în plus spațiul pentru înlănțuire în cadrul fiecărui nod.
- În funcție de circumstanțe una sau alta dintre implementări poate fi mai mult sau mai puțin avantajoasă.

## 6.4. Aplicații ale listelor înlănțuite

### 6.4.1. Problema concordanței

- **Formularea problemei:**
  - Se dă un **text** format dintr-o succesiune de **cuvinte**.
  - Se baleează textul și **se depistează cuvintele**.
  - Pentru fiecare cuvânt se verifică dacă este sau nu la **prima apariție**:
    - În caz că este la prima apariție, **cuvântul se înregistrează**.
    - În caz că el a mai fost găsit, se incrementează un contor asociat cuvântului care memorează **numărul de apariții**.
  - În final se dispune de **toate cuvintele distincte** din text și de **numărul de apariții** al fiecăruia.

- Se menționează că această problemă este importantă, deoarece ea reflectă într-o formă simplificată una din activitățile pe care le realizează un **compilator** și anume **construcția și exploatarea listei identificatorilor**.
- Programul **Concordanta** rezolvă această problemă utilizând drept suport o listă înlănțuită simplă, în următoarea manieră [6.4.1.a]:
  - Construiește o listă înlănțuită conținând cuvintele distincte ale unui text sursă.
  - Inițial lista este vidă, ea urmând a fi completată pe parcursul parcurgerii textului.
  - Procesul de căutare în listă împreună cu inserția sau incrementarea contorului este realizat de procedura **Cauta**.
  - Pentru simplificare, se presupune că "textul" este de fapt o succesiune de numere întregi pozitive care reprezintă "cuvintele".
  - Cuvintele se citesc de la tastatură ele terminându-se cu un cuvânt fictiv, în cazul de față numărul zero care precizează sfârșitul textului.
  - Căutarea în listă se face conform celor descrise în paragraful &6.3.2.3 cu deosebirea că variabila gasit s-a înlocuit cu negata ei.
  - Variabila pointer inceput, indică tot timpul începutul listei.
  - Se precizează faptul că inserările se fac la începutul listei iar procedura **Tiparire** reprezintă un exemplu de traversare a unei liste în sensul celor precizate anterior.

-----  
**{Concordanța - varianta Pascal}**

**PROGRAM** Concordanta;

```

TYPE TipReferinta = ^TipNod;                                [6.4.1.a]
      TipNod = RECORD
        cheie: integer;
        numar: integer;
        urmator: TipReferinta
      END;

```

```

VAR cuv: integer; inceput: TipReferinta; {*}

```

```

PROCEDURE Cauta(x:integer; VAR inceput:TipReferinta);
  VAR q: TipReferinta; negasit: boolean;
  BEGIN
    q:= inceput;
    negasit:= true;
    WHILE (q<>NIL) AND negasit DO
      IF q^.cheie=x THEN negasit:= false
      ELSE q:= q^.urmator;
    IF negasit THEN {nu s-a găsit, deci inserție}

```

```

        BEGIN
            q:= inceput;
            new(inceput);
            WITH inceput^ DO
                BEGIN
                    cheie:= x;
                    numar:= 1;
                    urmator:= q
                END
            END
        ELSE {s-a găsit, deci incrementare}
            q^.numar:= q^.numar+1
        END; {Cauta}

```

```

PROCEDURE Tiparire(q: TipReferinta);
VAR r: TipReferinta;
BEGIN
    r:= q;
    WHILE r<>NIL DO
        BEGIN
            WriteLn(r^.cheie,r^.numar);
            r:= r^.urmator
        END
    END; {Tiparire}

```

```

BEGIN
    inceput:= NIL;  {**}
    Read(cuv);
    WHILE cuv<>0 DO
        BEGIN
            Cauta(cuv,inceput);
            Read(cuv)
        END;
    Tiparire(inceput)
END.

```

---

**/\*Concordanța - varianta C\*/**

```

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

```

```

typedef unsigned boolean;
#define true (1)
#define false (0)

```

```

typedef struct tip_nod* tip_referinta;          /*[6.4.1.a]*/

```

```

typedef struct {
    int cheie;
    int numar;
    tip_referinta urmator;
}tip_nod;

```

```

int cuv;

```

```

tip_referinta inceput;          /**/

void cauta(int x, tip_referinta* inceput)
{
    tip_referinta q;
    boolean negasit;
    q= *inceput;
    negasit= true;
    while ((q!=NULL) && negasit)
        if (((tip_nod*)q)->cheie==x)
            negasit= false;
        else
            q= ((tip_nod*)q)->urmator;
    if (negasit){          /*nu s-a găsit, deci inserție*/
        q= *inceput;
        *inceput = (tip_nod*)malloc(sizeof(tip_nod));
        ((tip_nod*)(*inceput))->cheie= x;
        ((tip_nod*)(*inceput))->numar= 1;
        ((tip_nod*)(*inceput))->urmator= q;
    }
    else /*s-a găsit, deci incrementare*/
        ((tip_nod*)q)->numar= ((tip_nod*)q)->numar+1;
} /*cauta*/

void tiparire(tip_referinta q)
{
    tip_referinta r;
    r= q;
    while (r!=NULL)
    {
        printf("%i%i\n",((tip_nod*)r)->cheie,
            ((tip_nod*)r)->numar);
        r= ((tip_nod*)r)->urmator;
    }
} /*tiparire*/

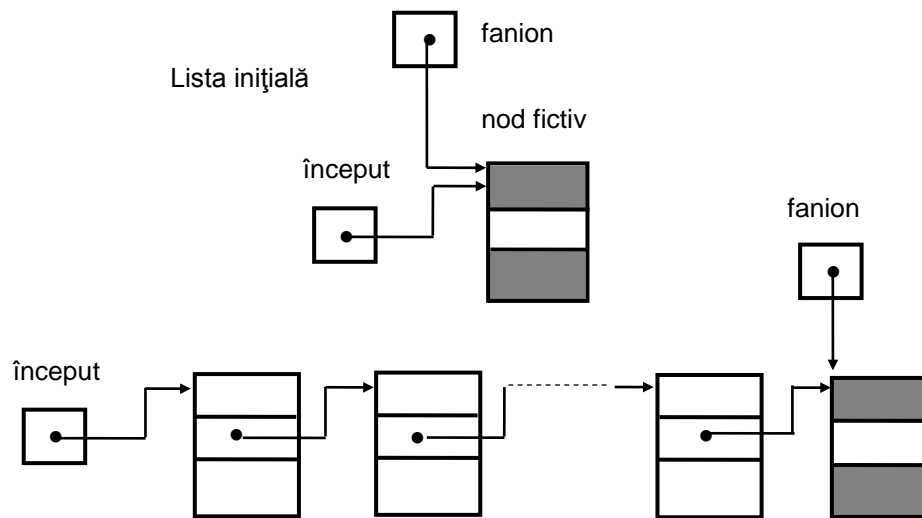
int main(int argc, const char* argv[])
{
    inceput= NULL;          /**/
    scanf("%i", &cuv);
    while (cuv!=0)
    {
        cauta(cuv,&inceput);
        scanf("%i", &cuv);
    }
    tiparire(inceput);
    return 0;
}

/*-----*/

```

- În continuare se descrie o optimizare a procedurii de căutare prin utilizarea "**metodei fanionului**".

- În acest scop, lista cuvintelor întâlnite se prelungește cu un **nod suplimentar** numit fanion (fig 6.4.1.a).



**Fig.6.4.1.a.** Implementarea unei liste înlănțuite utilizând tehnica nodului fanion

- Tehnica de căutare este similară celei utilizate în cazul tablourilor liniare (&1.4.2.1).
- Pentru aplicarea procedurii de căutare optimizate **Cauta1**, în programul **Concordanta** [6.4.1.a] trebuiesc efectuate două **modificări**:
  - La declararea variabilelor, în locul indicat prin [\*], se adaugă declararea nodului fanion `fanion: tipReferinta;`
  - Se modifică inițializarea listei de cuvinte, indicată în cadrul programului prin [\*\*], respectiv instrucțiunea `inceput := nil` se înlocuiește cu secvența `new(inceput), fanion := inceput`. Prin aceasta lista de cuvinte conține de la bun început un nod (cel fictiv).
- În aceste condiții, procedura **Cauta1** apare în secvența [6.4.1.b].
  - Față de varianta [6.4.1.a] condiția din cadrul instrucției **WHILE** este mai simplă, realizându-se un câștig simțitor de timp.
  - Desigur trebuie modificată și condiția de test din procedura **Tiparire** astfel încât să reflecte noua situație.

---

**{Căutare în liste înlănțuite utilizând metoda fanionului}**

```
PROCEDURE Cauta1(x:integer, VAR inceput:TipReferinta);
  VAR q: TipReferinta;
  BEGIN
    q:= inceput;
    fanion^.cheie:= x;
```



```

WHILE q^.cheie<>x DO q:= q^.urmator;
IF q=fanion THEN {elementul nu s-a găsit}
    BEGIN
        q:= inceput;
        new(inceput);
        WITH inceput^ DO
            BEGIN
                cheie:= x;
                numar:= 1;
                urmator:= q
            END
        END
    ELSE {s-a găsit}
        q^.numar:= q^.numar+1
END; {Cautal}
-----
/*Căutare în liste înlănțuite utilizând metoda fanionului*/

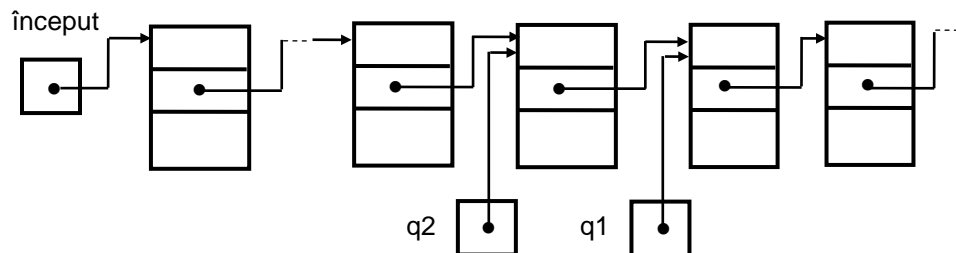
void cautal(int x, tip_referinta* inceput)
{
    tip_referinta q;
    q= *inceput;
    fanion->cheie= x;
    while (((tip_nod*)q)->cheie!=x)          /*[6.4.1.b]*/
        q= ((tip_nod*)q)->urmator;
    if (q==fanion){ /*elementul nu s-a găsit*/
        q= *inceput;
        *inceput = (tip_nod*)malloc(sizeof(tip_nod))
        ((tip_nod*)(*inceput))->cheie= x;
        ((tip_nod*)(*inceput))->numar= 1;
        ((tip_nod*)(*inceput))->urmator= q;
    }
    else /*s-a găsit*/
        ((tip_nod*)q)->numar= ((tip_nod*)q)->numar+1;
} /*cautal*/
/*-----*/

```

#### 6.4.2. Crearea unei liste ordonate. Tehnica celor doi pointeri

- În cadrul acestui paragraf se abordează problema creării unei liste astfel încât ea să fie mereu **ordonată** după **chei crescătoare**.
  - Cu alte cuvinte, odată cu crearea listei, aceasta se și sortează.
- În contextul problemei concordanței, acest lucru se realizează simplu deoarece înainte de inserția unui nod, acesta trebuie oricum **căutat** în listă.
  - (1) Dacă **lista** este **sortată**, atunci căutarea se va termina cu prima cheie mai mare decât cea căutată, apoi în continuare se inserează nodul în poziția care **precede** această cheie.

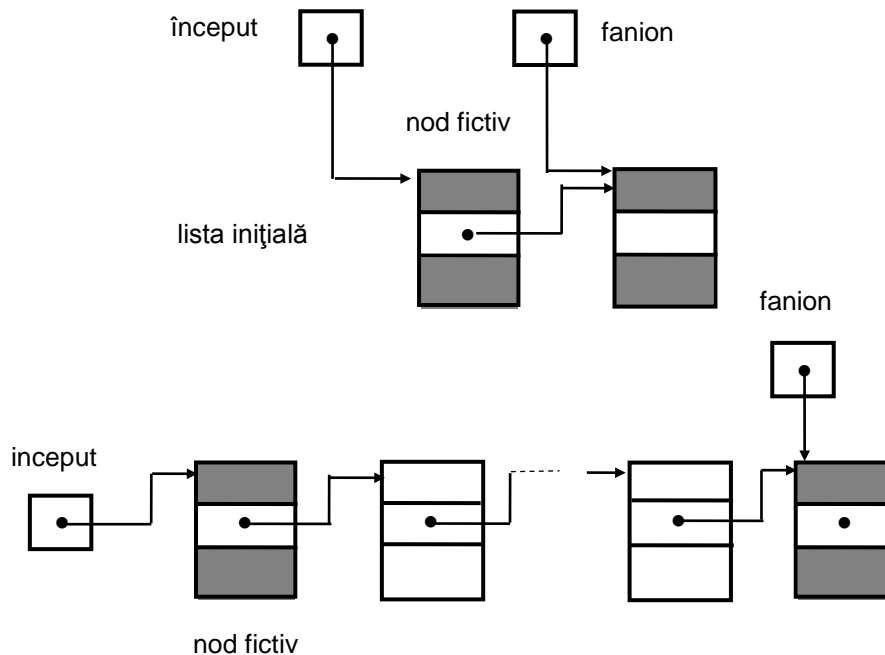
- (2) În cazul unei **liste nesortate**, căutarea înseamnă parcurgerea întregii liste, după care nodul se inserează la **începutul listei**.
- După cum se vede, procedeul (1) nu numai că permite obținerea listei sortate, dar procesul de căutare devine mai eficient.
  - Este important de observat faptul că la crearea unor **structuri tablou** sau **secvență nu** există posibilitatea simplă de a le obține gata sortate.
  - În schimb la **listele liniare sortate nu există** echivalentul unor **metode de căutare avansate** (spre exemplu căutarea binară) care sunt foarte eficiente la tablourile sortate.
- **Insertia** unui nod într-o **listă sortată** presupune inserția unui nod **înaintea** celui indicat de pointerul cu care s-a realizat căutarea.
  - O modalitate de rezolvare a unei astfel de situații a fost prezentată în paragraful [6.3.2.1].
- În continuare se va descrie o altă tehnică de inserție bazată pe utilizarea a **doi pointeri**  $q_1$  și  $q_2$ , care indică tot timpul **două noduri consecutive ale listei**, conform figurii 6.4.2.a.



**Fig.6.4.2.a.** Traversarea unei liste înlănțuite utilizând doi pointeri.

- Se presupune că lista este explorată utilizând metoda nodului fanion. Cheia de inserat  $x$  se introduce inițial în nodul fanion.
- Cei doi pointeri avansează simultan de-a lungul listei până când cheia nodului indicat de  $q_1$  devine mai mare sau egală cu cheia de inserat  $x$ .
  - Acest lucru ce se va întâmpla cu certitudine, cel mai târziu în momentul în care  $q_1$  devine egal cu fanionul.
  - Dacă în acest moment, cheia nodului indicat de  $q_1$  este strict mai mare decât  $x$  sau  $q_1 = \text{fanion}$ , atunci trebuie inserat un nou nod în listă între nodurile indicate de  $q_2$  și  $q_1$ .

- În caz contrar, s-a găsit cheia căutată și trebuie incrementat contorul `q1^.numar`.
- În implementarea acestui proces se va ține cont de faptul că, funcționarea sa corectă presupune existența inițial în listă a **cel puțin două noduri**, deci cel puțin un nod în afară de cel indicat de fanion.
- Din acest motiv lista se va implementa utilizând tehnica celor **două noduri fictive** (fig.6.4.2.b).



**Fig.6.4.2.b.** Implementarea unei liste utilizând tehnica celor două noduri fictive.

- În vederea realizării acestor deziderate programul **Concordanta** (secvența [6.4.1.a]) trebuie modificat după cum urmează:
  - (1) Se adaugă la partea de declarație a variabilelor, (locul indicat cu [\*]), declarația `fanion: TipRef`.
  - (2) Se înlocuiește instrucția `inceput:=nil` (indicată prin [\*\*]) cu următoarea secvență de instrucțiuni care crează lista vidă specifică implementării prin **tehnica celor doi pointeri** [6.4.2.a]:

---

```
new(inceput); new(fanion);                                [6.4.2.a]
inceput^.urmator:= fanion;
```

---

- (3) Se înlocuiește procedura **Cauta** cu **Cauta2** secvența [6.4.2.b].
-

**{Căutare în liste înlănțuite utilizând tehnica celor doi pointeri}**

```
PROCEDURE Cauta2(x: integer; inceput: TipReferinta);
VAR q1,q2,q3: TipReferinta;
BEGIN
    q2:= inceput;
    q1:= q2^.urmator;
    fanion^.cheie:= x;
    WHILE q1^.cheie<x DO
        BEGIN
            q2:= q1;                                [6.4.2.b]
            q1:= q2^.urmator
        END;
    IF (q1^.cheie=x) AND (q1<>fanion) THEN
        q1^.numar:= q1^.numar+1
    ELSE
        BEGIN {se creează un nou nod indicat de q3 și
                se inserează între q2^ și q1^}
            new(q3);
            WITH q3^ DO
                BEGIN
                    cheie:= x;
                    numar:= 1;
                    urmator:= q1
                END;
            q2^.urmator:= q3
        END
    END; {Cauta2}
```

---

**/\*Cautare în liste înlănțuite utilizând tehnica celor doi pointeri\*/**

```
void cauta2(int x, tip_referinta inceput)
{
    tip_referinta q1,q2,q3;
    q2= inceput;
    q1= ((tip_nod*)q2)->urmator;
    fanion->cheie= x;
    while (q1->cheie<x)
    {
        q2= q1;                                /*[6.4.2.b]*/
        q1= ((tip_nod*)q2)->urmator;
    }
    if (((tip_nod*)q1)->cheie==x) && (q1!=fanion))
        ((tip_nod*)q1)->numar= ((tipnod*)q1)->numar+1;
    else
    {
        /*se creează un nou nod indicat de q3 și
           se inserează între q2^ si q1^*/
        q3 = (tip_nod*)malloc(sizeof(tip_nod));
        ((tip_nod*)q3)->cheie= x;
        ((tip_nod*)q3)->numar= 1;
        ((tip_nod*)q3)->urmator= q1;
        ((tip_nod*)q2)->urmator= q3;
    }
}
```

```
} /*cauta2*/  
/*-----*/
```

- Aceste modificări conduc la crearea listei **sortate**.
- Trebuie însă observat faptul că **beneficiul** obținut în urma **sortării** este destul de **limitat**.
  - El se manifestă **numai** în cazul **căutării unui nod** care **nu se găsește în listă**.
    - Această operație necesită parcurgerea în medie a unei **jumătăți** de listă în cazul listelor **sortate** în comparație cu parcurgerea **întregii** liste dacă aceasta **nu** este **sortată**.
  - La căutarea unui nod care **se găsește în listă** se parcurge în medie **jumătate de listă** indiferent de faptul că **lista este sortată sau nu**.
    - Această concluzie este valabilă dacă se presupune că succesiunea cheilor sortate este un șir de variabile aleatoare cu distribuții identice.
- În definitiv, cu toate că **sortarea listei** practic **nu** costă nimic, se recomandă a fi utilizată numai în cazul unor texte cu multe cuvinte distincte în care același cuvânt se repetă de puține ori.

#### 6.4.3. Căutarea în liste cu reordonare

- Utilizarea structurii de date listă liniară este deosebit de avantajoasă în activitatea de compilare la crearea și exploatarea **listei identificatorilor**.
  - În general la parcurgerea unui text sursă, compilatorul inserează în listă fiecare identificator declarat împreună cu o serie de informații necesare procesului de compilare.
  - Aparițiile ulterioare ale unor astfel de identificatori, presupun căutarea lor în listă, în vederea obținerii informațiilor necesare în procesul de generare a codului.
  - La părăsirea domeniului lor de existență, identificatorii sunt suprimați din lista identificatorilor.
- În continuare, pornind de la contextul mai sus precizat, se va prezenta o altă **tehnică de căutare într-o listă înlănțuită** utilizabilă cu precădere la crearea și **exploatarea listei identificatorilor** de către un compilator.
- Studiindu-se un număr mare de programe sursă considerate tipice, s-a făcut următoarea **constatare experimentală**:
  - Aparițiile unui identificator oarecare în textul sursă al unui program au tendința de a se "**îngrămădi**" în anumite locuri ale programului, în timp ce în restul textului, apariția aceluiași identificator se produce cu o probabilitate mult mai redusă.

- Acesta este așa numitul **principiu al localizării**.
- Conform acestui principiu, apariția unui identificator oarecare, poate fi urmată curând, cu mare probabilitate, de una sau mai multe reapariții.
- Pornind de la această constatare se poate concepe o **metodă** de construire a **listei identificatorilor** care să conducă la o ameliorare substanțială a procesului de căutare.
- Metoda, denumită "**căutare în listă cu reordonare**", constă în aceea că ori de câte ori un identificator se caută și se găsește în listă, el se "**mută**" la începutul listei, astfel încât la proxima apariție el va fi găsit imediat.
  - Cu alte cuvinte, lista se **reordonează** după fiecare căutare finalizată cu găsirea nodului.
  - Dacă un nod **nu** este găsit în listă, el se inserează la începutul acesteia.
- În secvența [6.4.3.a] apare procedura **Cauta3** care implementează această metodă utilizând tehnica celor doi pointeri.
  - Pointerul q1 indică nodul găsit iar pointerul q2 nodul precedent.
  - Structurile de date sunt cele definite în [6.4.1.a].

---

#### **{Căutare în liste cu reordonare}**

```

PROCEDURE Cauta3(x:integer; VAR inceput:TipReferinta);
  VAR q1,q2,q3: TipReferinta;
  BEGIN
    q1:= inceput;
    fanion^.cheie:= x;
    IF q1=fanion THEN {se inserează primul nod}
      BEGIN
        new(inceput);
        WITH inceput^ DO
          BEGIN
            cheie:= x;
            numar:= 1;
            urmator:= fanion
          END
        END
      ELSE
        IF q1^.cheie=x THEN {este pe prima poziție}
          q1^.numar:= q1^.numar+1
        ELSE
          BEGIN {căutarea} [6.4.3.a]
            REPEAT
              q2:= q1;
              q1:= q2^.urmator
            UNTIL q1^.cheie=x;
            IF q1=fanion THEN {insertie}
              BEGIN
                q2:= inceput;

```

```

        new(inceput);
        WITH inceput^ DO
            BEGIN
                cheie:= x;
                numar:= 1;
                urmator:= q2
            END
        END
    ELSE {s-a găsit, deci reordonare}
        BEGIN
            q1^.numar:= q1^.numar+1;
            q2^.urmator:= q1^.urmator;
            q1^.urmator:= inceput;
            inceput:= q1
        END
    END
END; {Cauta3}

```

---

**/\*Căutare în liste cu reordonare\*/**

```

void cauta3(int x, tip_referinta* inceput)
{
    tip_referinta q1,q2,q3;
    q1= *inceput;
    ((tip_nod*)fanion)->cheie= x;
    if (q1==fanion){ /*se inserează primul nod*/
        *inceput = (tip_nod*)malloc(sizeof(tip_nod));
        ((tip_nod*)(*inceput))->cheie= x;
        ((tip_nod*)(*inceput))->numar= 1;
        ((tip_nod*)(*inceput))->urmator= fanion;
    }
    else
        if (((tip_nod*)q1)->cheie==x)/*este pe prima
                                     poziție*/
            ((tip_nod*)q1)->numar= ((tip_nod*)q1)->numar+1;
        else
            /*căutarea*/ /*[6.4.3.a]*/
            {
                do {
                    q2= q1;
                    q1= ((tip_nod*)q2)->urmator;
                } while (!(((tip_nod*)q1)->cheie==x));
                if (q1==fanion) /*insertie*/
                {
                    q2= *inceput;
                    *inceput= (tip_nod*)malloc(sizeof(tip_nod));
                    ((tip_nod*)(*inceput))->cheie= x;
                    ((tip_nod*)(*inceput))->numar= 1;
                    ((tip_nod*)(*inceput))->urmator= q2;
                }
                else /*s-a găsit, deci reordonare*/
                {
                    ((tip_nod*)q1)->numar=
                        ((tip_nod*)q1)->numar+1;
                    ((tip_nod*)q2)->urmator=

```

```

                                ((tip_nod*)q1)->urmator;
                                ((tip_nod*)q1)->urmator= *inceput;
                                *inceput= q1;
                                }
                                }
                                }
                                /*cauta3*/
/*-----*/

```

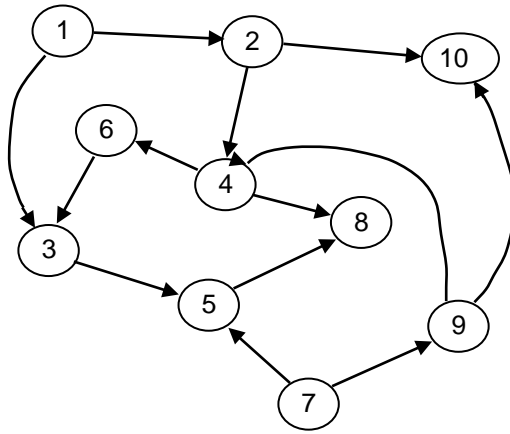
- Se observă că **reordonarea** presupune existența a cel puțin **două noduri în listă**. Acestea pot fi un nod real și nodul fanion.
  - Deoarece în implementarea realizată s-a utilizat **tehnica nodului fanion**, în cadrul procedurii s-a prevăzut în mod explicit o secvență care tratează inserția primului nod al listei.
  - În acest context, inițializarea listei se face înlocuind instrucțiunea notată cu **[\*\*]** în programul [6.4.1.a] cu secvența `new(inceput); fanion:=inceput;`.
- Cercetări empirice în care s-au comparat timpii de rulare ai programului **Concordanta**, utilizând lista sortată (**Cauta2**) respectiv tehnica căutării cu reordonare (**Cauta3**), au pus în evidență un factor de ameliorare în favoarea celei din urmă cuprins între 1.37 și 4.7 [Wi76].
  - Ameliorarea mai pronunțată apare la textele mai lungi.

#### 6.4.4. Sortarea topologică

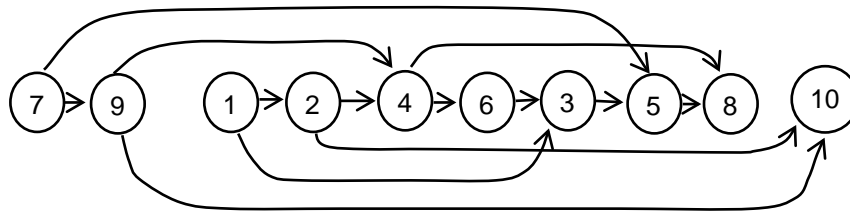
- Un exemplu de utilizare flexibilă a unor structuri de date dinamice este procesul **sortării topologice**.
  - Acesta este un proces de sortare, al unei mulțimi elemente peste care a fost definită o relație de **ordonare parțială**, adică ordonarea este valabilă doar pentru anumite perechi de elemente, **nu** pentru toate.
- Aceasta este de fapt o situație reală ilustrată prin următoarele **exemple**:
  - (1) Într-un **dicționar**, **cuvintele** sunt definite în termenii altor cuvinte.
    - Dacă un cuvânt  $w$  este definit în termenii cuvântului  $v$ , se va nota aceasta prin  $v \prec w$ .
    - **Sortarea topologică** a cuvintelor în **dicționar** înseamnă aranjarea lor într-o astfel de ordine încât să **nu** existe **nici o referință în față** (adică să nu fie utilizat nici un cuvânt înainte ca el să fi fost definit).
  - (2) Un **task** (spre exemplu un proiect ingineresc) poate fi de regulă divizat în **subtaskuri**.



- Terminarea unor subtaskuri trebuie în mod uzual să preceadă lansarea în execuție a altora.
  - Dacă un subtask  $v$  trebuie să preceadă un alt subtask  $w$  aceasta se va nota  $v \prec w$ .
  - **Sortarea topologică** a subtaskurilor înseamnă aranjarea lor într-o astfel de ordine încât la inițializarea ori căru subtask, toate subtaskurile care-l condiționează să fie terminate.
- (3) Într-un **program** anumite proceduri pot conține apeluri la alte proceduri.
    - Dacă o procedură  $v$  este apelată de o procedură  $w$ , aceasta se va preciza prin notația  $v \prec w$ .
    - **Sortarea topologică** presupune un astfel de aranjament al declarațiilor de proceduri încât să nu existe nici o referire înainte.
- În general, **ordonarea parțială** a unei **mulțimi**  $M$  presupune **existența** unei **relații** între elementele lui  $M$ .
  - Această **relație** este desemnată prin simbolul  $\prec$ , care înseamnă "precede" și satisface următoarele trei proprietăți:
    - (1) Dacă  $x \prec y$  și  $y \prec z$  atunci  $x \prec z$  (**tranzitivitate**).
    - (2) Dacă  $x \prec y$ , atunci  $y \not\prec x$  (**asimetrie**).
    - (3)  $x \not\prec x$  (**nereflexivitate**).
  - Se presupune că mulțimea  $M$  a elementelor care urmează a fi sortate topologic este **finită**.
  - În aceste condiții o **relație de ordonare parțială** poate fi ilustrată printr-o **diagramă** sau **graf** în care **nodurile** desemnează elementele iar **săgețile orientate** reprezintă relațiile dintre ele.
    - Un exemplu de **relație de ordonare parțială** reprezentată grafic apare în fig.6.4.4.a.
  - **Problema sortării topologice** este aceea de a transforma **ordinea parțială** într-o **ordine liniară**.
  - Din punct de vedere grafic, aceasta implică rearanjarea nodurilor într-un șir astfel încât toate săgețile să indice același sens, spre exemplu spre dreapta (fig.6.4.4.b).



**Fig.6.4.4.a.** Mulțime de elemente parțial ordonată



**Fig.6.4.4.b.** Ordonarea liniară a unei mulțimi parțial ordonate

- Proprietățile (1) și (3) ale relației de ordonare parțială garantează faptul că **graful nu conține bucle**.
  - Aceasta este de fapt **condiția necesară și suficientă** pentru ca **ordonarea liniară** a elementelor mulțimii, respectiv **sortarea topologică** să fie posibilă.
- **Algoritmul sortării topologice** este următorul:
  - (1) Se începe cu selectarea unui element care **nu** este **precedat** de nici un altul.
    - Trebuie să existe cel puțin unul, altfel graful conține o buclă.
  - (2) Elementul este extras din mulțimea M și este plasat în lista ordonată liniar care se creează.
  - (3) Mulțimea rezultată rămâne parțial **ordonată** și în consecință poate fi aplicat din nou același algoritm, până când ea devine vidă.
- Pentru a descrie acest algoritm mai riguros, trebuie precizate **structurile de date**, **reprezentarea** mulțimii M și a **relației de ordonare**.

- Este evident faptul că alegerea acestor reprezentări este determinată de operațiile care urmează să fie realizate, în mod particular de **operația de selecție a elementelor care nu au nici un predecesor**.
- În acest scop, **fiecare element** trebuie reprezentat prin trei **caracteristici**:
  - (1) Cheia sa de identificare.
  - (2) Setul său de succesori.
  - (3) Numărul predecesorilor săi.
- Deoarece numărul  $n$  al elementelor mulțimii  $M$  **nu** este cunoscut a priori, mulțimea inițială este de regulă organizată ca o listă înlănțuită numită **lista principalilor**.
  - În consecință, fiecare element trebuie să mai conțină în plus, legătura la elementul următor al listei principalilor.
  - Pentru simplificare se va presupune că identificatorii elementelor mulțimii, respectiv cheile asociate, sunt numere întregi (nu neapărat consecutive) cuprinse între 1 și  $n$ .
- În mod analog, mulțimea **succesorilor unui element** va fi reprezentată tot ca o listă înlănțuită.
  - Fiecare element al **listei succesorilor** este descris:
    - (1) Prin identificatorul propriu care este de fapt o referință în **lista principalilor**.
    - (2) Printr-o legătură la următorul element al **listei succesorilor** numită și **lista secundarilor**.
- Nodurile **listei principale**, în care fiecare element al mulțimii  $M$  apare exact o singură dată, se vor numi **principali**.
- Nodurile corespunzătoare elementelor din **lista succesorilor**, se vor numi **secundari**.
- În aceste condiții se pot defini următoarele **structuri de date** [6.4.4.a].

-----  
**{Sortarea topologică - structuri de date}**

```

TYPE TipPointerPrincipali = ^TipNodPrincipal;
      TipPointerSecundari = ^TipNodSecundar;
      TipCheie = integer;

      TipNodPrincipal = RECORD {nod în lista principalilor}
        cheie: TipCheie;
        contor: integer;
        urm: TipPointerPrincipali;
        secund: TipPointerSecundari;
      END;
    
```

[6.4.4.a]

```

TipNodSecundar = RECORD    {nod în lista secundarilor}
    id: TipPointerPrincipali;
    urm: TipPointerSecundari
END;

-----
/*{Sortarea topologică - structuri de date}*/

#include <stdlib.h>

typedef struct tip_nod_principal* tip_pointer_principali;
typedef struct tip_nod_secundar* tip_pointer_secundari;
typedef int tip_cheie;

typedef struct tip_nod_principal {
    tip_cheie cheie;
    int contor;
    tip_pointer_principali urm;                /*[6.4.4.a]*/
    tip_pointer_secundari secund;
} tip_nod_principal;    /*nod în lista principalilor*/

typedef struct tip_nod_secundar {
    tip_pointer_principali id;
    tip_pointer_secundari urm;
} tip_nod_secundar;    /*nod în lista secundarilor*/
/*-----*/

```

- Se presupune că mulțimea M și relațiile sale de ordonare sunt furnizate inițial ca o secvență de perechi de chei.
  - Astfel, datele de intrare pentru exemplul din fig.6.4.4.a apar în [6.4.4.b] unde simbolul  $\prec$  este prezent numai din motive de claritate.

```

-----
1<2  2<4  4<6  2<10  4<8  6<3  1<3                [6.4.4.b]
3<5  5<8  7<5  7<9  9<4  9<10
-----

```

- (1) Pentru început, programul de **sortare topologică** trebuie să citească **datele de intrare** și să construiască **structurile de date** aferente.
  - Aceasta se realizează prin citirea succesivă a perechilor de elemente  $x$  și  $y$  ( $x \prec y$ ).
  - Nodurile corespunzătoare sunt localizate prin căutare în **lista principalilor** și dacă **nu** sunt găsite sunt **insearte** în această listă la **sfârșitul** ei.
    - Această sarcină este îndeplinită de funcția **Caut** care returnează pointerul nodului căutat.
  - Pointerii corespunzători celor două elemente  $x$  și  $y$  în **lista principalilor** sunt notați cu  $p$  și  $q$ .

- În continuare în **lista secundarilor** lui  $x$  se adaugă prin inserție în față, un nod nou al cărui câmp identificator se va referi la nodul  $y$ , iar **contorul de predecesori** ai lui  $y$  va fi incrementat cu 1.
- Această primă parte a algoritmului se numește **FazaInițială** și apare în secvența [6.4.4.c].
  - Fragmentul de program prezentat utilizează funcția **Caut**( $w$ ) care returnează pointerul nodului din lista principalilor care are cheia  $w$ .
  - Se presupune că secvența perechilor de chei furnizate la intrare se încheie cu o cifră 0.
  - Pentru setul de date inițiale din [6.4.4.b], structura de date construită drept urmare a fazei inițiale a algoritmului de sortare topologică apare în figura 6.4.4.c.

```

-----
inceput,sfarsit,p,q: TipPointerPrincipali;
t: TipPointerSecundari;
x,y: TipCheie {chei}
z: integer; {contor}

{Sortarea topologică - Faza inițială}
Read(x);
new(inceput); sfarsit:= inceput; z:= 0;
WHILE x<>0 DO
    BEGIN
        Read(y); p:= Caut(x); q:= Caut(y);
        new(t); t^.id:= q; t^.urm:= p^.secund;
        p^.secund:= t; q^.contor:= q^.contor+1;
        Read(x)
    END;
    [6.4.4.c]
-----
/*-----*/
tippointerprincipali inceput, sfarsit,p,q;
tippointersecundari t;
tipcheie x,y; /*chei*/
int z; /*contor*/

/*Sortarea topologică - Faza inițială*/

scanf("%i", &x);
inceput = (tipnodprincipal*)malloc(sizeof(tipnodprincipal));
sfarsit= inceput; z= 0;
while (x!=0)
    {
        scanf("%i", &y);
        p= caut(x);
        q= caut(y);
        t = (tipnodsecundar*)malloc(sizeof(tipnodsecundar));
        t->id= q;
        t->urm= p->secund;
        p->secund= t;
    }
    /*[6.4.4.c]*/

```

```

    q->contor= q->contor+1;
    scanf("%i", &x);
}
/*-----*/

```

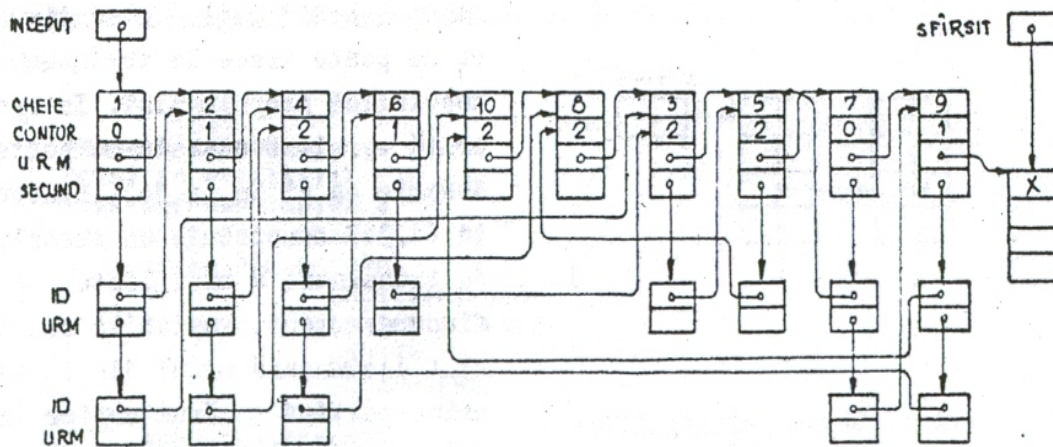


Fig.6.4.4.c. Structură de date pentru sortarea topologică

- (2) După ce **structura de date** a fost construită, urmează realizarea procesului de **sortare topologică** descris anterior.
  - **Sortarea topologică** constă în principiu în **selecția repetată** a unui element cu contorul de predecesori nul.
  - În consecință, pentru început se caută **nodurile cu zero predecesori** și se înlanțuie într-o listă numită **lista principalilor cu zero predecesori**.
  - Întrucât nu mai este nevoie de lista inițială a principalilor, câmpul de înlanțuire urm va fi utilizat pentru a înlanțui nodurile cu zero predecesori în **lista principalilor cu zero predecesori**.
- Această operație prin care se înlocuiește o listă printr-o altă listă apare deosebit de frecvent în procesul de prelucrare a listelor.
- Ea apare în detaliu în secvența [6.4.4.d] în care, din rațiuni de conveniență noua listă este construită în sens invers prin inserție în față.

---

**{Căutarea principalilor cu zero predecesori}**

```

p:= inceput; inceput:= nil;
WHILE p<>sfarsit DO
  BEGIN
    q:= p; p:= q^.urm;

```

[ 6.4.4.d ]

```

    IF q^.contor=0 THEN
        BEGIN {insereaza q^ la începutul noii liste}
            q^.urm:= inceput; inceput:= q
        END
    END;
END;
-----
/*-----*/

/*Căutarea principalilor cu zero predecesori*/

p= inceput; inceput= NULL;
while (p!=sfarsit)
{
    q= p; p= q->urm;
    if (q->contor==0)
    {
        /*insereaza q^ la începutul noii liste*/
        q->urm= inceput; inceput= q;
    }
}
/*-----*/

```

- (3) În continuare se poate trece la **sortarea topologică propriu-zisă**.
  - Rafinarea în două etape a ultimei faze a algoritmului de sortare topologică apare în [6.4.4.e] respectiv [6.4.4.f] și poartă denumirea de **Faza de ieșire**.
    - Ambele secvențe constituie exemple de **traversare a listelor**.
    - În fiecare moment, variabila p desemnează nodul din **lista principalilor** al cărui contor trebuie decrementat și testat.

#### {Faza de ieșire}

```

q:= inceput;
WHILE q<>nil DO
    BEGIN {tipărește elementul curent apoi îl șterge}
        WriteLn(q^.cheie); z:= z-1;
        t:= q^.secund; q:= q^.urm;
        *decrementează contorul de predecesori în toți
          succesorii săi din lista de secundari; dacă
          vreun contor devine zero, inserează acel nod în
          lista principalilor cu zero predecesori;
    END;

```

#### {Decrementează contorul de predecesori ...}

```

WHILE t<>nil DO
    BEGIN
        p:= t^.id; p^.contor:= p^.contor-1;
        IF p^.contor=0 THEN

```

```

        BEGIN {inserare p^ în lista principalilor}
            p^.urm:= q; q:= p
        END;
        t:= t^.urm
    END;

-----
/*-----*/

/*Faza de ieșire*/

q= inceput;                                /*[6.4.4.e]*/
while (q!=NULL)
{
    /*tiparește elementul curent apoi îl suprimă*/
    printf("%i\n", q->cheie); z= z-1;
    t= q->secund; q= q->urm;
    /*decrementează contorul de predecesori în toți
       succesorii săi din lista de secundari; dacă
       vreun contor devine zero, inserează acel nod în
       lista principalilor cu zero predecesori;*/
}
/*-----*/

/*Decrementează contorul de predecesori ...*/

while (t!=NULL)
{
    /*[6.4.4.f]*/
    p= t->id; p->contor= p->contor-1;
    if (p->contor==0)
    {
        /*inserare p^ în lista principalilor*/
        p->urm= q; q= p;
    }
    t= t->urm;
}
/*-----*/

```

- **Contorul** z a fost introdus pentru a contoriza nodurile principale generate în Faza inițială ([6.4.4.c]).
  - Acest contor este decrementat de fiecare dată când un element principal este scris în **Faza de ieșire**.
  - Valoarea sa trebuie să devină în cele din urmă zero.
  - Dacă acest lucru **nu** se întâmplă, înseamnă că în listă mai există elemente dintre care nici unul nu este lipsit de predecesori.
  - În acest caz, în mod evident mulțimea M **nu este parțial ordonată** și în consecință **nu poate fi sortată topologic**.
- Faza de ieșire este un exemplu de proces care gestionează o listă care pulsează, în care elementele sunt inserate și șterse într-o ordine imprevizibilă.



- Codul integral al programului **TopSort** apare în secvența [6.4.4.g].

-----  
**PROGRAM Topsort;**

```

TYPE TipPointerPrincipali = ^TipNodPrincipali;
      TipPointerSecundari = ^TipNodSecundar;
      TipCheie = integer;
      TipNodPrincipali = RECORD
        cheie: TipCheie;
        contor: integer;
        urm: TipPointerPrincipali;           [6.4.4.g]
        secund: TipPointerSecundari
      END;
      TipNodSecundar = RECORD
        id: TipPointerPrincipali;
        urm: TipPointerSecundari
      END;

VAR inceput, sfirsit, p, q: TipPointerPrincipali;
      t: TipPointerSecudari;
      z: integer; {contor noduri}
      x, y: TipCheie; {chei}

FUNCTION Caut(w: TipCheie): TipPointerPrincipali;
  {furnizează referința la nodul principal cu cheia w; dacă
   nu există îl inserează la sfârșit}
  VAR h: TipPointerPrincipali;

  BEGIN
    h:= inceput; sfirsit^.cheie:= w; {fanion}
    WHILE h^.cheie<>w DO h:= h^.urm;
    IF h=sfirsit THEN
      BEGIN {nu există nici un element cu cheia w în
              listă}
        new(sfirsit); z:= z+1;
        h^.contor:= 0; h^.secund:= nil; h^.urm:=sfirsit
      END;
    Caut:= h
  END; {Caut}

BEGIN {se inițializează lista principalilor cu un nod
        fictiv}
  new(inceput); sfirsit:= inceput; z:= 0;
  {Faza inițială}
  Read(x);
  WHILE x<>0 DO
    BEGIN
      Read(y); WriteLn(x,y);
      p:= Caut(x); q:= Caut(y);
      new(t); t^.id:= q; t^.urm:= p^.secund;
      p^.secund:= t; q^.contor:= q^.contor+1; Read(x)
    END;
  {Căutarea principalilor cu zero predecesori (contor=0)}
  p:= inceput; inceput:= nil;

```

```

WHILE p<>sfirsit DO
    BEGIN
        q:= p; p:= p^.urm;
        IF q^.contor=0 THEN
            BEGIN
                q^.urm:= inceput; inceput:= q
            END
        END;
    END;

{Faza de ieşire}
q:= inceput;
WHILE q<>nil DO
    BEGIN
        Write(q^.cheie); z:= z-1;
        t:= q^.secund; q:= q^.urm;
        WHILE t<>nil DO
            BEGIN
                p:= t^.id; p^.contor:= p^.contor-1;
                IF p^.contor=0 THEN
                    BEGIN {inserare p^ în lista principalilor}
                        p^.urm:= q; q:= p
                    END;
                t:= t^.urm
            END
        END;
    END;
    IF z<>0 THEN WriteLn('Multimea nu este partial
                        ordonata!');
END.
-----
/*Programul TopSort*/

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

typedef struct tipnodprincipal* tippointerprincipali;
typedef struct tipnodsecundar* tippointersecundari;
typedef int tipcheie;
typedef struct tipnodprincipal {
    tipcheie cheie;
    int contor;
    tippointerprincipali urm;
    tippointersecundari secund;
} tipnodprincipal;
typedef struct tipnodsecundar {
    tippointerprincipali id;
    tippointersecundari urm;
} tipnodsecundar;

tippointerprincipali inceput,sfirsit,p,q;
tippointersecundari t;
int z; /*contor noduri*/
tipcheie x,y; /*chei*/

tippointerprincipali caut(tipcheie w)
/*furnizează referința la noul principal cu cheia w*/

```

```

{
    tippointerprincipali h;
    tippointerprincipali caut_result;
    h= inceput; sfirsit->cheie= w;    /*fanion*/
    while (h->cheie!=w)  h= h->urm;
    if (h==sfirsit)
    {
        /*nu există nici un element cu cheia w în
        listă*/
        sfirsit =
            (tipnodprincipal*)malloc(sizeof(tipnodprincipal));
        z= z+1;
        h->contor= 0;
        h->secund= NULL;
        h->urm=sfirsit;
    }
    caut_result= h;
    return caut_result;
}    /*caut*/

int main(int argc, const char* argv[])
{
    /*se inițializează lista principalilor cu un nod
    fictiv*/
    inceput =
        (tipnodprincipal*)malloc(sizeof(tipnodprincipal));
    sfirsit= inceput;
    z= 0;

    /*Faza inițială*/

    scanf("%i", &x);
    while (x!=0)
    {
        scanf("%i", &y);
        printf("%i %i\n", x,y);
        p= caut(x);
        q= caut(y);
        t = (tipnodsecundar*)malloc(sizeof(tipnodsecundar));
        t->id= q;
        t->urm= p->secund;
        p->secund= t;
        q->contor= q->contor+1;
        scanf("%i", &x);
    }
    /*căutarea principalilor cu zero predecesori
    (contor=0)*/

    p= inceput;
    inceput= NULL;
    while (p!=sfirsit)
    {
        q= p;
        p= p->urm;
        if (q->contor==0)
        {
            q->urm= inceput;

```

```

        inceput= q;
    }
}

/*Faza de ieşire*/

q= inceput;
while (q!=NULL){
    printf("%i", q->cheie);
    z= z-1;
    t= q->secund;
    q= q->urm;
    while (t!=NULL){
        p= t->id;
        p->contor= p->contor-1;
        if (p->contor==0){ /*inserare p^ în lista
                                principalilor*/
            p->urm= q;
            q= p;
        }
        t= t->urm;
    }
}
if (z!=0)
    printf("Multimea nu este partial ordonata!\n");
return 0;
}
/*-----*/

```

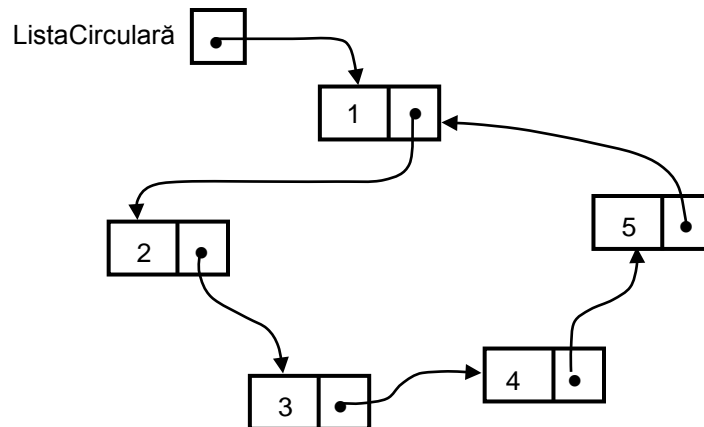
## 6.5. Structuri de date derivate din structura listă

- În cadrul acestui subcapitol vor fi prezentate câteva dintre structurile de date care derivă din structura de date listă, fiind considerate **liste speciale**.
- Este vorba despre: **listele circulare, listele dublu înlănțuite, stivele și cozile**.
- De asemenea se prezintă **funcția de asociere a memoriei** precum și tipurile abstracte de date care pot fi utilizate pentru implementarea ei.
- În general se păstrează regula ca pentru fiecare tip abstract de date să se prezinte câteva posibilități de implementare.

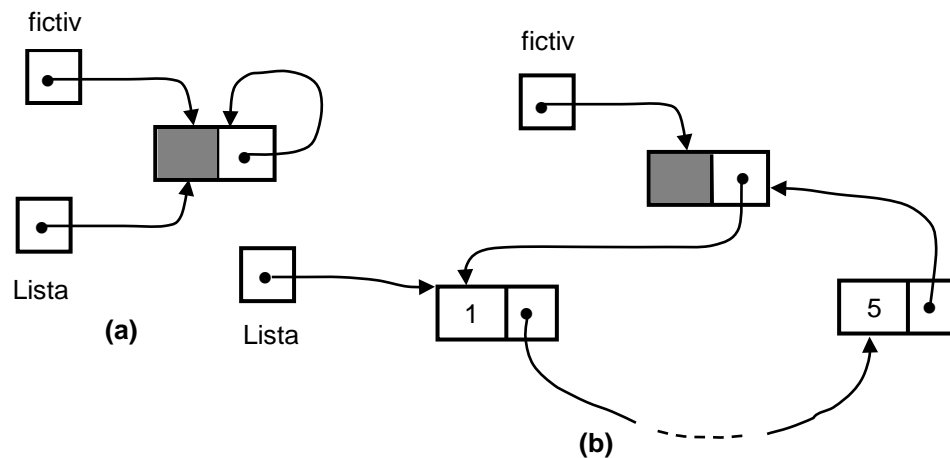
### 6.5.1. Liste circulare

- **Listele circulare** sunt liste înlănțuite ale căror înlănțuiri se **închid**.
- În aceste condiții se pierde noțiunea de început și sfârșit, lista fiind referită de un pointer care se deplasează de-a lungul ei (fig.6.5.1.a).

- **Listele circulare** ridică unele probleme referitoare la **inserția primului nod** în listă și la **suprimarea ultimului nod**.
- O modalitate simplă de rezolvare a acestor situații este aceea de a utiliza un **nod fictiv** într-o manieră asemănătoare celei prezentate la listele obișnuite (**tehnica nodului fictiv** &6.3.2).
- Această modalitate este ilustrată în figura 6.5.1.b.



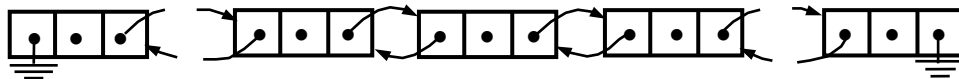
**Fig.6.5.1.a.** Listă circulară



**Fig.6.5.1.b.** Listă circulară implementată prin tehnica nodului fictiv.  
Listă vidă (a), listă normală (b)

### 6.5.2. Liste dublu înlanțuite

- Unele aplicații necesită **traversarea listelor** în ambele sensuri.
  - Cu alte cuvinte fiind dat un element oarecare al listei trebuie determinat cu rapiditate atât **succesorul** cât și **predecesorul** acestuia.
- Maniera cea mai rapidă de a realiza acest lucru este aceea de a memora în fiecare nod al listei referințele "**înainte**" și "**înapoi**".
  - Această abordare conduce la structura **listă dublu înlanțuită**. (fig.6.5.2.a).



**Fig.6.5.2.a.** Listă dublu înlanțuită

- Prețul care se plătește este:
  - (1) Prezența unui câmp suplimentar de tip pointer în fiecare nod.
  - (2) O oarecare creștere a complexității procedurilor care implementează operatorii de bază care prelucrează astfel de liste.
- Dacă implementarea acestor liste se realizează cu **pointeri** se pot defini tipurile de date din secvența [6.5.2.a].

-----  
**{Liste dublu înlanțuite - structuri de date}**

```
TYPE TipPointerNod = ^TipNod;  
    TipNod = RECORD  
        element: TipElement;                [6.5.2.a]  
        anterior,urmator: TipPointerNod  
    END;  
    TipPozitie: TipPointerNod;  
    TipListaDubluInlantuita: TipPointerNod;
```

-----  
**/\*Lista dublu inlantuită\*/**

```
typedef struct tipnod* tippointernod;  
typedef struct tipnod {                      /*[6.5.2.a]*/  
    tipelement element;  
    tippointernod anterior,urmator;
```

```

    } tipnod;
typedef tipinternod tippozitie;
typedef tipinternod tiplistadubluinlantuita;
/*-----*/

```

- Pentru exemplificare se prezintă procedura de **suprimare** a elementului situat în poziția p a unei liste dublu înlanțuite.
- În secvența [6.5.2.b] se prezintă maniera în care se realizează această acțiune, în accepțiunea faptului că nodul suprimat **nu** este nici **primul** nici **ultimul nod al listei**.
  - (1) Se localizează **nodul precedent** și se face câmpul urmator al acestuia să indice **nodul care urmează** celui indicat de p.
  - (2) se modifică câmpul anterior al **nodului care urmează** celui indicat de p astfel încât el să indice **nodul precedent** celui indicat de p.
  - (3) Nodul suprimat este indicat în continuare de p, ca atare spațiul de memorie afectat lui poate fi reutilizat în regim de alocare dinamică a memoriei.

---

**{Liste dublu înlanțuite - suprimarea unui nod}**

```

PROCEDURE Suprima(VAR p: TipPozitie);
BEGIN
    IF p^.anterior<>NIL THEN {nu este primul nod}
        p^.anterior^.urmator:= p^.urmator;
    IF p^.urmator<>NIL THEN {nu este ultimul nod}
        p^.urmator^.anterior:= p^.anterior;
    END; {Suprima}

```

---

**/\*Liste dublu înlanțuite - suprimarea unui nod\*/**

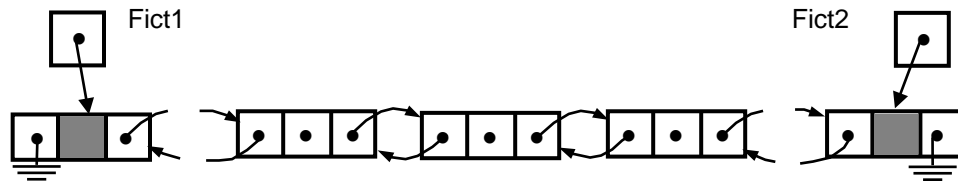
```

void suprima(tippozitie* p)
{
    /*[6.5.2.b]*/
    if ((*p)->anterior!=NULL) /*nu este primul nod*/
        (*p)->anterior->urmator= (*p)->urmator;
    if ((*p)->urmator!=NULL) /*nu este ultimul nod*/
        (*p)->urmator->anterior= (*p)->anterior;
} /*suprima*/
/*-----*/

```

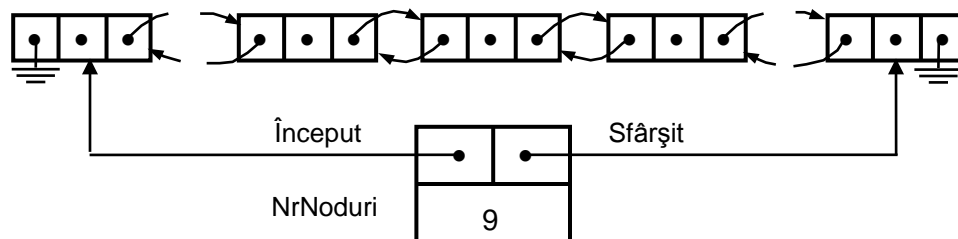
- În practica programării, se pot utiliza diferite **tehnici de implementare a listelor dublu înlanțuite**, derivate din tehnicile de implementare a listelor liniare.
  - Aceste tehnici simplifică implementarea operatorilor care prelucrează astfel de liste în mod deosebit în situații limită (listă vidă sau listă cu un singur nod).
- (1) O primă posibilitate o reprezintă **lista dublu înlanțuită cu două noduri fictive** (fig.6.5.2.b).

- Cele două noduri fictive (Fict1 și Fict2) permit ca **inserția** primului nod al listei respectiv **suprimarea** ultimului nod să se realizeze în manieră similară oricărui alt nod al listei.



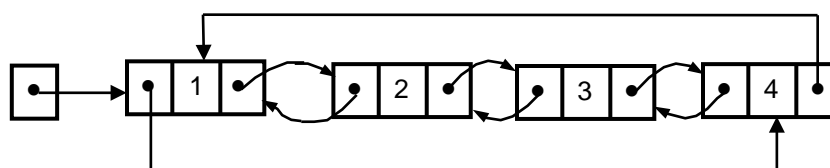
**Fig.6.5.2.b.** Listă dublu înlanțuită. Varianta cu două noduri fictive

- (2) O altă variantă de implementare se bazează pe utilizarea unei structuri care conține:
  - (1) Doi **indicatori** pentru cele două **capete ale listei**.
  - (2) Un **contor de noduri**, utilizat cu deosebire în gestionarea situațiilor limită (când lista e vidă sau conține 1 nod) (fig.6.5.2.c).



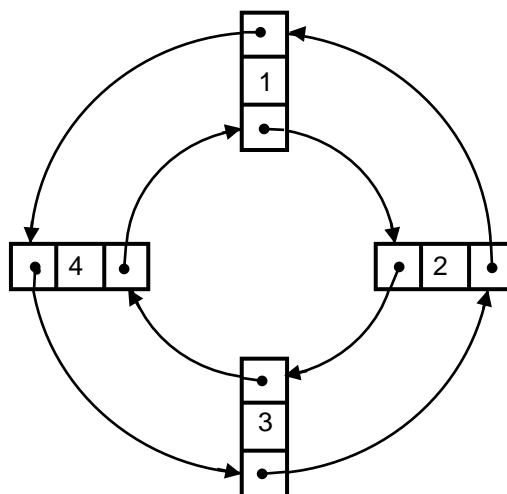
**Fig.6.5.2.c.** Listă dublu înlanțuită. Varianta cu indicatori la capete

- Listele dublu înlanțuite pot fi implementate și ca **liste circulare**.
- În figurile 6.5.2.d respectiv 6.5.2.e apare o astfel de listă în două reprezentări grafice echivalente.



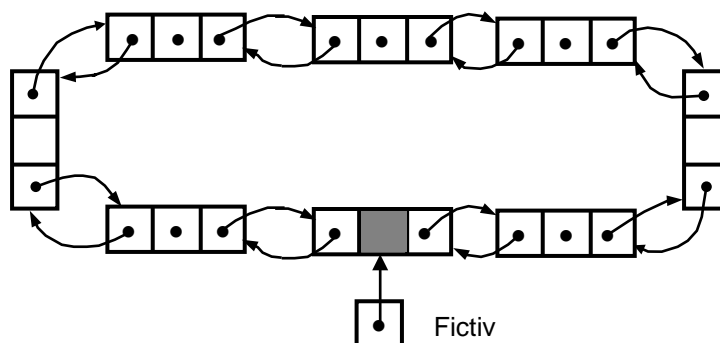
**Fig.6.5.2.d.** Listă dublu înlanțuită circulară





**Fig.6.5.2.e.** Listă dublu înlănțuită circulară

- Este de asemenea posibil a se utiliza la implementarea listelor dublu înlănțuite circulare **tehnica nodului fictiv**, adică un nod care practic "închide cercul".
  - Astfel, câmpul anterior al acestui nod indică ultimul nod al listei, iar câmpul său următor pe primul (fig.6.5.2.f).
  - Când lista este vidă, ambele înlănțuiri indică chiar nodul fictiv.



**Fig.6.5.2.f.** Listă dublu înlănțuită circulară. Varianta cu nod fictiv

### 6.5.3. Stive

- O **stivă** este un tip special de listă în care toate inserările și suprimările se execută la un singur capăt care se numește **vârful stivei**.
- Stivele se mai numesc structuri listă de tip **LIFO (last-in-first-out)** adică "**ultimul-introdus-primul-suprimat**" sau liste de tip "**pushdown**".

- **Modelul** intuitiv al unei **stive** este acela al unui vraf de cărți sau al unui vraf de farfurii pe o masă.
  - În mod evident cea mai convenabilă și în același timp cea mai sigură manieră de a lua un obiect sau de a adăuga un altul este, din motive ușor de înțeles, aceea de a acționa în **vârful** vrafului.

### 6.5.3.1. TDA Stivă

- În maniera consecventă de prezentare a tipurilor de date abstracte adoptată în acest manual, definirea **TDA Stivă** presupune precizarea:
  - (1) Modelului matematic asociat.
  - (2) Notărilor utilizate.
  - (3) Operatorilor definiți pentru acest tip.
- Toate aceste elemente apar precizate în secvența [6.5.3.1.a].

---

#### TDA Stivă

**Modelul matematic:** o secvență finită de noduri. Toate nodurile aparțin unui același tip numit **tip de bază**. O **stivă** este de fapt o listă specială în care toate inserțiile și toate suprimările se fac la un singur capăt care se numește **vârful stivei**.

#### **Notății:**

*s: TipStiva;*  
*x: TipElement.* [6.5.3.1.a]

#### **Operatori:**

1. **Initializeaza**(*s:TipStiva*){:*TipPozitie*}; - face stiva *s* vidă.
  2. **VarfSt**(*s:TipStiva*):*TipElement*; - furnizează elementul din vârful stivei *s*.
  3. **Pop**(*s:TipStiva*); - suprimă elementul din vârful stivei.
  4. **Push**(*x:TipElement,s:TipStiva*); - inserează elementul *x* în vârful stivei *s*. Vechiul vârf devine elementul următor ș.a.m.d.
  5. **Stivid**(*s:TipStiva*):*boolean*; - returnează valoarea adevărat dacă stiva *s* este vidă și fals în caz contrar.
-

- În secvența [6.5.3.1.b] apare un **exemplu** de implementare a **TDA Stivă** utilizând **TDA Listă** varianta restrânsă.

-----  
**{Implementarea TDA Stivă bazată pe TDA Lista (varianta restrânsă)}**

```

TYPE TipStiva = TipLista;
VAR s: TipStiva;
    p: TipPozitie;           [6.5.3.1.b]
    x: TipNod;
    b: boolean;

```

```

{Initializeaza(s:TipStiva);} p:= Initializeaza(s);
{VarfSt(s);}                x:= Furnizeaza(Primul(s),s);
{Pop(s);}                   Suprima(Primul(s),s);
{Push(x,s);}                Insereaza(s,x,Primul(s));
{Stivid(s);}                b:= Fin(s)=0;

```

-----

```

typedef tip_lista tip_stiva;
tip_stiva s;
tip_pozitie p;                /*[6.5.3.1.b]*/
tip_nod x;
boolean b;

/*Initializeaza(s:TipStiva);*/p= initializeaza(s);
/*VarfSt(s);*/                x= furnizeaza(primul(s),s);
/*Pop(s);*/                   suprima(&primul(s),s);
/*Push(x,s);*/                insereaza(s,x,primul(s));
/*Stivid(s);*/                b= fin(s)==0;
/*-----*/

```

- Acesta este în același timp și un exemplu de **implementare ierarhică** a unui **tip de date abstract** care ilustrează:
  - Pe de o parte **flexibilitatea** și **simplitatea** unei astfel de abordări.
  - Pe de altă parte, **invarianța** ei în raport cu **nivelurile ierarhiei**.
- Cu alte cuvinte, **modificarea** implementării **TDA Listă** **nu** afectează sub nici o formă implementarea **TDA Stivă** în accepțiunea păstrării nemodificate a **prototipurilor operatorilor definiți**.
- Utilizarea deosebit de frecventă și cu mare eficiență a **structurii de date stivă** în domeniul programării, a determinat **evoluția** acesteia de la statutul de **structură de date avansată** spre cel de **structură fundamentală**.
- Această tendință s-a concretizat în **implementarea hardware** a acestei structuri în toate sistemele de calcul moderne și în includerea operatorilor specificei tipului de date abstract stivă în **setul de instrucții cablate** al procesoarelor actuale.

### 6.5.3.2. Implementarea TDA Stivă cu ajutorul structurii tablou

- Întrucât **stiva** este o listă cu caracter mai special, **toate** implementările listelor descrise până în prezent sunt valabile și pentru stive.
- În particular, reprezentarea stivelor ca **liste înlănțuite** nu ridică nici un fel de probleme, operatorii **Push** și **Pop** operând doar cu **pointerul de început** și cu primul nod al listei.
- În ceea ce privește implementarea **TDA Stivă** cu ajutorul **tablourilor**:
- Implementarea listelor bazată pe **structura tablou** prezentată în (&6.3.1) **nu** este cea mai propice.
  - **Explicația:** fiecare **Push** și fiecare **Pop** necesită mutarea întregii stive, activitate care necesită un consum de timp proporțional cu numărul de elemente ale stivei.
- O utilizare mai eficientă a structurii tablou ține cont de faptul că inserțiile și suprimările se fac **numai în vârful stivei**.
  - Astfel se poate considera drept **bază a stivei** sfârșitul tabloului (indexul său cel mai mare), stiva crescând în sensul descreșterii indexului în tablou.
  - Un **indicator** numit **vârf** indică poziția curentă a ultimului element al stivei (fig.6.5.3.2.a).
- Structura de date abstractă care se definește pentru această implementare este următoarea [6.5.3.2.a].

---

#### {Implementarea stivelor cu ajutorul tablourilor}

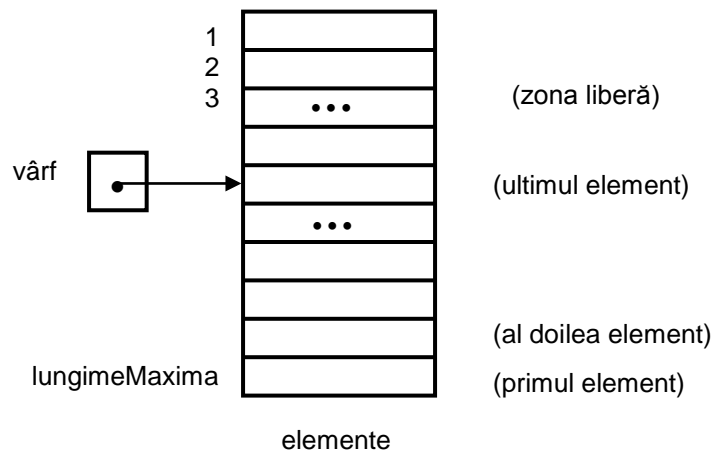
```
TYPE TipStiva = RECORD
    varf: integer;                                [6.5.3.2.a]
    elemente: ARRAY[1..lungimeMaxima] OF TipElement
END;
```

---

#### /\*Implementarea stivelor cu ajutorul tablourilor\*/

```
enum { lungime_maxima = 50};
typedef struct tip_stiva {
    int varf;                                     /*[6.5.3.2.a]*/
    tip_element elemente[lungime_maxima];
} tip_stiva;

/*-----*/
```



**Fig.6.5.3.2.a.** Implementarea TDA Stivă cu ajutorul structurii tablou

- O instanță a stivei constă din secvența `elemente[varf]`, `elemente [varf+1]`, ..., `elemente[lungimeMaxima]`.
- Stiva este **vidă** dacă `varf = lungime Maxima+1`.
- **Operatorii** specifici acestei implementări sunt prezentați în [6.5.3.2.b].

-----  
**{Structura stivă - implementare bazată pe tablouri - operatori specifici}**

```
PROCEDURE Initializeaza(VAR s: TipStiva);
BEGIN
    s.varf:= lungimeMaxima+1
END; {Initializeaza}
```

```
FUNCTION Stivid(s: TipStiva): boolean;
BEGIN
    IF s.varf>lungimeMaxima THEN
        Stivid:= true
    ELSE Stivid:= false
END; {Stivid}
```

```
FUNCTION VarfSt(VAR s: TipStiva): TipElement;
BEGIN
    IF Stivid(s) THEN
        BEGIN
            er:= true; mesaj('stiva este vida')
        END
    ELSE
        VarfSt:= s.elemente[s.varf]
END; {VarfSt} [6.5.3.2.b]
```

```
PROCEDURE Pop(VAR s: TipStiva, VAR x: TipElement);
BEGIN
    IF Stivid(s) THEN
        BEGIN
```

```

        er:= true; mesaj('stiva este vida')
    END
ELSE
    BEGIN
        x:= s.elemente[s.varf]; s.varf:= s.varf+1
    END
END; {Pop}

PROCEDURE Push(x: TipElement; VAR s: TipStiva);
BEGIN
    IF s.varf=1 THEN
        BEGIN
            er:= true; mesaj('stiva este plina')
        END
    ELSE
        BEGIN
            s.varf:= s.varf-1; s.elemente[s.varf]:= x
        END
    END; {Push}

```

---

```

/*Structura stivă - implementare bazată pe tablouri -
operatori specifici*/

#include <stdio.h>
typedef unsigned boolean;
#define true 1
#define false (0)

void initializeaza(tip_stiva* s)
{
    s->varf= lungime_maxima+1;
} /*initializeaza*/

boolean stivid(tip_stiva s)
{
    boolean stivid_result;
    if (s.varf>lungime_maxima)
        stivid_result= true;
    else stivid_result= false;
    return stivid_result;
} /*stivid*/

tip_element varfst(tip_stiva* s)
{
    boolean er;
    tip_element varfst_result;
    if (stivid(*s)){
        er= true;
        printf("stiva este vida");
    }
    else /*[6.5.3.2.b]*/
        varfst_result= s->elemente[s->varf-1];
    return varfst_result;
} /*varfst*/

```

```

void pop(tip_stiva* s, tip_element* x)
{
    boolean er;
    if (stivoid(*s)){
        er= true;
        printf("stiva este vida");
    }
    else{
        *x= s->elemente[s->varf-1];
        s->varf= s->varf+1;
    }
}    /*pop*/

void push(tip_element x, tip_stiva* s)
{
    boolean er;
    if (s->varf==1){
        er= true;
        printf("stiva este plina");
    }
    else{
        s->varf= s->varf-1;
        s->elemente[s->varf-1]= x;
    }
}    /*push*/

/*-----*/

```

### 6.5.3.3. Exemple de utilizare a stivelor

- **Exemplul 6.5.3.3.a.** Evaluarea unei expresii în format postfix.
  - Se consideră o expresie aritmetică în format postfix.
  - Se cere să se redacteze o funcție care evaluează expresia în cauză.
- În această situație se poate utiliza următoarea **tehnică**:
  - (1) Se parcurge expresia de la stânga la dreapta.
  - (2) Se execută operațiile pe măsură ce se întâlnesc, înlocuind de fiecare dată grupul [**operand, operand, operator**] cu valoarea obținută în urma evaluării.
- Se presupune că **forma corectă postfix** a expresiei aritmetice de evaluat este memorată în **lista PostFix**, ale cărei noduri conțin fie un operand fie un operator.
- Funcția **EvaluarePostfix** realizează **evaluarea expresiei** utilizând **stiva Eval** în care memorează valori numerice aflate în așteptarea execuției operațiilor aritmetice necesare [6.5.3.3.a].

---

{Evaluarea unei expresii în format postfix}

**FUNCTION EvaluarePostfix** (PostFix: TipLista): REAL;

{Evaluează o expresie postfix reprezentată ca și o listă de noduri. Se presupune că expresia postfix este corectă}

```
VAR Eval: TipStiva;
    NrVirf, NrSecund, Raspuns: REAL;
    t: NodPostFix;
BEGIN
    Initializeaza(Eval);
    WHILE *mai există noduri în PostFix DO      [6.5.3.3.a]
        BEGIN
            *citește un nod al listei PostFix în t;
            IF t este un numar THEN    {t este un operand}
                Push(t, Eval)
            ELSE
                BEGIN    {t este un operator}
                    NrVirf:= VarfSt(Eval);
                    Pop(Eval);
                    NrSecund:= VarfSt(Eval);
                    Pop(Eval);
                    Raspuns:= NrSecund <t> NrVirf;    {operatorul t}
                    Push(Raspuns, Eval)
                END
            END
        EvaluatePostfix:= VarfSt(Eval)
    END; {EvaluatePostfix}
```

-----

```
float evaluate_postfix (tiplista postfix)
/*Evaluează o expresie postfix reprezentată ca și o listă de
noduri. Se presupune ca expresia postfix este corecta*/
{
    tip_stiva eval;
    float nr_virf, nr_secund, raspuns;
    nod_postfix t;                                     /*[6.5.3.3.a]*/

    float evaluatepostfix_result;
    initializeaza(&eval);
    while *mai exista noduri în postfix
    {
        *citeste un nod al listei postfix în t;
        if t este un numar
            push(t, &eval);
        else
            {
                /*t este un operator*/
                nr_virf= varfst(&eval);
                pop(&eval, 0);
                nr_secund= varfst(&eval);
                pop(&eval, 0);
                raspuns= nr_secund <t> nr_virf; /*operatorul t*/
                push(raspuns, &eval);
            }
    }
    evaluatepostfix_result= varfst(&eval);
    return evaluatepostfix_result;
}    /*evaluate_postfix*/
```



- **Exemplul 6.5.3.3.b.** Eliminarea recursivității.
- După cum s-a precizat în 5.2.4, recursivitatea poate fi **eliminată** practic în orice situație.
  - Eliminarea ei poate conduce la creșterea performanței, însă algoritmul devine de regulă mai complicat și mai greu de înțeles.
- Scopul acestui paragraf este acela de a prezenta un exemplu de **caz general** al eliminării recursivității, adică a manierei în care un **algoritm recursiv** oarecare poate fi transformat într-un **algoritm nerecursiv**.
  - În acest scop se poate utiliza o **structură de date stivă**.
- Pentru **exemplificare** se va prezenta o soluție **recursivă** și una **nerecursivă** a unei versiuni simplificate a problemei clasice a "**sacului călătorului**" ("knapsack problem").
- Formularea problemei:
  - Fiind dată o **valoare** GT (greutatea totală) și un **set de greutate** reprezentate prin întregi pozitivi,  $a_1, a_2, \dots, a_n$ .
  - Se cere să se afle dacă greutatea se pot selecta astfel încât **suma lor să fie exact** GT.
- Spre exemplu dacă GT = 10 și greutatea sunt 7, 5, 4, 4, 1 atunci pot fi selectate a doua, a treia și ultima greutate deoarece 5+4+1=10.
  - Aceasta justifică denumirea de "sac al călătorului" dată problemei, întrucât un călător poate duce în principiu un bagaj de o greutate precizată GT.
  - Varianta completă a acestei probleme care ține cont atât de greutatea obiectelor selectate cât și de valoarea lor, a fost discutată și prezentată în variantă recursivă.
- Algoritmul **recursiv** care rezolvă problema în varianta sa **simplă** apare în secvența [6.5.3.3.b].

-----  
**{Problema Knapsack - Problema sacului călătorului -  
formularea recursivă}**

```
FUNCTION Knapsack(GT:integer; Candidat:integer):boolean;  
  BEGIN  
    IF GT=0 THEN  
      Knapsack:=true [6.5.3.3.b]  
    ELSE  
      IF (GT<0) OR (candidat>n) THEN  
        Knapsack:=false  
      ELSE {soluția cu candidatul curent}
```

```

[*]          IF Knapsack(GT-G[Candidat],Candidat+1) THEN
              BEGIN
                  Write(G[Candidat]);
                  Knapsack:=true
              END
              ELSE {soluția fără candidatul curent}
[**]          Knapsack:= Knapsack(GT,Candidat+1);
              Writeln
              END; {Knapsack}
-----
/* Problema Knapsack - Problema sacului călătorului -
formularea recursivă */

boolean knapsack(int gt, int candidat)
{
    boolean knapsack_result;
    if (gt==0)
        knapsack_result=true;           /*[6.5.3.3.b]*/
    else
        if ((gt<0) || (candidat>n))
            knapsack_result=false;
        else /*soluția cu candidatul curent*/
/*[*]*/      if (knapsack(gt-g[candidat-1],candidat+1))
            {
                printf("%i", g[candidat-1]);
                knapsack_result=true;
            }
            else /*soluția fără candidatul curent*/
/*[**]*/      knapsack_result=knapsack(gt,candidat+1);
    printf("\n");
    return knapsack_result;
} /*knapsack*/
/*-----*/

```

- Funcția recursivă **Knapsack** operează asupra unui **tablou** de forma **G: ARRAY[1..n] OF integer** care memorează **greutățile obiectelor**.
  - Un apel al funcției **Knapsack(s,i)** stabilește dacă există un set de elemente cuprinse în domeniul  $G[i], G[n]$  a căror sumă este exact  $s$  și în caz afirmativ le tipărește.
- **Modul de lucru** preconizat este următorul.
  - În primul rând funcția verifică dacă:
    - (1)  $s = 0$  caz în care soluția problemei a fost găsită și rezultatul este un succes.
    - (2)  $s < 0$  sau  $i > n$ , caz în care nu s-a găsit o sumă egală cu  $s$  sau, au fost parcurse toate greutatețile fără a se găsi o sumă egală cu  $s$ .

- Dacă nici una din aceste situații **nu** este valabilă, atunci se apelează **Knapsack**( $s - G[i], i + 1$ ) pentru a verifica dacă există o soluție care îl **include** pe  $G[i]$ .
  - Dacă există o astfel de soluție, atunci se afișează  $G[i]$ .
  - Dacă **nu** există o astfel de soluție, se apelează **Knapsack**( $s, i + 1$ ), pentru a verifica dacă există o soluție care **nu** îl include pe  $G[i]$ .
- Metoda cea mai generală de **transformare** a unui **algoritm recursiv** într-unul **iterativ** este cea bazată pe introducerea unei **stive** definite și gestionate de către programator.
- Un **nod** al acestei stive conține următoarele elemente (contextul apelului):
    - (1) Valorile curente ale parametrilor de apel ai procedurii.
    - (2) Valorile curente ale tuturor variabilelor locale ale procedurii.
    - (3) O indicație referitoare la adresa de retur, adică referitoare la **locul** în care revine controlul execuției în momentul în care apelul curent al instanței procedurii se termină.
  - În cazul procedurii **Knapsack**, se raționează după cum urmează.
  - (1) În primul rând se observă că ori de câte ori se realizează un **apel al procedurii** materializat prin introducerea unui nod nou în stiva utilizator, **numărul** de candidați crește cu 1.
    - În aceste condiții, variabila `Candidat` poate fi definită ca și o **variabilă globală** care este incrementată cu 1 la fiecare introducere în stivă și decrementată cu 1 la fiecare extragere.
  - (2) O a doua simplificare, se poate face în legătură cu modificarea **adresei de retur** păstrată în stivă.
    - În mod concret, **adresa de retur** pentru această funcție este:
      - Fie situată într-o **altă procedură** care a apelat inițial funcția **Knapsack**.
      - Fie este **adresa următoare** unuia dintre cele două apeluri recursive din interiorul funcției, însemnate cu  $[*]$  – (inclusiunea candidatului) respectiv  $[**]$  – (excluderea candidatului) în secvența [6.5.3.3.b].
    - Cele trei situații pot fi modelate printr-o variabilă de tip `Stare` care poate avea una din următoarele valori:
      - `extern` - indicând un apel din afara funcției **Knapsack**.

- **inclus** - indicând un apel recursiv din locul precizat de **[\*]**, apel care îl include pe **G[Candidat]** în soluție.
- **exclus** - indicând un apel recursiv din locul precizat de **[\*\*]**, care îl exclude pe **G[Candidat]**.
- Dacă se memorează valoarea variabilei de tip **Stare** ca și un indice pentru **adresa de revenire**, atunci **GT** poate fi tratată ca și o **variabilă globală**.
  - Când valoarea variabilei de tip **Stare** se modifică din extern în **inclus** se scade **G[Candidat]** din **GT**.
  - Când aceasta se modifică din **inclus** în **exclus** se adaugă **G[Candidat]** la **GT**.
- Pentru a reprezenta efectul revenirii din procedură în sensul precizării dacă s-a găsit sau nu o soluție, se utilizează variabila globală **Victorie**.
  - O dată pusă pe adevărat, **Victorie** rămâne adevărată și determină **golirea stivei** și afișarea greutăților aflate în starea **inclus**.
- În noile condiții, **stiva** ca atare va fi declarată ca o **listă de stări** [6.5.3.3.c]:

-----  
**{Eliminarea recursivității - Problema sacului călătorului - definirea stivei}**

```

TYPE Stare =(extern,inclus,exclus);           [6.5.3.3.c]
      TipStiva ={o declarație potrivită de stivă având
                  noduri de tip Stare}

```

-----  
**/\* Problema sacului călătorului - definirea stivei\*/**

```

typedef enum {extern_, inclus, exclus} stare;
                                                    /*[6.5.3.3.c]*/
typedef stare tip_element;

tip_stiva = /*o declarație potrivită de stivă având
              noduri de tip_element*/
/*-----*/

```

- Având în vedere cele mai sus precizate, în secvența [6.5.3.3.d] apare **procedura nerecursivă Knapsack** care:
  - Operează asupra unui tablou de greutăți **G**.
  - Este concepută în termenii operatorilor definiți asupra unei **structuri de date abstracte stivă**.
- Deși această procedură poate fi mai rapidă decât funcția **Knapsack** recursivă:
  - Este specifică strict problemei în cauză.

- Conține în mod evident mai multe linii de cod.
  - Este mai complicată.
  - Este mai dificil de înțeles.
- Din aceste motive eliminarea recursivității se recomandă a fi utilizată atunci când factorul **viteză** este deosebit de important.

```

-----
PROCEDURE KnapsackNerecursiv (GT: integer);
  VAR Candidat: integer;
      Victorie: boolean;
      s: TipStiva;
  BEGIN
    Candidat:= 1;
    Victorie:= false;
    Initializare(s);
    Push(extern,s); {inițializare stivă cu G[1] - amorsare
      proces}

    REPEAT
      IF Victorie THEN
        BEGIN {golire stivă și afișarea greutăților
          include în soluție}
          IF VarfSt(s)=inclus THEN
            Writeln(G[Candidat]);
            Candidat:= Candidat-1;
            Pop(s)
          END
        ELSE
          [6.5.3.3.d]
          IF GT=0 THEN
            BEGIN {soluție găsită}
              Victorie:= true; Candidat:= Candidat-1;
              Pop(s)
            END
          ELSE
            IF(((GT<0 AND (VarfSt(s)=extern)) OR
              (Candidat>n)) THEN
              BEGIN {nici o soluție nu este posibilă
                cu aceasta alegere}
                Candidat:= Candidat-1; Pop(s)
              END
            ELSE {nu există încă o soluție; se anali-
              zează starea candidatului curent din s}
              IF VarfSt(s)=extern THEN
                BEGIN {prima încercare de
                  incluere a unui candidat}
                  GT:= GT-G[Candidat];
                  Candidat:= Candidat+1;
                  Pop(s); Push(inclus,s);
                  Push(extern,s)
                END
              ELSE
                IF VarfSt(s)=inclus THEN
                  BEGIN {încercare excludere

```

```

                                candidat}
                                GT:= GT + G[Candidat];
                                Candidat:= Candidat+1;
                                Pop(s); Push(exclus,s);
                                Push(extern,s)
                                END
                                ELSE
                                BEGIN {Varfst(s)=exclus;
                                abandonare selecție curentă}
                                Pop(s); Candidat:= Candidat-1
                                END

                                UNTIL Stivid(s)
                                END; {KnapsackNecursiv}
-----
/*-----*/
void knapsacknerecursiv (int gt)
{
    int candidat;
    boolean victorie;
    tip_stiva s;

                                                                    /*[6.5.3.3.d]*/

    candidat= 1;
    victorie= false;
    initializeaza(&s);
    push(extern_,&s); /*initializare stiva cu G[1]*/
    do {
        if (victorie)
        {
            /*golire stivă și afișarea greutăților
            include în soluție*/
            if (varfst(&s)==inclus)
                printf("%i\n", g[candidat-1]);
            candidat= candidat-1;
            pop(&s, 0);
        }
        else
        if (gt==0)
        {
            /*soluție găsită*/
            victorie= true;
            candidat= candidat-1;
            pop(&s, 0);
        }
        else
        if ((gt<0 && (varfst(&s)==extern_)) ||
            (candidat>n))
        {
            /*nici o soluție nu este posibilă
            cu această alegere*/
            candidat= candidat-1; pop(&s, 0);
        }
        else /*nu exista încă o soluție; se
        consideră starea candidatului curent*/
        if (varfst(&s)==extern_)
        {
            /*prima încercare de
            includere a unui candidat*/
            gt= gt-g[candidat-1];
            candidat= candidat+1;

```

```

    }
    else
        if (varfst(&s)==inclus)
        {
            /*încercare excludere
            candidat*/
            gt= gt + g[candidat-1];
            candidat= candidat+1;
            pop(&s, 0); push(exclus,&s);
            push(extern_,&s);
        }
        else
        {
            /*varfst(s)=exclus;
            abandonare selecție curentă*/
            pop(&s,0); candidat= candidat-1;
        }
    } while (!(stivid(s)));
} /*knapsack_necursiv*/
/*-----*/

```

#### 6.5.4. Cozi

- Cozile sunt o altă categorie specială de liste în care elementele sunt inserate la un capăt (**spate**) și sunt suprimate la celălalt (**început**).
  - Cozile se mai numesc liste "**FIFO**" ("**first-in first-out**") adică liste de tip "**primul-venit-primul-servit**".
- Operațiile care se execută asupra **cozii** sunt analoage celor care se realizează asupra **stivei** cu diferența că inserările se fac la **spatele** cozii și **nu** la **începutul** ei și că ele diferă ca și terminologie.

##### 6.5.4.1. Tipul de date abstract Coadă

- În acord cu abordările anterioare în secvența [6.5.4.1.a] se definesc **două** variante ale **TDA Coadă**.
- În secvența [6.5.4.1.b] se prezintă un exemplu de implementare a **TDA Coadă** bazat pe **TDA Listă**.

---

##### TDA Coadă

**Modelul matematic:** o secvență finită de noduri. Toate nodurile aparțin unui aceluiași tip numit **tip de bază**. O **coadă** este de fapt o listă specială în care toate inserțiile se fac la un capăt (**spate**) și toate suprimările se fac la celălalt capăt (**cap**).

##### **Notății:**

*C*: *TipCoadă*;  
*x*: *TipElement*;

[6.5.4.1.a]

*b: boolean;*

#### **Operatori (setul 1):**

1. **Initializeaza**(*C: TipCoadă*){: *TipPozitie*}; - face coada *C* vidă.
2. **Cap**(*C: TipCoadă*): *TipElement*; - funcție care returnează primul element al cozii *C*.
3. **Adauga**(*x: TipElement, C: TipCoadă*); - inserează elementul *x* la spatele cozii *C*.
4. **Scoate**(*C: TipCoadă*); - suprimă primul element al lui *C*.
5. **Vid**(*C: TipCoadă*): *boolean*; - este adevărat dacă și numai dacă *C* este o coadă vidă.

#### **Operatori (setul 2):**

1. **CreazaCoadăVidă**(*C: TipCoadă*); - face coada *C* vidă.
2. **CoadăVidă**(*C: TipCoadă*): *boolean*; - este adevărat dacă și numai dacă *C* este o coadă vidă.
3. **CoadăPlină**(*C: TipCoadă*): *boolean*; - este adevărat dacă și numai dacă *C* este o coadă plină (operator dependent de implementare).
4. **InCoadă**(*C: TipCoadă x: TipElement*); - inserează elementul *x* la spatele cozii *C* (**EnQueue**).
5. **DinCoadă**(*C: TipCoadă, x: TipElement*); - suprimă primul element al lui *C* (**DeQueue**).

-----  
**{Exemplu de implementare a TDA Coadă cu ajutorul TDA Listă (setul 1 de operatori)}**

**TYPE** TipCoadă = TipLista;

**VAR** C: TipCoadă;

    p: TipPozitie;

    x: TipElement;

[6.5.4.1.b]

    b: boolean;

{**Initializeaza**(*C*);}

    p:= **Initializeaza**(*C*);

{**Cap**(*C*);}

    x:= **Furnizeaza**(**Primul**(*C*),*C*);

{**Adauga**(*x,C*);}

**Inseareaza**(*C,x,Fin*(*C*));

{**Scoate**(*C*);}

**Suprima**(**Primul**(*C*),*C*);

{**Vid**(*C*),*C*);}

    b:= **Fin**(*C*)=0;

-----  
/\*-----\*/

**/\*Exemplu de implementare a TDA Coadă cu ajutorul TDA Lista (setul 1 de operatori).\*/**



```

typedef tip_lista tip_coada;
tip_coada c;
tip_pozitie p;
tip_element x;
boolean b;

/*Initializeaza(C);*/          p= initializeaza(c);
/*Cap(C);*/                   x= furnizeaza(primul(c),c);
/*Adauga(x,C);*/               insereaza(c,x,fin(c));
/*Scoate(C);*/                 suprima(primul(c),c);
/*Vid(C),C);*/                 b= fin(c)==0;
/*-----*/

```

#### 6.5.4.2. Implementarea cozilor cu ajutorul pointerilor

- Ca și în cazul stivelor, orice **implementare** a listelor este valabilă și pentru **cozi**.
- Pornind însă de la observația că inserțiile se fac numai la **spatele** cozii, procedura **Adauga** poate fi concepută mai eficient.
  - Astfel, în loc de a parcurge de fiecare dată coada de la început la sfârșit atunci când se dorește adăugarea unui element, se va păstra un **pointer** la **ultimul element al cozii**.
  - De asemenea, ca și la toate tipurile de liste, se va păstra și pointerul la **primul element al listei** utilizat în execuția comenzilor **Cap** și **Scoate**.
- În implementare se poate utiliza un **nod fictiv** ca și prim nod al cozii (tehnica “**nodului fictiv**”), caz în care pointerul de început va indica acest nod.
  - Această convenție care permite o manipulare mai **convenabilă** a cozilor vide, va fi utilizată în continuare în implementarea bazată pe pointeri a **structurii de date coadă**.
- **Tipurile de date** care se utilizează în acest scop sunt următoarele [6.5.4.2.a]:

-----  
 {Implementarea cozilor cu ajutorul pointerilor - definirea structurilor de date}

```

TYPE TipReferintaNod=^TipNod;
      TipNod = RECORD
          element: TipElement;
          urm: TipReferintaNod
      END;

```

[6.5.4.2.a]

-----  
 /\* Implementarea cozilor cu ajutorul pointerilor - definirea structurilor de date \*/

```

#include <stdlib.h>
typedef struct tip_nod* tip_referinta_nod;
typedef struct tip_nod {

```

```

tip_element element;                                /*[6.5.4.2.a]*/
tip_referinta_nod urm;
} tip_nod;
/*-----*/

```

- În aceste condiții se poate defini o **structură de date coadă** care constă din **doi pointeri** indicând **începutul** respectiv **spatele** unei liste înlănțuite.

- Primul nod al cozii este unul **fictiv** în care câmpul element este ignorat.
- Această convenție, după cum s-a menționat mai înainte permite o reprezentare și o manipulare mai simplă a cozii vide.

- Astfel TipCoadă se poate defini după cum se prezintă în [6.5.4.2.b].

-----

**{Implementarea cozilor cu ajutorul pointerilor - definirea lui TipCoadă}**

```

TipCoadă = RECORD
    inceput,spate: TipReferintaNod                [6.5.4.2.b]
END;

```

-----

**/\* Definirea lui tipcoada \*/**

```

typedef struct tip_coadă {
    tip_referinta_nod inceput,spate;                /*[6.5.4.2.b]*/
} tip_coadă;
/*-----*/

```

- În continuare în [6.5.4.2.c] se prezintă secvențele de program care implementează operatorii (setul 1) definiți asupra cozilor.

-----

**{Implementarea cozilor cu ajutorul pointerilor - implementarea operatorilor specifici (setul 1)}**

```

PROCEDURE Initializeaza(VAR C: TipCoadă);
BEGIN
    NEW(C.inceput); {crează nodul fictiv}
    C.inceput^.urm:= nil;
    C.spate:= C.inceput; {nodul fictiv este primul și
                        ultimul nod al cozii}
END;{Initializeaza}

FUNCTION Vid(C: TipCoadă): boolean;
BEGIN
    IF C.inceput:=C.spate THEN Vid:=true ELSE Vid:=false
END;{Vid}

```

```

FUNCTION Cap(C: TipCoadă): TipElement;
BEGIN
    IF Vid(C) THEN
        BEGIN
            er:= true; MESAJ('coada este vidă')

```

[6.5.4.2.c]

```

        END
    ELSE
        Cap:= C.inceput^.urm^.element
    END;{Cap}

PROCEDURE Adauga(x: TipElement; VAR C: TipCoadă);
BEGIN
    NEW(C.spate^.urm); {se adaugă un nod nou la
                        spatele cozii}
    C.spate:= C.spate^.urm;
    C.spate^.element:= x;
    C.spate^.urm:= nil
END;{Adauga}

PROCEDURE Scoate(VAR C: TipCoadă);
BEGIN
    IF Vid(C) THEN
        BEGIN
            er:= true; WRITE('coada este vidă')
        END
    ELSE
        C.inceput:= C.inceput^.urm
    END;{Scoate}

```

---

```

/* Implementarea cozilor cu ajutorul pointerilor -
implementarea operatorilor specifici (setul 1)- C*/

void initializeaza(tip_coadă* c)
{
    c->inceput = (tip_nod*)malloc(sizeof(tip_nod)); /*crează
nodul fictiv*/
    c->inceput->urm= NULL;
    c->spate= c->inceput; /*nodul fictiv este primul și
                        ultimul nod al cozii*/
}/*initializeaza*/

boolean vid(tip_coadă c)
{
    boolean vid_result;
    if (c->inceput=c->spate)
        vid_result=true;
    else vid_result=false;
    return vid_result;
}/*vid*/

void cap(tip_coadă c)
{
    /*[6.5.4.2.c]*/
    void cap_result;
    if (vid(c))
    {
        er= true;
        printf("coada este vida");
    }
    else
        cap_result= c.inceput->urm->element;

```

```

    return cap_result;
} /*cap*/

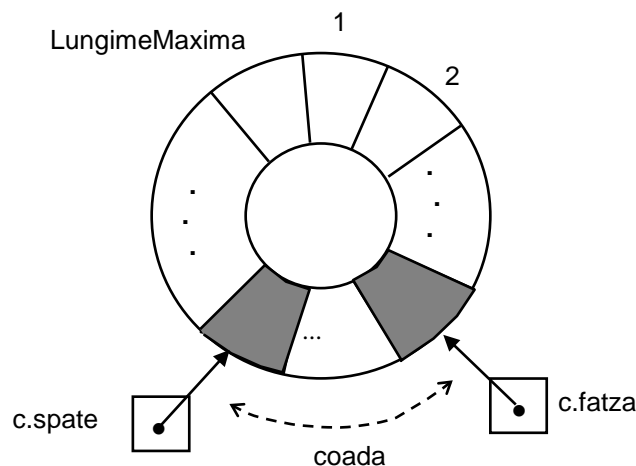
void adauga(tip_element x, tip_coada* c)
{
    c->spate->urm = (tipnod*)malloc(sizeof(tipnod));
    /*se adaugă un nod nou la spatele cozii*/
    c->spate = c->spate->urm;
    c->spate->element = x;
    c->spate->urm = NULL;
} /*adauga*/

void scoate(tip_coada* c)
{
    if (vid(*c))
    {
        er = true;
        printf("coada este vida");
    }
    else
        c->inceput = c->inceput->urm;
} /*scoate*/
/*-----*/

```

#### 6.5.4.3. Implementarea cozilor cu ajutorul tablourilor circulare

- Reprezentarea **listelor** cu ajutorul **tablourilor**, prezentată în &6.3.1, **poate** fi utilizată și pentru **cozi**, dar **nu** este foarte eficientă.
  - Într-adevăr, indicatorul la ultimul element al listei, permite implementarea **simplă**, într-un număr fix de pași, a operației **Adauga**.
  - Execuția operației **Scoate** presupune însă **mutarea** întregii cozi cu o poziție în tablou, operație care necesită un timp  $O(n)$ , dacă coada are lungimea  $n$ .
- Pentru a depăși acest dezavantaj se poate utiliza o **structură de tablou circular** în care prima poziție urmează ultimei, după cum rezultă din figura 6.5.4.3.



#### Fig.6.5.4.3. Implementarea unei cozi cu ajutorul unui tablou circular

- Coada se găsește undeva în jurul cercului, în **poziții consecutive**, având **spatele** undeva înaintea **feței** la parcurgerea cercului în sensul acelor de ceasornic.
- **Adăugarea** unui element presupune mișcarea indicatorului `C.spate` cu o poziție în sensul acelor de ceasornic și introducerea elementului în această poziție.
- **Scoaterea** unui element din coadă, presupune simpla mutare a indicatorului `C.fatza` cu o poziție în sensul rotirii acelor de ceasornic.
  - Astfel coada **se rotește** în sensul rotirii acelor de ceasornic după cum se **adaugă** sau se **scot** elemente din ea.
- În aceste condiții atât procedura **Adauga** cât și **Scoate** pot fi redactate astfel încât să presupună un **număr fix de pași de execuție**, cu alte cuvinte să fie  $O(1)$ .
- În cazul implementării ce apare în continuare, indicatorul `C.fatza` indică **primul element al cozii**, iar indicatorul `C.spate` **ultimul element al cozii**.
- Acest mod de implementare ridică însă o problemă referitoare la sesizarea **cozii vide** și a **cele pline**.
- Presupunând că structura **coadă** reprezentată în figura 6.5.4.3 este **plină** (conține `LungimeMaxima` elemente), indicatorul `C.spate` va indica **poziția adiacentă** lui `C.fatza` la parcurgerea în sensul arcelor de ceasornic (trigonometric negativ).
- Pentru a preciza reprezentarea **cozii vide** se presupune o coadă formată dintr-un singur element.
  - În acest caz `C.fatza` și `C.spate` indică aceeași poziție.
  - Dacă se scoate **singurul** element, `C.fatza` avansează cu o poziție în față (în sensul acelor de ceasornic) indicând **coada vidă**.
  - Se observă însă că această situație este **identică** cea anterioară care indică coada plină, adică `C.fatza` se află cu o poziție în față lui `C.spate`, la parcurgerea în sensul arcelor de ceasornic (trigonometric negativ) a cozii.
- În vederea rezolvării acestei situații, în tablou se vor introduce **doar** `LungimeMaxima-1` elemente, deși în tablou există `LungimeMaxima` poziții.
  - Astfel testul de **coadă plină** conduce la o valoare adevărată dacă `C.spate` devine egal cu `C.fatza` după **două** avansări succesive.
  - Testul de **coadă vidă** conduce la o valoare adevărată dacă `C.spate` devine egal cu `C.fatza` după avansul cu o poziție.

- Evident avansările se realizează în sensul acelor de ceasornic al parcurgerii cozii.
- În continuare se prezintă implementarea **operatorilor** definiți peste o **structură de date coadă** definită după cum urmează [6.5.4.3.a]:

-----  
 {Implementarea cozilor cu ajutorul tablourilor circulare -  
 definirea structurilor de date}

```
TYPE TipCoadă = RECORD                                [6.5.4.3.a]
    elemente: ARRAY[1..LungimeMaxima] OF TipElement;
    fatza,spate: TipIndice
END;
```

-----  
 /\* Implementarea cozilor cu ajutorul tablourilor circulare -  
 definirea structurilor de date \*/

```
typedef struct tip_coadă {                            /*[6.5.4.3.a]*/
    tip_element elemente[lungime_maxima];
    tip_indice fatza,spate;
} tip_coadă;
/*-----*/
```

- Secvența de instrucții corespunzătoare apare în [6.5.4.3.b].
- Funcția **Avanseaza**(C) furnizează poziția următoare celei curente în tabloul circular.

-----  
 {Implementarea cozilor cu ajutorul tablourilor circulare -  
 implementarea operatorilor specifici}

```
FUNCTION Avanseaza(VAR i: TipIndice): TipIndice;
BEGIN
    Avanseaza:= (i mod LungimeMaxima)+1
END; {Avanseaza}
```

```
PROCEDURE Initializeaza(VAR C: TipCoadă);
BEGIN
    C.fatza:= 1;                                     [6.5.4.3.b]
    C.spate:= LungimeMaxima
END; {Initializeaza}
```

```
FUNCTION Vid(C: TipCoadă): boolean;
BEGIN
    IF Avanseaza(C.spate)=C.fatza THEN
        Vid:= true
    ELSE
        Vid:= false
    END; {Vid}
```

```
FUNCTION Cap(VAR C: TipCoadă): TipElement;
BEGIN
    IF Vid(C) THEN
        BEGIN
```

```

        er:= true; MESAJ('coada e vidă')
    END
ELSE
    Cap:= C.elemente[C.fatza]
END; {Cap}

PROCEDURE Adauga(x: TipElement; VAR C: TipCoadă);
BEGIN
    IF Avanseaza(Avanseaza(C.spate))=C.fatza THEN
        BEGIN
            er:= true; MESAJ('coada este plină')
        END
    ELSE
        BEGIN
            C.spate:= Avanseaza(C.spate);
            C.elemente[C.spate]:= x
        END
    END; {Adauga}

PROCEDURE Scoate(VAR C: TipCoadă);
BEGIN
    IF Vid(C) THEN
        BEGIN
            er:= true;
            MESAJ ('coada este vidă')
        END
    ELSE
        C.fatza:= Avanseaza(C.fatza)
    END; {Scoate}

```

-----

*/\* Implementarea cozilor cu ajutorul tablourilor circulare - implementarea operatorilor specifici \*/*

```

tip_indice avanseaza(tip_indice* i)
{
    tip_indice avanseaza_result;
    avanseaza_result= (*i % lungime_maxima)+1;
    return avanseaza_result;
} /*avanseaza*/

void initializeaza(tip_coadă* c)
{
    c->fatza= 1; /*[6.5.4.3.b]*/
    c->spate= lungime_maxima;
} /*initializeaza*/

boolean vid(tip_coadă c)
{
    boolean vid_result;
    if (avanseaza(&c.spate)==c.fatza)
        vid_result= true;
    else
        vid_result= false;
    return vid_result;
} /*vid*/

```

```

tip_element cap(tip_coadă* c)
{
    tip_element cap_result;
    if (vid(*c))
    {
        er= true; printf("coada e vida");
    }
    else
        cap_result= c->elemente[c->fatza-1];
    return cap_result;
} /*cap*/

void adauga(tip_element x, tip_coadă* c)
{
    if (avanseaza(&avanseaza(&c->spate))==c->fatza)
    {
        er= true; printf("coada este plina");
    }
    else
    {
        c->spate= avanseaza(&c->spate);
        c->elemente[c->spate-1]= x;
    }
} /*adauga*/

void scoate(tip_coadă* c)
{
    if (vid(*c))
    {
        er= true;
        printf ("coada este vida");
    }
    else
        c->fatza= avanseaza(&c->fatza);
} /*scoate*/

/*-----*/

```

- Problemele legate de sesizarea **cozii vide** și a celei **pline** pot fi rezolvate **mai simplu** prin introducerea unui **contor** al numărului de elemente din coadă.
  - Astfel valoarea 0 a acestui contor semnifică coadă vidă iar valoarea DimMax coadă plină.
- Structura de date corespunzătoare acestei abordări apare în secvența [6.5.4.3.c].
- Procedurile care implementează operatorii specifici în aceste circumstanțe pot fi dezvoltate cu ușurință în baza exemplelor prezentate anterior.

-----  
 {Implementarea cozilor cu ajutorul tablourilor circulare -  
 definirea structurilor de date (varianta 2)}

```

CONST DimMax = ...;
        DimCoadă = DimMax+1;

```



```

TYPE TipElement = ...;
    TipIndex = 0..DimMax;
    TipContor = 0..DimCoadă;
    TipTablou = ARRAY[TipIndex] OF TipElement;
    TipCoadă = RECORD
        fatza,spate: TipIndex;
        contor: TipContor;
        elemente: TipTablou
    END;

VAR C: TipCoadă;
    x: TipElement;

-----
/* Implementarea cozilor cu ajutorul tablourilor circulare -
definirea structurilor de date (varianta 2*/

enum { dim_max = 100,
        dim_coadă = dim_max+1};

typedef int tip_element;
typedef unsigned char tip_index;
                                                    /*[6.5.4.3.c]*/
typedef unsigned char tip_contor;

typedef tip_element tip_tablou[dim_max+1];
typedef struct tip_coadă {
    tip_index fatza,spate;
    tip_contor contor;
    tip_tablou elemente;
} tip_coadă;

tip_coadă c;
tip_element x;
/*-----*/

```

#### 6.5.4.4. Aplicație. Conversia infix-postfix

- Aplicația de față realizează conversia unei **expresii infix** în **reprezentare postfix** utilizând o implementare **nerecursivă**.
  - Ambele expresii sunt reprezentate prin intermediul unor **liste înlanțuite**.
- Se vor utiliza două structuri de date auxiliare:
  - (1) O **stivă** StivaOp (stiva operatorilor) care memorează operatorii pentru care nu s-au determinat încă ambii operanzi.
  - (2) O **coadă** CoadăPost care înregistrează forma postfix a expresiei care se construiește.
- Fiecărui **operator** din expresia infix i se asociază o **valoare de precedență**:

- Operatorii '\*' și '/' au cea mai mare precedență.
- Operatorii '+' și '-' precedența următoare.
- În vederea simplificării proiectării algoritmului, în mod artificial se acordă parantezei stânga '(' prioritatea cea mai scăzută.
- Codul efectiv al procedurii apare în secvența [6.5.4.4.a].
  - După cum se observă transferurile se realizează din StivaOp în CoadăPost, care în final va putea fi utilizată ca și intrare a procedurii de evaluare.

---

**{Aplicație: Conversie din format Infix în format Postfix}**

```

PROCEDURE ConversieInPostFix(Infix: TipLista,
                               VAR CoadăPost: TipCoadă);
VAR StivaOp: TipStiva;

PROCEDURE Transfer(VAR S: TipStiva, VAR Q: TipCoadă);
{Transferă vârful stivei S la sfârșitul cozii Q. Se
presupune că atomii din S și Q aparțin aceluiași tip}
BEGIN
    Adauga(VarfSt(S),Q);
    Pop(S)                                     [6.5.4.4.a]
END;{Transfer}
BEGIN
    Initializeaza(StivaOp);
    Initializeaza(CoadăPost);
    WHILE *mai există noduri în lista Infix DO
        BEGIN
            *ia un nod t, din lista Infix;
[1]    IF *t este un număr THEN
            Adauga(t, CoadăPost)
        ELSE
[2]    IF StivId(StivaOp) THEN
            Push(t, StivaOp)
        ELSE
[3]    IF *t este paranteză stânga THEN
            Push(t, StivaOp)
        ELSE
[4]    IF *t este paranteză dreaptă THEN
            BEGIN
                WHILE *VarfSt(StivaOp) este
                    diferit de paranteză stânga DO
                    Transfer (StivaOp,CoadăPost);
                    {descarcă până la proxima paranteză
                     stânga din stivă}
                Pop(StivaOp){descarcă paranteză stânga
                     din stivă}
            END
        ELSE

```

```

[5]          BEGIN
              WHILE *precedența lui t ≤ precedența
                lui Varfst(StivaOp) DO
                  Transfer(StivaOp, CoadăPost)
                  Push(t, StivaOp)
              END
          END;{WHILE}
[6] WHILE NOT Stivid(StivaOp) DO
      Transfer(StivaOp, CoadăPost) {transferă operatorii
                                   rămași}

      END;{ConversieInPostFix}
-----
/* Conversie din format Infix în format Postfix */

void conversie_in_postfix(tip_lista infix,
                          tip_coadă* coada_post);

static void transfer(tip_stiva* s, tip_coadă* q){
    /*Transfera vârful stivei s la sfârșitul cozii q. Se
      presupune ca atomii din s și q aparțin aceluiași tip*/
      adauga(varfst(s), q);
      pop(s, 0);                                     /*[6.5.4.4.a]*/
} /*transfer*/

void conversie_in_postfix(tip_lista infix, tip_coadă*
coada_post) {
    tip_stiva stiva_op;
    initializeaza(&stiva_op);
    initializeaza(coada_post);
    while (mai exista noduri în lista infix)
    {
        *ia un nod t, din lista infix;
    /*[1]*/if (t este un numar)
        adauga(t, coada_post);
    else
    /*[2]*/ if (stivid(stiva_op))
        push(t, &stiva_op);
    else
    /*[3]*/ if (t este paranteza stânga)
        push(t, &stiva_op);
    else
    /*[4]*/ if (t este paranteza dreapta){
        while (varfst(stiva_op) este diferit de
                paranteza stânga)
            transfer (&stiva_op, coada_post);
        pop(&stiva_op, 0); /*descarcă până la
                           proxima paranteză stânga din stiva*/
    else{
    /*[5]*/ while (precedenta lui t = precedenta
                    lui varfst(stiva_op))
            transfer(&stiva_op, coada_post);
            push(t, &stiva_op);}
    } /*while*/
    /*[6]*/while (! stivid(stiva_op))
        transfer(&stiva_op, coada_post); /*transferă

```

```

                                                                    operatorii rămași*/
} /*conversie_in_postfix*/
/*-----*/

```

- Procedura **ConversieInPostfix**, ilustrează o mică parte a activității pe care **compilatorul** sau **interpretorul** o realizează în momentul translatării expresiilor aritmetice în cod executabil.
  - De fapt sunt eludate fazele de analiză sintactică și semantică ale expresiei precum și natura efectivă a operanzilor (constante, variabile etc.).
- **Varianta recursivă** a aceleiași aplicații a fost prezentată la exemple de algoritmi recursivi.

### 6.5.5. Cozi bazate pe prioritate

- **Coadă bazată pe prioritate** (“**priority queue**”) este structura de date abstractă care permite **inserția** unui element și **suprimarea celui mai prioritar** element.
- O astfel de structură diferă atât față de structura **coadă** (din care se suprimă **primul** venit, deci **cel mai vechi**) cât și față de **stivă** (din care se suprimă **ultimul** venit, deci **cel mai nou**).
- De fapt **cozile bazate pe prioritate** pot fi concepute ca și structuri care **generalizează cozile și stivele**.
  - **Cozile și stivele** pot fi implementate prin cozi bazate pe prioritate utilizând **priorități corespunzătoare**.
- **Aplicațiile** cozilor bazate pe prioritate sunt:
  - Sisteme de simulare - unde cheile pot corespunde unor evenimente "de timp" ce trebuie să decurgă în ordine temporală.
  - Planificarea job-urilor.
  - Traversări speciale ale unor structuri de date, etc.

#### 6.5.5.1. TDA Coadă bazată pe prioritate

- Considerând **coada bazată pe prioritate** drept o structură de date abstractă ale cărei elemente sunt articole cu **chei** afectate de **priorități**, definirea acestora apare în [6.5.5.1.a].

---

#### TDA Coadă bazată pe prioritate

**Modelul matematic:** o secvență finită de noduri. Toate nodurile aparțin unui aceluiasi tip numit **tip de bază**. Fiecare nod are o asociată o prioritate. O **coadă bazată**

**pe prioritate** este de fapt o listă specială în care se permite inserția normală și suprimarea doar a celui mai prioritar nod.

**Notății:**

*q*: *TipCoadăBazatăPePrioritate*; [6.5.5.1.a]  
*x*: *TipElement*;

**Operatori:**

1. **Initializează**(*q*: *TipCoadăBazatăPePrioritate*); - face coada *q* vidă.

2. **Inserează**(*x*: *TipElement*,  
*q*: *CoadăBazatăPePrioritate*); - inserția unui nou element *x* în coada *q*.

3. **Extrage**(*q*: *CoadăBazatăPePrioritate*): *TipElement*;  
- extrage cel mai prioritar element al cozii *q*.

4. **Inlocuiește**(*q*: *CoadăBazatăPePrioritate*,  
*x*: *TipElement*): *TipElement*; - înlocuiește cel mai prioritar element al cozii *q* cu elementul *x*, mai puțin în situația în care noul element este cel mai prioritar element. Returnează cel mai prioritar element.

5. **Schimbă**(*q*: *TipCoadăBazatăPePrioritate*, *x*:  
*TipElement*; *p*: *TipPrioritate*); - schimbă prioritatea elementului *x* al cozii *q* și îi conferă valoarea *p*.

6. **Suprimă**(*q*: *TipCoadăBazatăPePrioritate*,  
*x*: *TipElement*); - suprimă elementul *x* din coada *q*.

7. **Vid**(*q*: *TipCoadăBazatăPePrioritate*): *boolean*; - operator care returnează **true** dacă și numai dacă *q* este o coadă vidă.

- 
- Operatorul **Inlocuiește** constă dintr-o **inserție** urmată de **suprimarea** celui mai prioritar element.
    - Este o operație diferită de succesiunea suprimare-inserție deoarece necesită creșterea pentru moment a dimensiunii cozii cu un element.
    - Acest operator se definește separat deoarece în anumite implementări poate fi conceput foarte eficient.
  - În mod analog operatorul **Schimbă** poate fi implementat ca și o **suprimare**, urmată de o **inserție**, iar generarea unei cozi ca și o succesiune de operatori **Inserează**.

- **Cozile bazate pe prioritate** pot fi în general implementate în diferite moduri unele bazate pe structuri simple, altele pe structuri de date avansate, fiecare presupunând însă performanțe diferite pentru operatorii specifici.
- În continuare vor fi prezentate unele dintre aceste posibilități.

#### 6.5.5.2. Implementarea cozilor bazate pe prioritate cu ajutorul tablourilor

- **Implementarea unei cozi bazate pe prioritate** se poate realiza prin memorarea elementelor cozii într-un **tablou neordonat**.
- În secvența [6.5.5.2.a] apare structura de date corespunzătoare acestei abordări iar în secvența [6.5.5.2.b] implementarea operatorilor **Inseareaza** și **Extrage**.

-----  
{Implementarea cozilor bazate pe prioritate cu tablouri - structuri de date}

```

TYPE TipElement = RECORD
    prioritate: TipPrioritate;
    info: TipInfo
END;

```

[6.5.5.2.a]

```

    TipCoadaBazataPePrioritate = RECORD
        elemente: ARRAY[1..DimMax] OF TipElement;
        nrElemente: 0..DimMax
    END;

```

-----  
/\* Implementarea cozilor bazate pe prioritate cu tablouri - structuri de date - C \*/

```

typedef struct tip_element {
    tip_prioritate prioritate;
    tip_info info;
} tip_element;

```

/\*[6.5.5.2.a]\*/

```

typedef struct tip_coada_bazata_pe_prioritate {
    tip_element elemente[dim_max];
    unsigned char nr_elemente;
} tip_coada_bazata_pe_prioritate;

```

-----  
{Implementarea cozilor bazate pe prioritate cu tablouri - operatorii Inseareaza și Extrage}

```

PROCEDURE Inseareaza(x: TipElement,
                    q: TipCoadaBazataPePrioritate);
BEGIN {O(1)}
    q.nrElemente:= q.nrElemente+1;
    q.elemente[nrElemente]:= x
END; {Inseareaza}

```

```

FUNCTION Extrage(q: TipCoadabazataPePrioritate):
TipElement;

    VAR j,max: 1..DimMax;
    BEGIN {O(n)} [6.5.5.2.b]
        max:= 1;
        FOR j:=1 TO q.nrElemente DO
            IF q.elemente[j].prioritate >
                q.elemente[max].prioritate THEN max:= j;
        Extrage:= q.elemente[max];
        q.elemente[max]:= q.elemente[nrElemente];
        q.nrElemente:= q.nrElemente-1
    END; {Extrage}

-----
/* Implementarea cozilor bazate pe prioritate cu tablouri -
operatorii Insereaza și Extrage */

void insereaza(tip_element x, tip_coadabazata_pe_prioritate
q)
{
    /*01*/
    q.nr_elemente= q.nr_elemente+1;
    q.elemente[q.nr_elemente-1]= x;
}/*insereaza*/

tip_element extrage(tip_coadabazata_pe_prioritate q)
{
    unsigned char j,max; /*O(n)*/ [6.5.5.2.b]
    tip_element extrage_result;
    max= 1;
    for( j=1; j <= q.nr_elemente; j ++)
        if (q.elemente[j-1].prioritate >
            q.elemente[max-1].prioritate) max= j;
    extrage_result= q.elemente[max-1];
    q.elemente[max-1]= q.elemente[q.nr_elemente-1];
    q.nr_elemente= q.nr_elemente-1;
    return extrage_result;
}/*extrage*/

/*-----*/

```

- Pentru **inserție** se incrementează nrElemente și se adaugă elementul de inserat pe ultima poziție a tabloului q.elemente, operație care este constantă în timp ( $O(1)$ ).
- Pentru **extragere** se baleează întreg tabloul găsiindu-se cel mai mare element, apoi se introduce ultimul element pe poziția celui care a fost extras și se decrementează nrElemente. Operatorul necesită o regie  $O(n)$ .
- Implementarea lui **Inlocuieste** este similară, ea presupunând o inserție urmată de o extragere.
- Redactarea celorlalți operatori în acest context nu ridică nici un fel de probleme.

- În implementarea **cozilor bazate pe prioritate** se pot utiliza și **tablouri ordonate**.
  - Elementele cozii sunt păstrate în tablou în **ordinea crescătoare a priorităților**.
  - În aceste condiții, operatorul **Extrage**, returnează **ultimul element** `q.elemente[nrElemente]` și decrementează `nrElemente`, operație ce durează un interval constant de timp.
  - Operatorul **Insereaza** presupune mutarea spre dreapta a elementelor mai mari ca și elementul de inserat, operație care durează  $O(n)$ .
- Operatorii care se referă la **cozi bazate pe prioritate** pot fi utilizați în implementarea unor **algoritmi de sortare**. Spre exemplu:
  - (1) Se aplică în mod **repetat** operatorul **Insereaza** pentru a introduce elemente într-o **coadă bazată pe priorități**.
  - (2) Se **extrag** pe rând elementele cu operatorul **Extrage** până la golirea cozii.
  - (3) Se obține **secvența sortată** a elementelor în **ordinea descrescătoare a priorității lor**.
- Dacă se utilizează o **coadă bazată pe prioritate** reprezentată ca un **tablou neordonat** se obține **sortarea prin selecție**.
- Dacă se utilizează o **coadă bazată pe prioritate** reprezentată ca un **tablou ordonat**, se obține **sortarea prin inserție**.

#### 6.5.5.3. Implementarea cozilor bazate pe prioritate cu ajutorul listelor înlănțuite

- În implementarea cozilor bazate pe prioritate pot fi utilizate și **listele înlănțuite** în variantă **ordonată** sau **neordonată**.
- Această abordare **nu** modifică principal implementarea operatorilor specifici.
  - Ea face însă posibilă realizarea mai **eficientă** a **fazelor de inserție și suprimare** a elementelor cozii datorită flexibilității listelor înlănțuite.
- În continuare se prezintă un **exemplu** de realizare a operației **Extrage**, utilizând pentru implementarea cozilor bazate pe prioritate **listele înlănțuite neordonate**.
- Se utilizează următoarele structuri de date [6.5.5.3.a].

---

{Implementarea cozilor bazate pe prioritate cu liste înlănțuite neordonate - structuri de date}

**TYPE** TipReferintaNod = ^TipNod;





```

        cu cheia cea mai mare }
        mare:= curent^urm^.element
    END; {IF}
    curent:= curent^.urm
    END; {WHILE}
    Extrage:= anterior^.urm ; {poziționează
        pointerul pe cel mai mare element}
    curent:= anterior^.urm;
    anterior^.urm:= anterior^.urm^.urm; {suprimare}
    DISPOSE(curent)
    END; {ELSE}
END; {Extrage}
-----
/* Implementarea cozilor bazate pe prioritate cu liste
înlănțuite neordonate - operatorul Extrage - C */

#include <stdlib.h>

tip_referinta_nod extrage(coada_bazata_pe_prioritate* q){
    tip_referinta_nod curent; /*nodul din fata celui baleat*/
    tip_element mare; /*valoarea celui mai mare element*/
    tip_referinta_nod anterior; /*nodul anterior celui mai
        mare*/
    tip_referinta_nod extrage_result;
    if (vid(*q))
        printf("coada este vida"); /*[6.5.5.3.b]*/
    else{
        mare= (*q)->urm->element; /*q indica nodul fictiv*/
        anterior= *q; curent= (*q)->urm;
        while (curent->urm != NULL){/*compară valoarea lui
            mare cu valoarea următoare elementului curent*/
            if (curent->urm->element > mare){
                anterior= curent; /*anterior reține pointerul
                nodului ce precede nodul cu cheia cea mai mare */
                mare= curent->urm->element;
            }/*if*/
            curent= curent->urm;
        }/*while*/
        extrage_result= anterior->urm; /*poziționează
            pointerul pe cel mai mare element*/
        curent= anterior->urm;
        anterior->urm= anterior->urm->urm;
    }/*else*/
    return extrage_result;
}/*extrage*/
/*-----*/

```

- Implementarea celorlalte funcții referitoare la cozile bazate pe prioritate utilizând liste neordonate nu ridică nici un fel de probleme.
- Aceste **implementări simple** ale **cozilor bazate pe prioritate** în multe situații sunt mai utile decât cele bazate pe modele sofisticate.

- Astfel, implementarea bazată pe **liste neordonate** e potrivită în situațiile în care se fac multe inserții și mai puține extrageri.
- În schimb, implementarea bazată pe **liste ordonate** este potrivită dacă prioritățile elementelor care se inserează au tendința de a fi apropiate ca valoare de prioritatea maximă.

#### 6.5.5.4. Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor

- Un **ansamblu** este un **arbore binar parțial ordonat** reprezentat printr-un **tablou liniar**, ale cărui elemente satisfac condițiile ansamblului (&3.2.5).
  - În consecință, cea mai mare (prioritară) cheie este poziționată întotdeauna pe **prima poziție** a tabloului care materializează **ansamblul**.
- Algoritmii care operează asupra structurilor de date **ansamblu** necesită în cel mai defavorabil caz  $O(\log_2 N)$  pași.
- **Ansamblele** pot fi utilizate în implementarea **cozilor bazate pe prioritate**.
- Există algoritmi care prelucrează ansamblul de **sus în jos**, alții care îl prelucrează de **jos în sus**.
- Se presupune că referitor la elementele **cozii** (ansamblului):
  - (1) Fiecare element are asociată o **prioritate**.
  - (2) Elementele sunt memorate într-un **tablou** ansamblu cu dimensiunea maximă precizată (DimMax).
  - (3) **Dimensiunea curentă** a tabloului este păstrată în variabila nrElemente care face parte din definiția **cozii bazate pe prioritate** [6.5.5.4.a].

-----  
 {Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor - structuri de date}

```

TYPE TipElement = RECORD
    prioritate: TipPrioritate;
    info: TipInfo                                     [6.5.5.4.a]
END;

TipIndice = 1..DimMax;

TipCoadabazataPePrioritate = RECORD
    ansamblu: ARRAY[TipIndice] OF TipElement;
    nrElemente: 0..DimMax
END;
  
```

-----  
 /\* Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor - structuri de date - C \*/

```

typedef struct tip_element {
    tip_prioritate prioritate;
    tip_info info;                                /*[6.5.5.4.a]*/
} tip_element;

typedef unsigned char tip_indicel;

typedef struct tip_coada_bazata_pe_prioritate {
    tip_element ansamblu[dimmax];
    unsigned char nr_elemente;
} tip_coada_bazata_pe_prioritate;
/*-----*/

```

- Pentru **construcția** unui ansamblu se utilizează de regulă operatorul **Insereaza**.
  - În capitolul 3 s-a studiat posibilitatea **extinderii** ansamblului spre **stânga** prin plasarea elementului de inserat în vârful ansamblului și deplasarea sa spre baza acestuia până la locul potrivit, prin interschimbare cu fiul cel mai prioritar (operatorul numit **Deplasare**). Mai corect, un astfel de operator ar trebui să se numească **DownHeap**.
  - O altă posibilitate de realizare a **inserției** o reprezintă **extinderea** spre **dreapta** a ansamblului, prin introducerea elementului de inserat pe **ultima** sa poziție.
  - Această operație poate viola însă **regulile ansamblului** dacă **noul element** introdus are prioritatea mai mare ca și părintele său.
    - În această situație se realizează **avansul în sus** al elementului în **ansamblu** prin **interschimbare** cu părintele său.
    - Procesul se repetă până când prioritatea elementului introdus devine **mai mică** ca și **prioritatea părintelui său**, sau a ajuns pe **prima** poziție a ansamblului.
- Procedura **UpHeap** prezentată în secvența [6.5.5.4.b] implementează această operator respectiv **avansul** elementului nou introdus de pe **ultima poziție a ansamblului**, de **jos în sus** până la locul potrivit.
  - Această este metoda inversă celei implementate de către procedura **Deplasare** utilizată la sortarea "heap sort" (§3.2.5).
  - **Ansamblul** se prelungește spre stânga cu poziția 0 care **nu aparține ansamblului** dar care este utilizată pe post de **fanion** în procesul de ascensiune.
- Alături de procedura **UpHeap** apare și procedura care implementează inserția propriu-zisă (**Insertie**).

---

{Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor - operatorii UpHeap și Insereaza}

```

PROCEDURE UpHeap(q: CoadabazataPePrioritate,
                  k: TipIndice);
VAR v: TipElement;
BEGIN
    v:= q.ansamblu[k]; {elementul nou introdus este pe
                       poziția k}
    q.ansamblu[0].prioritate:=(cea mai mare prioritate)+1;
                                {poziția 0 pe post de fanion}
    WHILE q.ansamblu[k DIV 2].prioritate <=
                                v.prioritate DO {look ahead}
        BEGIN
            q.ansamblu[k]:= q.ansamblu[k DIV 2];
            k:= k DIV 2
        END;
    q.ansamblu[k]:= v
END; {UpHeap}

```

```

PROCEDURE Insertie(x: TipElement,
                   q: CoadabazataPePrioritate);
BEGIN
    q.nrElemente:= q.nrElemente+1;
    q.elemente[q.nrElemente]:= x;
    UpHeap(q,q.nrElemente)
END; {Insertie}

```

---

```

/* Implementarea cozilor bazate pe prioritate cu ajutorul
ansamblelor - operatorii UpHeap și Insereaza - C*/

```

```

void upheap(tip_coadabazata_pe_prioritate q,
             tip_indice k){
    tip_element v;
    v= q.ansamblu[k-1];
    q.ansamblu[0].prioritate=(/*cea mai mare prioritate*/)+1;
                                /*poziția 0 pe post de fanion*/
    while (q.ansamblu[k/2].prioritate <= v.prioritate)
        /*look ahead*/
        {
            q.ansamblu[k]= q.ansamblu[k/2];
            k= k/2;
        }
    q.ansamblu[k]= v;
    /*[6.5.5.4.b]*/
} /*upheap*/

```

```

void insertie(tip_element x,
               tip_coadabazata_pe_prioritate q){
    tip_element v;
    q.nr_elemente= q.nr_elemente+1;
    q.ansamblu[q.nr_elemente]= x;
    upheap(q,q.nr_elemente);
} /*insertie*/
/*-----*/

```

- Dacă în operatorul **UpHeap**,  $(k \text{ DIV } 2)$  se înlocuiește cu  $(k-1)$  se obține în esență **sortarea prin inserție**.
  - În acest caz, găsirea locului în care se inserează noul element se realizează verificând și deplasând **secvențial** elementele cu câte o poziție spre dreapta.
- În procedura **UpHeap** deplasarea se face **nu** liniar (secvențial) ci din **nivel în nivel** de-a lungul ansamblului.
- La fel ca la sortarea prin inserție, **interschimbarea nu** este totală,  $v$  fiind implicat mereu în această operație.
- Poziția 0 a ansamblului  $q.elemente[0]$  se utilizează pe post de **fanion** care se asignează inițial cu o prioritate mai **mare** decât a oricărui alt element.
- Operatorul **Inlocuieste** presupune înlocuirea celui mai prioritar element, adică cel situat în rădăcina ansamblului, cu un nou element care **se deplasează de sus în jos** în ansamblu până la locul potrivit în concordanță cu definiția ansamblului.
- Operatorul **Extrage** presupune:
  - (1) Extragerea elementului cel mai prioritar (situat pe poziția  $q.elemente[1]$ ).
  - (2) Introducerea lui  $q.elemente[q.nrElemente]$  (ultimul element al ansamblului) pe prima poziție.
  - (3) Decrementarea numărului de elemente ( $q.nrElemente$ ).
  - (4) Deplasarea primului element de **sus în jos**, spre baza ansamblului, până la locul potrivit.
- Deplasarea de sus în jos în ansamblu de la poziția  $k$  spre baza acestuia este materializată de operatorul **DownHeap** [6.5.5.4.c].

-----  
 {Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor - operatorul DownHeap}

```
PROCEDURE DownHeap(q: CoadăBazatăPePrioritate,
                   k: TipIndice);
```

```
  VAR j: TipIndice;
      v: TipElement;
      ret: boolean;
```

```
  BEGIN
```

```
    v:= q.elemente[k];
```

```
                                [6.5.5.4.c]
```

```
    ret:= false;
```

```
    WHILE(k < q.nrElemente DIV 2) AND (NOT ret) DO
```

```
      BEGIN
```

```
        j:= k+k;
```

```
        IF j < q.nrElemente THEN
```

```

        IF q.elemente[j].prioritate <
            q.elemente[j+1].prioritate THEN j:= j+1;
    IF v.prioritate >= q.elemente[j].prioritate THEN
        ret:= true
    ELSE
        BEGIN
            q.elemente[k]:= q.elemente[j]; k:= j
        END
    END; {WHILE}
    q.elemente[k]:= v
END; {DownHeap}
-----
/* Implementarea cozilor bazate pe prioritate cu ajutorul
ansamblelor - operatorul DownHeap -C */

void downheap(tip_coada_bazata_pe_prioritate q,
              tip_indice1 k){
    tip_indice1 j;
    tip_element v;
    boolean ret;
    v= q.ansamblu[k];                      /*[6.5.5.4.c]*/
    ret= false;
    while((k < q.nr_elemente/2) && (! ret))
    {
        j= k+k;
        if (j < q.nr_elemente)
            if (q.ansamblu[j].prioritate <
                q.ansamblu[j+1].prioritate) j= j+1;
        if (reccmp(v.prioritate, q.ansamblu[j].prioritate)
            >= 0)
            ret= true;
        else
        {
            q.ansamblu[k]= q.ansamblu[j]; k= j;
        }
    } /*while*/
    q.ansamblu[k]= v;
} /*downheap*/
/*-----*/

```

- Deplasarea în ansamblu se realizează interschimbând elementul de pe poziția curentă k cu cel mai prioritar dintre fiii săi și avansând (coborând) pe nivelul următor.
- Procesul continuă până când elementul din k devine mai prioritar decât oricare dintre fiii săi sau s-a ajuns la baza ansamblului.
- Ca și în situația anterioară **nu** este necesară interschimbarea completă întrucât v este implicat tot timpul.
- Bucla **WHILE** este prevăzută cu două ieșiri:

- Una corelată cu atingerea **bazei ansamblului** ( $k > q.nrElemente \text{ DIV } 2$ ).
- A doua corelată cu **găsirea poziției** elementului de inserat în interiorul ansamblului (variabila booleană *ret*).
- Cu aceste precizări, implementarea operatorilor **Extrage** și **Inlocuieste** este imediată [6.5.5.4.d].

-----  
 {Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor - operatorii Extrage și Inlocuieste}

```

FUNCTION Extrage(q: CoadaBazataPePrioritate): TipElement;
BEGIN
  Extrage:= q.elemente[1];
  q.elemente[1]:= q.elemente[q.nrElemente];
  q.NrElemente:= q.NrElemente-1;
  DownHeap(q,1)
END; {Extrage}                                [6.5.5.4.d]
  
```

```

FUNCTION Inlocuieste(q: CoadaBazataPePrioritate
                      x: TipElement): TipElement;
BEGIN
  q.elemente[0]:= x;
  DownHeap(q,0);
  Inlocuieste:= q.elemente[0]
END; {Inlocuieste}
  
```

-----  
 /\* Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor - operatorii Extrage și Inlocuieste - C \*/

```

tip_element extrage(tip_coada_bazata_pe_prioritate q)
{
  tip_element extrage_result;
  extrage_result= q.ansamblu[1];
  q.ansamblu[1]= q.ansamblu[q.nr_elemente];
  q.nr_elemente= q.nr_elemente-1;
  downheap(q,1);
  return extrage_result;
} /*extrage*/                                /*[6.5.5.4.d]*/
  
```

```

tip_element inlocuieste(tip_coada_bazata_pe_prioritate q,
                       tip_element x)
{
  tip_element inlocuieste_result;
  q.ansamblu[0]= x;
  downheap(q,0);
  inlocuieste_result= q.ansamblu[0];
  return inlocuieste_result;
} /*inlocuieste*/
/*-----*/
  
```



- În cazul operatorului **Inlocuieste** se utilizează și poziția  $q.elemente[0]$  ai cărei fii sunt pozițiile 0 (ea însăși) și 1.
  - Astfel dacă  $x$  este mai prioritar decât oricare element al ansamblului, ansamblul rămâne nemodificat, altfel  $x$  este deplasat în ansamblu.
  - În toate situațiile este returnat  $q.elemente[0]$  în calitate de cel mai prioritar element.
- Operatorii **Suprima** și **Schimba** pot fi implementați utilizând combinații simple ale metodelor de mai sus.
  - Spre exemplu dacă prioritatea elementului situat pe poziția  $k$  este ridicată atunci poate fi apelată procedura **UpHeap**( $q, k$ ), iar dacă prioritatea sa este coborâtă, procedura **DownHeap**( $q, k$ ) rezolvă situația.

#### 6.5.5.5. Tipul de date abstract ansamblu

- Se reiterează observația că, din nou, conceptul de **ansamblu (heap)** a fost asimilat cu un **tip de date abstract**, utilizat de această dată drept suport pentru implementarea **cozilor bazate pe prioritate**.
- În consecință, ne propunem să definim un **TDA Ansamblu**.
- **TDA Ansamblu** constă din **modelul matematic** descris de un **arbore binar parțial ordonat** peste care se definesc **operatorii** de construcție (extensie) ai ansamblului **UpHeap**, și **DownHeap**, operatorul de extragere **Extrage**, operatorul **DimensiuneAnsamblu** precum și un operator **InitializeazăAnsamblu** care creează ansamblul vid.
- În sinteză o posibilă variantă de **TDA Ansamblu** apare în secvența [6.5.5.5.a] .

---

#### TDA Ansamblu

**Modelul matematic:** un arbore binar parțial ordonat, implementat cu ajutorul unei structuri tablou liniar specifice. Elementele ansamblului aparțin unui același tip numit **tip de bază**.

#### **Notății:**

$a$ : TipAnsamblu; [6.5.5.5.a]  
 $x$ : TipElement;  
 $n$ : intreg;  
 $p$ : TipPoziție

#### **Operatori:**

1.  $a = \text{InitializeazăAnsamblu:TipAnsamblu};$  - operator care inițializează pe  $a$  ca ansamblu vid, și face dimensiunea ansamblului egală cu 0.
  2.  $\text{UpHeap}(a:\text{TipAnsamblu}, x:\text{TipElement}, p:\text{TipPozitie});$  - extensia spre dreapta a ansamblului  $a$ . Operator care introduce elementul  $x$  pe poziția  $p$  (ultima sa poziție) și îl deplasează în ansamblu de jos în sus până la poziția specifică priorității, sau în vârful acestuia dacă prioritatea lui  $x$  este maximă. Incrementează dimensiunea ansamblului.
  3.  $\text{DownHeap}(a:\text{TipAnsamblu}, x:\text{TipElement});$  - extensia spre stânga a ansamblului  $a$ . Operator care introduce elementul  $x$  în vârful ansamblului (pe prima sa poziție) și îl deplasează în ansamblu de sus în jos până la poziția specifică priorității, sau până la baza acestuia dacă prioritatea lui  $x$  este minimă. Incrementează dimensiunea ansamblului.
  4.  $x = \text{Extrage}(a:\text{TipAnsamblu}):\text{TipElement};$  - operator care extrage și furnizează în  $x$  cel mai prioritar element al ansamblului  $a$  reducând cu o unitate dimensiunea acestuia. În continuare, ansamblul este restructurat, astfel încât în vârful său să ajungă cel mai prioritar dintre elementele rămase în urma extracției.
  5.  $n = \text{DimensiuneAnsamblu}(a:\text{TipAnsamblu});$  - operator care furnizează dimensiunea curentă a ansamblului  $a$ .
- 

- După cum s-a precizat, operatorii care acționează asupra **TDA Ansamblu** se încadrează cu toții în performanța  $O(\log_2 n)$  unde  $n$  este dimensiunea ansamblului.
  - Excepție fac operatorii **InițializeazăAnsamblu** și **Dimensiune Ansamblu** care sunt  $O(1)$ .
- Un exemplu de definire și implementare a unui **ansamblu** conceput ca și un arbore **binar parțial ordonat**, reprezentat printr-o **structură tablou** s-a realizat în capitolul 3, subcapitolul (§ 3.2.5).
- Modalități de implementare a unora dintre operatorii specifici au fost prezentate atât în subcapitolul mai sus menționat (§3.2.5) cât și pe parcursul acestui subcapitol (vezi secvențele [6.5.5.4.b] și [6.5.5.4.c]).
- Funcție de aplicație pot fi definiți și alți operatori pentru **TDA Ansamblu**, cum ar fi spre exemplu operatorii referitori la verificarea sau schimbarea **priorității elementelor**. Un astfel de exemplu va fi prezentat în volumul II al acestui manual în partea referitoare la grafuri.

## 6.6. Structura de date multilistă

- Se numește **multilistă**, o structură de date ale cărei noduri conțin **mai multe câmpuri de înlănțuire**.
  - Cu alte cuvinte, un nod al unei astfel de structuri poate aparține în același timp la **mai multe liste înlănțuite simple**.
- În literatura de specialitate termenii consacrați pentru a desemna o astfel de structură sunt:
  - "Braid".
  - "Multilist".
  - "Multiply linked list" [De89].
- În figura 6.6.a apare o reprezentare grafică a unei **structuri multilistă** iar în secvența [6.6.a] se prezintă un **exemplu de definire** a unei astfel de structuri.

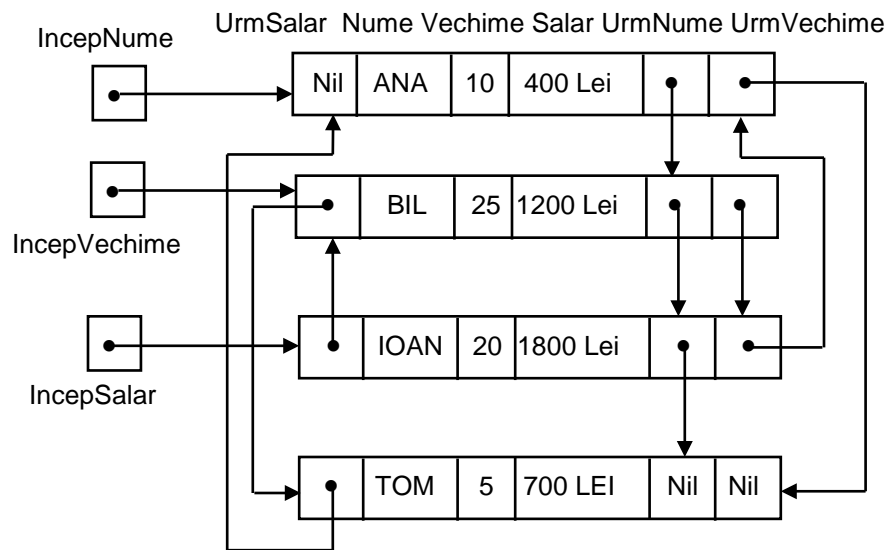


Fig.6.6.a. Exemplu de structură de date multilistă

-----  
{Structura multilistă - definirea structurilor de date}

```
TYPE TipNume = STRING[20];
```

```
TipInfo = RECORD  
    Nume: TipNume;  
    Vechime: integer;  
    Salar: real
```

[ 6.6.a]

```

    END;

    TipReferintaNod = ^TipNod;

    TipNod = RECORD
        Informație: TipInfo;
        UrmSalar,UrmNume,UrmVechime: TipReferintaNod
    END;

VAR incepNume,incepVechime,incepSalarii: TipReferintaNod;
-----
/* Structura multilistă - definirea structurilor de date - C
*/

typedef char* tip_nume;

typedef struct tip_infol {
    tip_nume nume;
    int vechime;
    float salar;                                /*[6.6.a]*/
} tip_infol;

typedef struct tip_nod* tip_referinta_nod;

typedef struct tip_nod {
    tip_infol informatie;
    tip_referinta_nod urm_salar,urm_nume,urm_vechime;
}tipnod;

tip_referinta_nod incep_nume,incep_vechime,incep_salarii;
/*-----*/

```

- **Avantajul** utilizării unor astfel de structuri este evident.
  - Prezența mai **multor înlănțuiri** într-un același nod, respectiv **apartenența simultană** a aceluiași nod la mai multe liste, asigură acestei structuri de date o **flexibilitate** deosebită.
- Acest avantaj, coroborat cu o manipulare relativ **facilă** specifică structurilor înlănțuite este exploatat la implementarea **bazelor de date**, cu precădere a celor relaționale.
- Aria de utilizare a structurilor multilistă este însă mult mai extinsă.
  - Spre exemplu, o astfel de structură poate fi utilizată cu succes la memorarea **matricelor rare**.
    - Este cunoscut faptul că **matricele rare** sunt matrice de mari dimensiuni care conțin de regulă un **număr redus** de elemente restul fiind poziționate de obicei pe zero.
    - Din acest motiv memorarea lor în forma obișnuită a tablourilor bidimensionale presupune o **risipă** mare de memorie.

- Spre a evita această risipă se poate utiliza **structura multilistă** din secvența [6.6.b].
  - În această structură, fiecărui **element valid** al matricei  $i$  se asociază un nod al structurii.
  - În nod se memorează:
    - Indicii **elementului curent**.
    - **Înlănțuirile la proximal element valid** din același rând respectiv la proximal element valid situat pe aceeași coloană.

---

**{Aplicație la structura multilistă - memorarea matricilor rare}**

```
CONST  randMax = ...;
        colMax  = ...;
```

```
TYPE    TipReferintaNod = ^TipNod;
```

```
        TipNod = RECORD
            rand: 1..randMax;
            coloana: 1..colMax;
            info: ...;
            dreapta, jos: TipReferintaNod;
        END;
```

[6.6.b]

```
VAR matrice: TipReferintaNod;
```

---

**/\* Aplicație la structura multilistă - memorarea matricilor rare - C \*/**

```
enum { ran_dmax = 10,
        col_max = 10};
```

```
typedef struct tip_nod* tip_referinta_nod;
```

```
typedef struct tip_nod {
    unsigned rand;
    unsigned coloana;
    int info;
    tip_referinta_nod dreapta, jos;
} tip_nod;
/*[6.6.b]*/
```

```
tip_referinta_nod matrice;
```

```
/*-----*/
```

## 6.7. Liste generalizate

- Singura structură de date definită în limbajul de programare **LISP** este **lista**.

- După cum se cunoaște limbajul LISP este destinat **manipulării simbolurilor** fiind utilizat cu precădere în domeniul **Inteligenței Artificiale**.
- Unitatea structurală fundamentală definită în **LISP** este **atomul** care este conceput ca un șir de caractere și / sau cifre.
- În acest context, o **listă** este o mulțime de paranteze conținând orice număr de **atomi** și **liste**. Spre exemplu:
  - $B = (b)$  reprezintă în LISP o listă care conține un singur element.
  - $L = (a, b, (c, d, e), f)$  reprezintă o listă cu o structură mai complexă, care conține o sublistă.
- Atomii și listele sunt memorate în LISP cu ajutorul unei structuri de date speciale numită **listă generalizată**.
- Nodurile unei **liste generalizate** conțin trei câmpuri:
  - (1) Atom - câmp boolean.
  - (2) Un câmp dependent de valoarea câmpului Atom .
    - Dacă Atom este **TRUE** atunci acest câmp se numește info și memorează o informație adică un **atom**.
    - Dacă Atom este **FALSE** câmpul se numește lista și memorează o **referință la o listă**;
  - (3) Urm – câmp de înlănțuire.
- În figura 6.7.a apar reprezentările grafice ale atomului 'a' și ale listelor B și L definite anterior, în accepțiunea acestei definiții.
- Pentru o mai bună înțelegere a acestui concept, în secvența [6.7.a] apare specificarea PASCAL respectiv C a structurii de date ListăGeneralizată.
  - Din cauza restricției de implementare a articolului cu variante în aceste limbaje, implementare care impune plasarea variantelor la sfârșitul articolului, câmpul de înlănțuire Urm apare în declarația nodului la începutul acestuia.

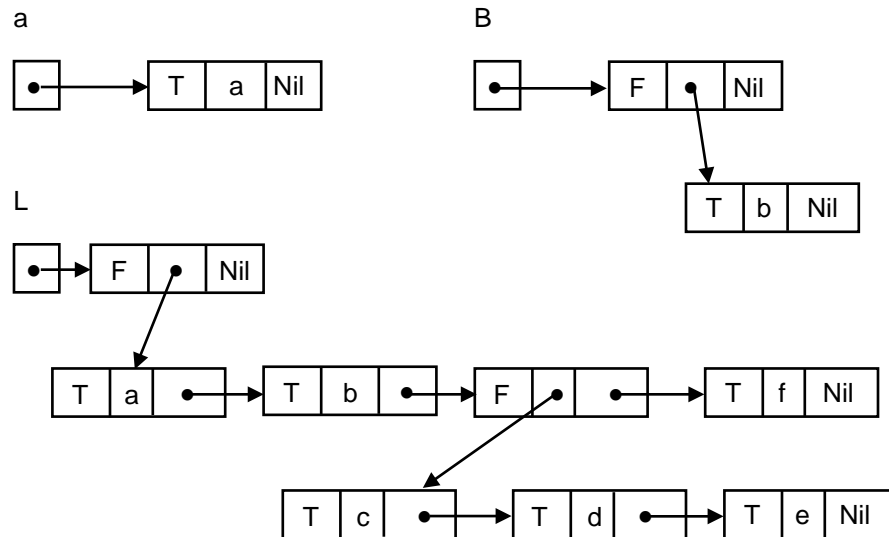


Fig.6.7.a. Exemple de liste generalizate

---

{Liste generalizate - definirea structurilor de date -  
(varianta Pascal)}

**TYPE** TipListaGeneralizata = ^TipNod;

TipNod = **RECORD**

Urm: TipListaGeneralizat;

[6.7.a]

**CASE** Atom: boolean **OF**

**TRUE**:(info: CHAR);

**FALSE**: (lista: TipListaGeneralizata)

**END**;

**VAR** a,B,L: TipListaGeneralizata;

---

/\* Liste generalizate - definirea structurilor de date -  
(varianta C)\*/

**typedef** unsigned boolean;

#define true (1)

#define false (0)

**typedef struct** tip\_nod\* tip\_lista\_generalizata;

**typedef struct** tip\_nod{ /\*[6.7.a]\*/  
tip\_lista\_generalizata urm;  
boolean atom;  
**union**{  
char info;  
tip\_lista\_generalizata lista;  
}u;  
}tip\_nod;

tip\_lista\_generalizata a,b,l;

### 6.7.1. Tipul de date abstract Listă Generalizată

- Definirea **TDA Listă Generalizată** presupune conform uzanțelor acestui manual precizarea modelului matematic, a notațiilor și a operatorilor specifici.
- Acest lucru este realizat în [6.7.1.a].

#### TDA Listă Generalizată

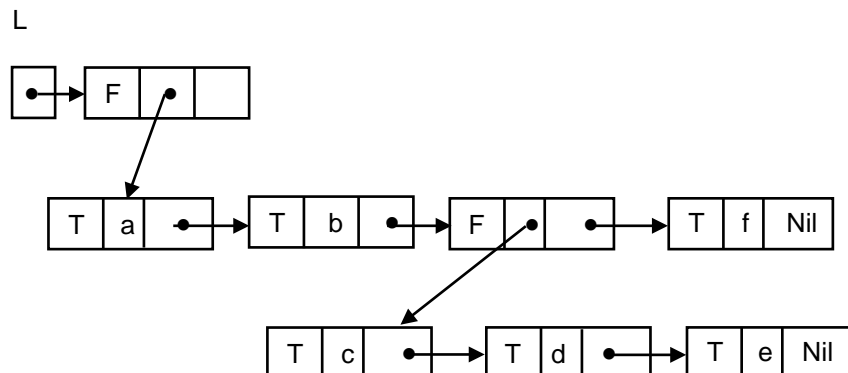
**Modelul matematic:** o secvență finită de noduri. Toate nodurile aparțin unui același tip numit tip de bază. Fiecare nod conține trei câmpuri: (1) un câmp boolean - Atom; (2) un câmp dependent de valoarea câmpului Atom: dacă Atom este TRUE atunci acest câmp se numește info și memorează o informație adică un atom, dacă Atom este FALSE câmpul se numește lista și memorează o referință la o listă; (3) un câmp de înlănțuire - Urm.

#### Notații:

*lista: TipListaGeneralizata;* [6.7.1.a]

#### Operatori:

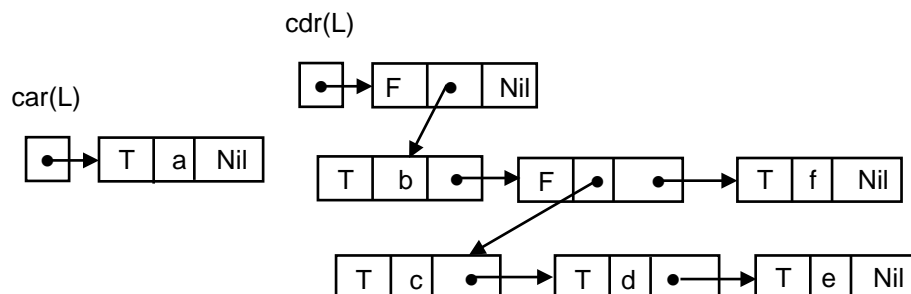
- car** (*lista:TipListaGeneralizata*):*TipListaGeneralizata*; - operator care returnează **primul element al listei** (care poate fi atom sau listă).
- cdr** (*listă:TipListaGeneralizata*):*TipListaGeneralizata*; - operator care returnează **restul listei**, adică ceea ce rămâne după înlăturarea primului element.
- cons** (*M,N:TipListaGeneralizata*):*TipListaGeneralizata*; - operator care realizează construcția unei liste în care M este primul element iar N restul listei.





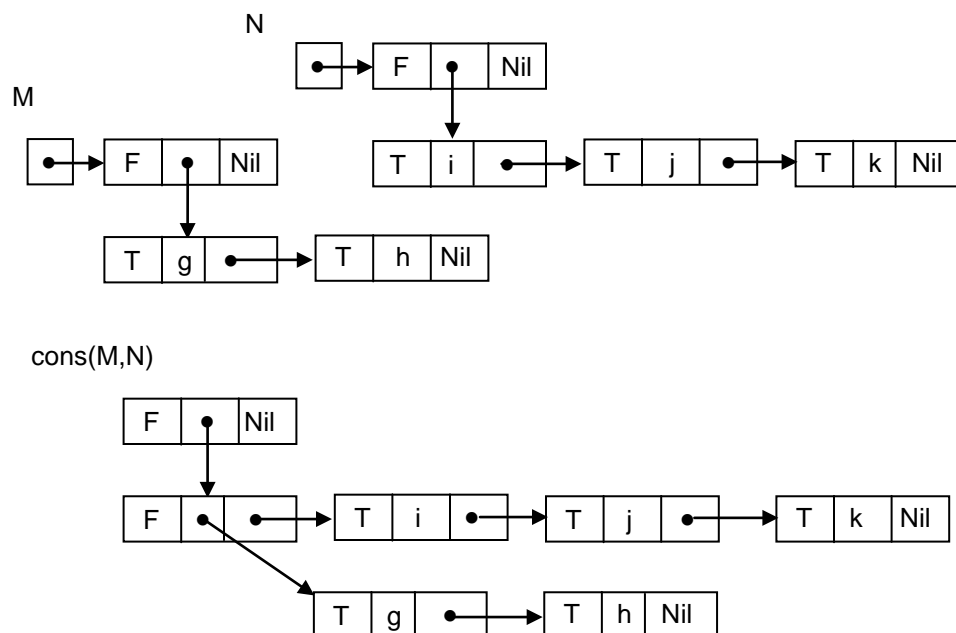
**Fig.6.7.1.a.** Exemplu de listă generalizată

- Spre **exemplu** pentru lista generalizată  $L = (a, b, (c, d, e), f)$  din figura 6.7.1.a, operatorii **car**(L)=a și **cdr**(L)=(b, (c, d, e), f) sunt vizualizați în figura 6.7.1.b.



**Fig.6.7.1.b.** Operatorii car și cdr aplicați asupra listei generalizate L din fig.6.7.1.a

- În mod asemănător, pentru listele generalizate  $M = (g, h)$  și  $N = (i, j, k)$ , aplicarea operatorului **cons** conduce la lista generalizată **cons**(M,N)=( (g,h), i, j, k ) care apare reprezentată grafic în fig. 6.7.c.



**Fig.6.7.1.c.** Exemplu de utilizare a operatorului cons

## 6.8. Tipul de date abstract asociere

- **Mapping-ul** sau **asocierea memoriei** este o **funcție** definită pe mulțimea elementelor unui tip denumit **tip domeniu** cu valori în mulțimea elementelor unui alt tip (eventual același) numit **tip valoare** (codomeniu).
  - De fapt o **asociere**  $M$ , pune în **corespondență** un element  $V$  aparținând **tipului valoare** cu un element  $D$  al **tipului domeniu** prin **relația**  $M(D) = V$ .
- Anumite asocieri, ca spre exemplu  $PATRAT(i) = i^2$ , pot fi implementate simplu cu ajutorul unor **funcții** care precizează **expresia aritmetică de calcul**, sau prin alte mijloace simple, de determinare a lui  $M(D)$  pornind de la  $D$ .
- În multe situații însă, **nu** este posibilă descrierea asocierii  $M(D)$  decât prin memorarea valorii  $V$  corespunzătoare pentru fiecare  $D$  în parte.
- În cadrul acestui paragraf, se vor face în continuare referiri la metodele de **implementare** ale unor astfel de asocieri.
- Pentru început se va preciza **tipul de date abstract asociere**.

### 6.8.1. TDA Asociere

- În cadrul unei **asocieri**  $M$ , fiind dat un element  $D$  aparținând lui **TipDomeniu** se pot defini următorii operatori:
  - (1) Un operator **Initializează** care realizează inițializarea domeniului valorilor cu asocierea vidă, adică cea a cărei domeniu este vid.
  - (2) Un operator **Atribuie** care permite **introducerea unui nou element** în asocierea  $M$  sau **modificarea valorii** lui  $M(D)$ , precizând valorile corespunzătoare din **domeniul**, respectiv **codomeniul** valorilor.
  - (3) Un operator boolean **Determină** care:
    - (a) Verifică dacă  $M(D)$  **este definit** sau **nu** pentru un  $D$  dat.
    - (b) În caz afirmativ **returnează** valoarea adevărat și furnizează valoarea lui  $M(D)$ .
    - (c) În caz negativ **returnează** valoarea fals.
- Acești operatori sunt precizați sintetic în secvența [6.8.1.a] care descrie **TDA Asociere**.

---

#### TDA Asociere

**Modelul matematic:** o funcție (algebrică) definită pe

mulțimea elementelor unui tip denumit *TipDomeniu* și cu valori în mulțimea elementelor unui alt tip denumit *TipValoare*.

**Notății:**

*M*: *TipAsociere*; [6.8.1.a]  
*D*: *TipDomeniu*;  
*V*: *TipValoare*;

**Operatori:**

1. **Initializează**(*M*: *TipAsociere*); - face asocierea *M* vidă.

2. **Atribuie**(*M*: *TipAsociere*, *D*: *TipDomeniu*,  
*V*: *TipValoare*); - operator care definește pe *M*(*D*) ca fiind egal cu *V*, indiferent dacă *M*(*D*) a fost definit sau nu anterior.

3. **Determina**(*M*: *TipAsociere*, *D*: *TipDomeniu*,  
*V*: *TipValoare*): *boolean*; - operator care returnează valoarea adevărat și asignează variabile *V* cu *M*(*D*) dacă aceasta din urmă există; altfel returnează valoarea fals.

-----

## 6.8.2. Implementarea TDA Asociere cu ajutorul tablourilor

- În multe situații **tipul domeniu** al funcției de asociere este un **tip elementar** care poate fi utilizat ca și indice într-o **structură tablou**.
  - În limbajul Pascal gama acestor indici poate fi extinsă la oricare tip scalar (subdomeniu, caracter sau enumerare).
- În cele ce urmează se va utiliza **tipul asociere** definit în secvența [6.8.2.a].

-----  
{TDA Asociere - implementare cu ajutorul tablourilor -  
definirea structurilor de date}

**TYPE** TipDomeniu = primaValoare ..ultimaValoare; [6.8.2.a]

**TYPE** TipAsociere = **ARRAY**[TipDomeniu] **OF** TipValoare;

-----  
/\* TDA Asociere - implementare cu ajutorul tablourilor -  
definirea structurilor de date - C \*/

**enum** { prima\_valoare = 0,  
ultima\_valoare = 100};

**typedef** unsigned char tip\_domeniu;

/\*[6.8.2.a]\*/

**typedef** tip\_valoare tip\_asociere[ultima\_valoare-

```

                                prima_valoare+1];
/*-----*/

```

- Presupunând în continuare că '*nedefinit*' este o constantă a lui TipValoare, operatorii aferenți **TDA Asociere** apar definiți în secvența [6.8.2.b].

```

-----
{TDA Asociere - implementare cu ajutorul tablourilor -
operatorii Initializeaza, Atribuie și Determina}

```

```

PROCEDURE Initializeaza(VAR M: TipAsociere);
  VAR i: TipDomeniu;
  BEGIN
    FOR i:= primaValoare TO ultimaValoare DO
      M[i]:= nedefinit
    END;{Initializeaza}

```

```

PROCEDURE Atribuie(VAR M: TipAsociere; D: TipDomeniu;
V: TipValoare);
  BEGIN
    M[D]:= V
  END;{Atribuie}

```

```

FUNCTION Determina(VAR M:TipAsociere; D: TipDomeniu;
                    VAR V: TipValoare): boolean;
  BEGIN
    IF M[D]:= nedefinit THEN
      Determina:= FALSE
    ELSE
      BEGIN
        V:= M[D]
        Determina:= TRUE
      END
    END;{Determina}

```

```

-----
/* TDA Asociere - implementare cu ajutorul tablourilor -
operatorii Initializeaza, Atribuie și Determina - C */

```

```

void initializeaza(tip_asociere m){
  tip_domeniu i;
  for( i= prima_valoare; i <= ultima_valoare; i ++){
    m[i-prima_valoare]= nedefinit;
  }/*initializeaza*/

void atribuie(tip_asociere m, tip_domeniu d,tip_valoare v){
  m[d-prima_valoare]= v;
}/*atribuie*/                                     /*[6.8.2.b]*/

boolean determina(tip_asociere m, tip_domeniu d,
                  tip_valoare* v){
  boolean determina_result;
  if (m[d-prima_valoare]= nedefinit)
    determina_result= false;

```

```

        else
        {
            *v= m[d-prima_valoare];
            determina_result= true;
        }
    return determina_result;
} /*determina*/
/*-----*/

```

### 6.8.3. Implementarea TDA Asociere cu ajutorul listelor înlănțuite

- Există multe posibilități de a implementa funcțiile de asociere cu domenii finite.
  - Astfel tabelele HASH (§7.) reprezintă în anumite situații o soluție excelentă. Ele vor fi studiate în capitolul următor.
- În paragraful de față se va descrie o altă posibilitate.
  - Orice structură **asociere**, poate fi reprezentată ca o listă de perechi  $(D_1, V_1), (D_2, V_2), \dots, (D_k, V_k)$ , unde
    - $D_1, D_2, \dots, D_k$  sunt elementele ale **domeniului**.
    - $V_i$  sunt valori ale **codomeniului** pe care asocierea le pune în corespondență cu  $D_i$  pentru  $i = 1, 2, \dots, k$ .
  - Pentru a implementa astfel de perechi de elemente se poate utiliza o **structură de date listă**.
  - În acest sens, **tipul abstract de date asociere** poate fi definit ca și o **listă înlănțuită**, implementată în oricare din manierele cunoscute, în care TipElement are următoarea structură [6.8.3.a].

-----  
**{TDA Asociere - implementare cu ajutorul listelor înlănțuite - structuri de date}**

```

TYPE TipElement = RECORD
    argument: TipDomeniu;
    valoare: TipValoare
END;

```

[6.8.3.a]

-----  
**/\* TDA Asociere - implementare cu ajutorul listelor înlănțuite - structuri de date -C \*/**

```

typedef struct tip_element {
    tip_domeniu argument;
    tip_valoare valoare;
} tip_element;
/*-----*/

```

- Cei trei operatori care gestionează **structura asociere**, descriși în termenii operatorilor definiți asupra **TDA Lista** apar în secvența [6.8.3.b].

---

{**TDA Asociere** - implementare bazată pe **TDA Listă** (varianta restrânsă) - operatorii **Initializeaza**, **Atribuie** și **Determina**}

**PROCEDURE Initializeaza**(VAR M: TipAsociere);

**BEGIN**

[identica cu cea de la liste] [6.8.3.b]

**END;**{**Initializeaza**}

**PROCEDURE Atribuie**(VAR M: TipAsociere; D: TipDomeniu;

V: TipValoare);

**VAR** x: TipElement;

p: TipPozitie;

**BEGIN**

**Creaza**(x);

x.argument:= D;

x.valoare:= V;

p:= **Primul**(M)

**WHILE** p<>**Fin**(M) **DO**

**IF** **Furnizeaza**(p,M).argument=D **THEN**

**Suprima**(p,M) {suprimă M(D)}

**ELSE**

p:= **Urmator**(p,M):

**Insereaza**(M,x,**Primul**(M)) {inserează (D,V) în listă}

**END;**{**Atribuie**}

**FUNCTION Determina**(VAR M: TipAsociere; D: TipDomeniu;

**VAR** V: TipValoare): boolean;

**VAR** p: TipPozitie;

gasit: boolean;

**BEGIN**

p:= **Primul**(M);

gasit:= **false**;

**WHILE** (p<>**Fin**(M)) **AND NOT** gasit **DO**

**BEGIN**

**IF** **Furnizeaza**(p,M).argument=D **THEN**

**BEGIN**

V:= **Furnizeaza**(p,M).valoare;

gasit:= **true**

**END;**{**IF**}

p:= **Urmator**(p,M)

**END;**{**WHILE**}

**Determina**:= gasit

**END;**{**Determina**}

---

/\* **TDA Asociere** - implementare bazată pe **TDA Listă** (varianta restrânsă) - operatorii **initializeaza**, **atribuie** și **determina** - varianta C \*/

**void initializeaza**(tip\_asociere m){

```

        ; /*identica cu cea de la liste*/          /*[6.8.3.b]*/
    }/*initializeaza*/

void atribuie(tip_asociere m, tip_domeniu d,tip_valoare v){
    tip_element x;
    tip_pozitie p;
    creaza(x);
    x.argument= d;
    x.valoare= v;
    p= primul(m);
    while (p!=fin(m))
        if (furnizeaza(p,m).argument==d)
            suprima(p,m); /*suprimă M(D)*/
        else
            p= urmator(p,m);
    insereaza(m,x,primul(m)); /*insereaza (D,V) în listă*/
}/*atribuie*/

boolean determina(tip_asociere m, tip_domeniu d,
                  tip_valoare* v){
    tip_pozitie p;
    boolean gasit;
    boolean determina_result;
    p= primul(m);
    gasit= false;
    while ((p!=fin(m)) && ! gasit)
    {
        if (furnizeaza(p,m).argument==d)
        {
            *v= furnizeaza(p,m).valoare;
            gasit= true;
        }/*if*/
        p= urmator(p,m);
    }/*while*/
    determina_result= gasit;
    return determina_result;
}/*determina*/
/*-----*/

```

## 6.9. Aplicații propuse

### Aplicația 6.9.1

Se consideră următoarele variante de implementare a TDA Listă înlănțuită:

- Cu ajutorul cursorilor ca listă neordonată;
- Cu ajutorul cursorilor ca listă ordonată;
- Cu ajutorul pointerilor ca listă neordonată;
- Cu ajutorul pointerilor ca listă ordonată;

- Cu ajutorul pointerilor utilizând tehnica reordonării;
- Cu ajutorul pointerilor ca listă circulară;
- Cu ajutorul pointerilor ca listă dublu înlanțuită.

Se cere:

- Pentru fiecare variantă de implementare să se redacteze operatorii specifici de inserție, suprimare, căutare și parcurgere a listei.
- Să se realizeze o analiză comparată a performanței implementării operatorului de căutare în variantele mai sus enumerate.

### Aplicația 6.9.2

Se consideră următoarele variante de implementare a TDA Listă înlanțuită cu ajutorul pointerilor:

- Varianta normală;
- Varianta bazată pe tehnica nodului fictiv;
- Varianta bazată pe tehnica nodului fanion;
- Varianta bazată pe tehnica celor două noduri fictive.

Se cere să se analizeze comparativ avantajele și dezavantajele fiecărei variante de implementare în raport cu operatorii specifici definiți pe TDA Listă.

### Aplicația 6.9.3

Se cere să se implementeze într-o manieră performantă un produs program pentru gestionarea unei colecții de polinoame identificate prin câte un indice, care să permită realizarea în manieră interactivă a următorilor operatori:

- `initializeaza` - inițializează colecția de polinoame;
- `initialzeaza(i)` - inițializează polinomul `i`;
- `citeste(i)` - citește interactiv polinomul `i` (în forma unor perechi coeficient - putere a lui `x`);
- `afiseaza` - afișează colecția de polinoame;
- `afiseaza(i)` - afișează polinomul `i`;
- `evaluateaza(i)` - pentru o valoare citită a lui `x` afișează valoarea polinomului `i`;



- `aduna(i, j, s)` - creează polinomul `s` ca și sumă a polinoamelor `i` și `j`;
- `diferenta(i, j, d)` - creează polinomul `d` ca și diferență a polinoamelor `i` și `j`;
- `produs(i, j, p)` - creează polinomul `p` ca și produs al polinoamelor `i` și `j`;
- `exit` - ieșirea din program.

#### **Aplicația 6.9.4**

Se cere să se realizeze o implementare a TDA Ansamblu. În prealabil, TDA ansamblu se va completa cu operatori referitori la verificarea priorității unui element, la modificarea priorității unui element, respectiv a unui operator de înlocuire a unui element al ansamblului cu altul.

#### **Aplicația 6.9.5**

Se cere să se realizeze un studiu al performanțelor implementărilor TDA Coadă bazată pe prioritate cu ajutorul tablourilor, tablourilor circulare, listelor înlănțuite și ansamblelor.