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 defineşte pornind de la noţiunea de **vector**.
    - **Elementele** unei liste sunt toate de **acelaşi** 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 funcţie de necesităţi pot **creşte** sau **descreşte** şi ale căror **elemente** pot fi **referite, inserate** sau **şterse** în orice poziţie din cadrul listei.
    - Două sau mai multe **liste** pot fi **concatenate** sau **scindate** în subliste.
- În practica programării listele apar în mod obişnuit în aplicaţii referitoare la regăsirea informaţiei, implementarea translatoarelor de programe, simulare, modelare, etc,.

6.2. TDA Listă

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

$a_1, a_2, \ldots, a_n$ [6.2.a]

    - unde $n \geq 0$ şi fiecare a_i aparţine **tipului de bază**.
- Numărul n al nodurilor se numeşte **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** funcţie de **poziţia** lor în cadrul listei.

  - De regulă, se spune că nodul a_i se află pe poziţia `i`.
  - Se spune că a_i **precede** pe a_{i+1} pentru `i=1,2,...,n-1`.
  - Se spune că a_i **succede** (urmează) lui a_{i-1} pentru `i=2,3,4, ...,n`.

- Este de asemenea convenabil să se postuleze existenţa **poziţiei** următoare **ultimului element** al **listei**.

  - În această idee se introduce funcţia ***Fin***`(TipLista L)` care returnează poziţia următoare poziţiei `n` în lista `L` având `n` elemente.
  - Se observă că ***Fin***`(L)` are o **distanţă** variabilă faţă de începutul listei, funcţie de faptul că lista creşte sau se reduce, în timp ce alte poziţii au o distanţă fixă faţă de începutul listei.

- Pentru a defini un **tip de date abstract**, în cazul de faţă **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 aplicaţiile, iar pe de altă parte natura setului depinde esenţial atât de **maniera de implementare** cât şi de cea de **utilizare** a **listelor**.

- În continuare se prezintă **două seturi reprezentative de operatori** care acţionează asupra listelor, unul restrâns şi 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 aparţin tipului de bază `Tip_Nod`.
  - `x` de `Tip_Nod` este un **obiect** al acestui tip (deci un nod al listei).
  - `p` este o variabilă de `Tip_Pozitie`. Tipul poziţie 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 secvenţă formată din zero sau mai multe elemente numite **noduri** toate încadrate într-un anumit tip numit **tip de bază.**

Notaţii:

Tip_Lista L;
Tip_Pozitie p;
Tip_Nod x; [6.2.1.a]

Operatori:

1. *Tip_Pozitie* ***Fin****(Tip_Lista L)* - operator care returnează poziţia următoare ultimului nod al listei, adică poziţia următoare sfârşitului ei. În cazul liste vide ***Fin****(L)=0;*

2. ***Insereaza****(Tip_Lista L, Tip_Nod x, Tip_Pozitie p)* - inserează în lista *L*, nodul *x* în poziţia *p*. Toate nodurile care urmează acestei poziţii se mută cu un pas spre poziţiile superioare, astfel încât $a_1,a_2,...,a_n$ devine $a_1,a_2,...,a_{p-1},x,a_p,...a_n$. Dacă *p* este ***Fin****(L)* atunci lista devine $a_1,a_2,...a_n,x$. Dacă *p*>***Fin****(L)* rezultatul este nedefinit;

3. *Tip_Pozitie* ***Cauta****(Tip_Nod x, Tip_Lista L)* - caută nodul *x* în lista *L* şi returnează poziţia nodului. Dacă *x* apare de mai multe ori, se furnizează poziţia primei apariţii. Dacă *x* nu apare de loc, se returnează valoarea ***Fin****(L);*

4. *Tip_Nod* ***Furnizeaza****(Tip_Pozitie p, Tip_Lista L)* - operator care returnează nodul situat pe poziţia *p* în lista *L*. Rezultatul este nedefinit dacă *p*=***Fin****(L)* sau dacă în *L* nu există poziţia *p*. Se precizează că tipul operatorului ***Furnizeaza*** trebuie să fie identic cu tipul de bază al listei;

5. ***Suprima****(Tip_Pozitie p, Tip_Lista L)* - suprimă elementul aflat pe poziţia *p* în lista *L*. Dacă *L* este $a_1,a_2,...,a_n$ atunci după suprimare *L* devine $a_1,a_2,...a_{p-1},a_{p+1},...,a_n$. Rezultatul este nedefinit dacă *L* nu are poziţia *p* sau dacă *p*=***Fin****(L);*

6. *Tip_Pozitie* ***Urmator****(Tip_Pozitie p, Tip_Lista L)* - operator care returnează poziţia următoare poziţiei *p* în cadrul listei. Dacă *p* este ultima poziţie în *L* atunci ***Urmator****(p,1)*=***Fin****(L)*. ***Urmator*** nu este definit pentru *p*=***Fin****(L);*

7. *Tip_Pozitie* ***Anterior****(Tip_Pozitie p, Tip_Lista L)* - operator care returnează poziţia anterioară poziţiei *p* în cadrul listei. Dacă *p* este prima

poziţie în *L* atunci ***Anterior*** nu este definit;

8. *Tip_Pozitie* ***Initializeaza****(Tip_Lista L)*- operator care face lista *L* vidă şi returnează poziţia ***Fin****(L)=0;*

9. *Tip_Pozitie* ***Primul****(Tip_Lista L)* - returnează valoarea primei poziţii în lista *L*. Dacă *L* este vidă, poziţia returnată este ***Fin****(L)=0;*

10. ***TraverseazaLista****(Tip_Lista L,* ***Procedura*** *ProcesareNod(...))* - parcurge nodurile listei *L* în ordinea în care apar ele în listă şi aplică fiecăruia procedura *ProcesareNod.*

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

  - 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 definiţi, algoritmul de eliminare a adreselor duble din listă poate fi formulat astfel [6.2.1.b].

```
/*Exemplu - Eliminarea nodurilor duplicat din cadrul unei
liste - se utilizeaza TDA Lista 1 definit anterior*/

/*definirea structurii unui nod al listei*/
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 poziţia 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);
    } /*while*/
} /*elimina*/
```

- În legătură cu cea de-a doua buclă **`while`**, se poate face o **observaţie importantă** referitoare la variabila `q`.

    - Dacă se suprimă din listă elementul situat pe poziţia `q`, elementele aflate pe poziţiile `q+1`, `q+2`, etc, retrogradează cu o poziţie în listă.
    - Dacă în mod întâmplător `q` este ultimul element al listei, atunci valoarea sa devine ***`Fin`***`(L)`.
    - Dacă în continuare s-ar executa instrucţia `q=`***`Urmator`***`(q,L)`, lucru dictat de logica algoritmului, s-ar obţine 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 **condiţiei**, respectiv dacă condiţia instrucţiei **`if`** este adevărată se execută **numai** suprimarea iar în caz contrar **numai** avansul.

6.2.2. TDA Listă 2. Set de operatori extins

- În acelaşi 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 **înlănţuiri** [SH90].

TDA Listă 2

(Set de operatori extins) [6.2.2.a]

Modelul matematic: o secvenţă finită de noduri. Toate nodurile aparţin unui acelaşi tip numit **tip de bază.** Fiecare nod constă din două părţi: o parte de informaţii şi o a doua parte conţinând legătura la nodul următor. O variabilă specială indică primul nod al listei.

Notaţii:

```
Tip_Nod - tipul de bază;
Tip_Lista - tip indicator la tipul de bază;
Tip_Info - partea de informaţii a lui TipNod;
Tip_Indicator_Nod - tip indicator la tipul de bază
     (identic cu TipLista);
Tip_Lista incLista - variabilă care indică
     începutul listei;
Tip_Indicator_Nod curent, p, pnod; - indică noduri
     în lista;
Tip_Nod element;
Tip_Info info; parte de informaţii a unui nod;
b - valoare booleană;
```

null - indicatorul vid.

Operatori:

1. ***CreazaListaVida****(Tip_Lista incLista)* - variabila *incLista* devine **null.**

2. *boolean* ***ListaVida****(Tip_Lista incLista)* - operator care returnează **TRUE** dacă lista este vidă respectiv **FALSE** altfel.

3. ***Primul****(Tip_Lista incLista, Tip_Indicator_Nod curent)* - operator care face ca variabila *curent* să indice primul nod al listei precizată de *incLista*.

4. *boolean* ***Ultimul****(Tip_Lista incLista, Tip_Indicator_Nod curent)* - operator care returnează **TRUE** dacă *curent* indică ultimul nod al listei.

5. ***InserInceput****(Tip_Lista incLista, Tip_Indicator_Nod pnod)* - inserează la începutul listei *incLista* nodul indicat de *pnod*.

6. ***InserDupa****(Tip_Lista incLista, Tip_Indicator_Nod curent, Tip_Indicator_Nod pnod)* - inserează nodul indicat de *pnod* după nodul indicat de *curent*. Se presupune că *curent* indică un nod din listă.

7. ***InserInFatza****(Tip_Lista incLista, Tip_Indicator_Nod curent, Tip_Indicator_Nod pnod)* - inserează nodul indicat de *pnod* în faţa nodului indicat de *curent*.

8. ***SuprimaPrimul****(Tip_Lista incLista)* - suprimă primul nod al listei *incLista*.

9. ***SuprimaUrm****(Tip_Lista incLista, Tip_Indicator_Nod curent)* - suprimă nodul următor celui indicat de *curent* în lista *incLista*.

10. ***SuprimaCurent****(Tip_Lista incLista, Tip_Indicator_Nod curent)* - suprimă nodul indicat de *curent* în lista *incLista.*

11. ***Urmatorul****(Tip_Lista incLista, Tip_Indicator_Nod curent)* - *curent* se poziţionează pe următorul nod al listei *incLista*. Dacă *curent* indică ultimul nod al listei el va deveni **null**.

12. ***Anterior****(Tip_Lista incLista, Tip_Indicator_Nod curent)* - *curent* se poziţionează pe nodul anterior celui curent în lista *incLista.*

13. ***MemoreazaInfo****(Tip_Lista incLista, Tip_Indicator_Nod curent, Tip_Info info)* - atribuie nodului indicat

de *curent* informaţia *info* în lista *incLista*.

14. ***MemoreazaLeg****(Tip_Lista incLista, Tip_Indicator_Nod curent, Tip_Indicator_Nod p)* - atribuie câmpului *urm* (de legătură) al nodului indicat de *curent* valoarea *p*.

15. *Tip_Info* ***FurnizeazaInfo****(Tip_Lista incLista, Tip_Indicator_Nod curent)* - returnează partea de informaţie a nodului indicat de *curent*.

16. *Tip_Indicator_Nod* ***FurnizeazaUrm****(Tip_Lista incLista, Tip_Indicator_Nod curent);* - returnează legătura nodului *curent* (valoarea câmpului urm).

17. ***TraverseazaLista****(Tip_Lista incLista,* ***Procedura*** *ProcesareNod(...))* - parcurge nodurile listei *incLista* în ordinea în care apar ele în listă şi aplică fiecăruia procedura *ProcesareNod*.

6.3. Tehnici de implementare a listelor

- De regulă pentru structurile de date **fundamentale** există **construcţii de limbaj** care le reprezintă, construcţii care îşi găsesc un anumit corespondent în particularităţile **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**, observaţie valabilă şi 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** şi **funcţiile** care implementează **operatorii specifici** prelucrării listelor vor fi descrişi 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 locaţii succesive de memorie.**
- În această reprezentare:

- O listă poate fi uşor **traversată**.
- Noile noduri pot fi **adăugate** în mod simplu la **sfârşitul listei**.
- **Inserţia** unui nod în mijlocul listei presupune însă deplasarea tuturor nodurilor următoare cu o poziţie spre sfârşitul listei pentru a face loc noului nod.
- **Suprimarea** oricărui nod cu excepţia ultimului, presupune de asemenea deplasarea tuturor celorlalte în vederea eliminării spaţiului creat.
- Inserţia şi suprimarea unui nod necesită un **efort** de execuţie O*(n)*.

- În **implementarea bazată pe tablouri,** `tip_lista` se defineşte ca o **structură** (**articol**) cu două câmpuri.
  - (1) Primul câmp este un **tablou** numit `noduri`, cu elemente de `tip_nod`.
    - 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 aplicaţie.
  - (2) Cel de-al doilea câmp este un **indicator** (`ultim`) care indică poziţia în tablou a **ultimului nod** al listei.
- Cel de-al `i`-lea nod al listei se găseşte în cel de-al `i`-lea element al tabloului, pentru $1 \le i \le ultim$ (fig.6.3.1.a).
- **Poziţia** în cadrul listei se reprezintă prin valori întregi, respectiv cea de-a `i`-a poziţie prin valoarea `i`.
- Funcţia ***Fin***`(L)` returnează valoarea $ultim+1$.

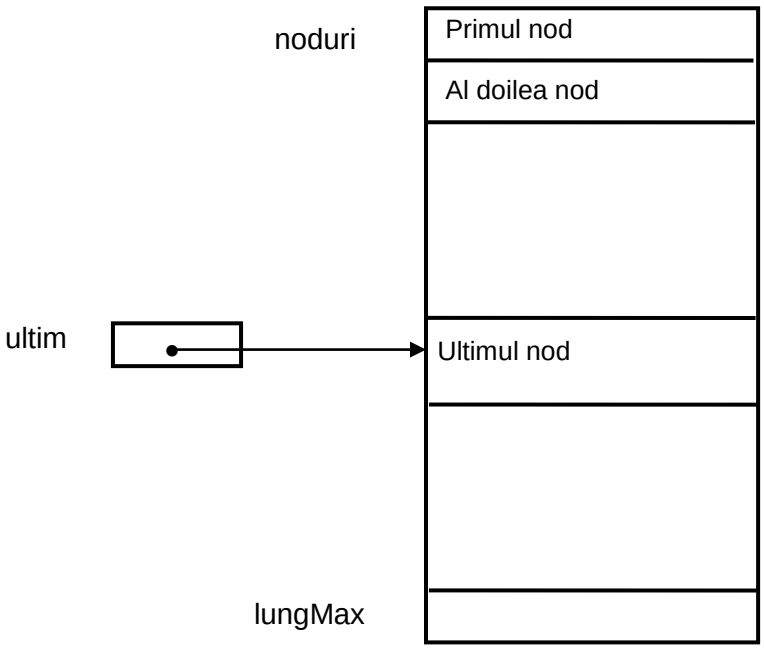

Fig.6.3.1.a. Implementarea listelor cu ajutorul structurii tablou.

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

```
/*Implementarea listelor cu ajutorul structurii tablou*/

#define lung_max = 100

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

typedef int tip_pozitie;
```

- Secvenţa de program [6.3.1.b] prezintă modul în care se pot implementa operaţiile specifice setului **restrâns** de operatori: ***Fin, Insereaza, Suprima*** şi ***Cauta*** utilizând implementarea bazată pe tablouri a listelor.

- Se fac următoarele precizări:

  - Dacă se încearcă inserţia 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 găseşte elementul căutat, ***Cauta*** returnează poziţia `ultim+1`.
  - S-a prevăzut parametrul boolean `er`, care în caz de **eroare** se returnează cu valoarea adevărat şi care poate fi utilizat pentru tratarea erorii sau pentru întreruperea execuţiei programului.

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

#include <stdlib.h>                                /*[6.3.1.b]*/

#define lung_max 100
#define n 30

/*definire structura de date nod al listei*/
typedef struct{
  int nr_curent;
  char* nume;
  char* adresa;
} tip_nod;

/*definire structura de date listă*/
typedef struct{
   tip_nod noduri[lung_max];
   int ultim;
} tip_lista;
```

```
typedef int tip_pozitie;

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

boolean reccmp(tip_nod, tip_nod); /*comparare conţinut
                                    noduri*/

tip_pozitie fin(tip_lista* l)
  /*returneaza poziţia următoare sfârşitului listei*/
  {
     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 poziţia p a listei; performanţa O(n)*/
  {
    tip_pozitie q;
    *er= false;
    if(l->ultim>=lung_max)
        {
          *er= true;
          printf("lista este plina");
        }
      else
        if((p>l->ultim+1) || (p<1))
            {
              *er= true;
              printf("poziţia 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 poziţia p a listei*/
  {
    tip_pozitie q;                  /*performanţa 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 poziţia lui x în listă*/
    {
      tip_pozitie q;
      boolean gasit;
      tip_pozitie cauta_result;
      q= 1; gasit= false;          /*performanţa 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 conţinut noduri*/
    {
     if ((x.nr_curent==y.nr_curent) &&
           !(strncmp(x.nume,y.nume,n)) &&
           !(strncmp(x.adresa,y.adresa,n)))
         return true;
       else
         return false;
    } /*reccmp*/
/*-----------------------------------------------------------*/
```

- În acest context, implementarea celorlalţi operatori **nu** ridică probleme deosebite:
    - Operatorul ***Primul*** returnează întotdeauna valoarea 0.
    - 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 `-1`.
- La prima vedere pare tendenţioasă redactarea unor proceduri care să guverneze **toate accesele** la o anumită **structură de date**.
- Cu toate acestea acest lucru are o importanţă 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 **detaliile lor de implementare**:

- (1) Pe de-o parte creşte **eleganţa, înţelegerea** şi **siguranţa** în funcţionare a programului.
- (2) Pe de altă parte **modificarea** programului sau a structurii de date propriu-zise se poate realiza facil, **doar** prin modificarea structurii şi/sau a procedurilor care o definesc, fără a mai fi necesară căutarea şi modificarea în program a locurilor din care se fac accese la respectiva structură.
- (3) Această **flexibilitate** poate să joace de asemenea un rol esenţial în cazul efortului necesar dezvoltării unor produse software de mari dimensiuni.

6.3.2. Implementarea listelor cu ajutorul pointerilor

- **Listele liniare** se pot implementa şi 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 C */

typedef int tip_info;                          /*[6.3.2.a]*/

/*definire indicator la un nod al listei înlănţuite*/
typedef struct tip_nod* tip_pointer_nod;

/*definire tip nod al listei înlănţuite*/
typedef struct {
    int cheie;
    tip_pointer_nod urm;
    tip_info info;
} tip_nod;

/*definire tip listă înlănţuită*/
typedef tip_pointer_nod tip_lista;
-------------------------------------------------------------
```

- După cum se observă, în cazul definirii unui **nod al structurii listă înlănţuită** s-au pus în evidenţă trei câmpuri:
  - O **cheie** care serveşte la identificarea nodului.
  - Un **pointer** de înlănţuire la nodul următor (`urm`).
  - Un **câmp** `info` conţinând informaţia 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 secvenţa [6.3.2.a] se observă că o listă liniară poate fi definită ca şi o **structură recursivă** având o componentă de tip **identic** cu cel al structurii complete.

    - Caracteristica esenţială a unei astfel de structuri rezidă în prezenţa **unei singure înlănţuiri**.

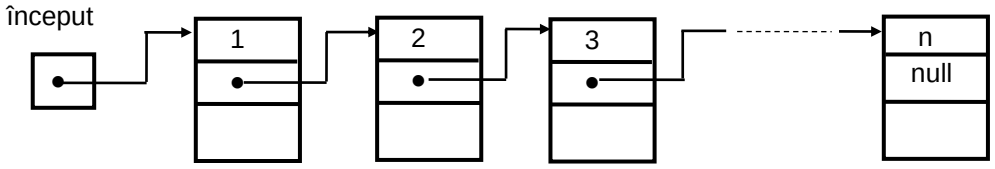

Fig.6.3.2.a. Exemplu de listă liniară

- În continuare, în cadrul acestui paragraf se prezintă câteva **tehnici de implementare a listelor liniare** ca şi **structuri recursive**.

- (1) Există posibilitatea ca **pointerul** `inceput` care indică începutul listei, să indice o **componentă** de `tip_nod` având câmpurile `cheie` şi `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 situaţii prelucrarea listelor înlănţuite (fig.6.3.2.b).

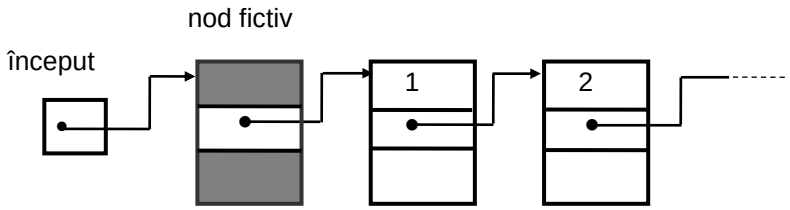

Fig.6.3.2.b. Implementarea listelor prin tehnica nodului fictiv

- (2) Există de asemenea posibilitatea utilizării unui **nod fictiv final** pe post de **fanion** având înlănţuirea **`nil`** sau care se înlănţuie cu el însuşi [Se88].

    - Această tehnică de implementare este cunoscută sub denumirea de **tehnica nodului fanion** (fig.6.3.2.c).

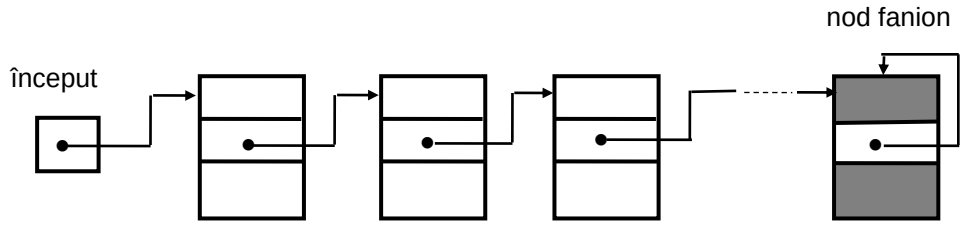

Fig.6.3.2.c. Implementarea listelor prin tehnica nodului fanion

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

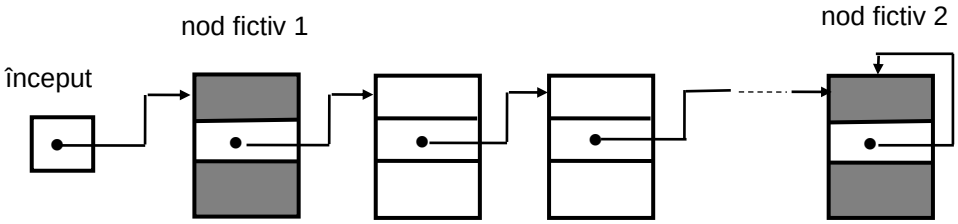

Fig.6.3.2.d. Implementarea listelor cu ajutorul tehnicii celor două noduri fictive

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

6.3.2.1. Tehnici de inserţie a nodurilor şi de creeare a listelor înlănţuite

- Presupunând că este dată o **structură de date listă,** în continuare se prezintă o secvenţă de cod pentru inserţia unui nod nou în listă.
- Iniţial, **inserţia** 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].

```
-------------------------------------------------------------
/*inserţie la începutul listei înlănţuite*/

   tip_lista inceput;                        /*[6.3.2.1.a]*/
   tip_pointer_nod auxiliar;

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

- În figura 6.3.2.1.a se prezintă grafic maniera în care se desfăşoară o astfel de inserţie.

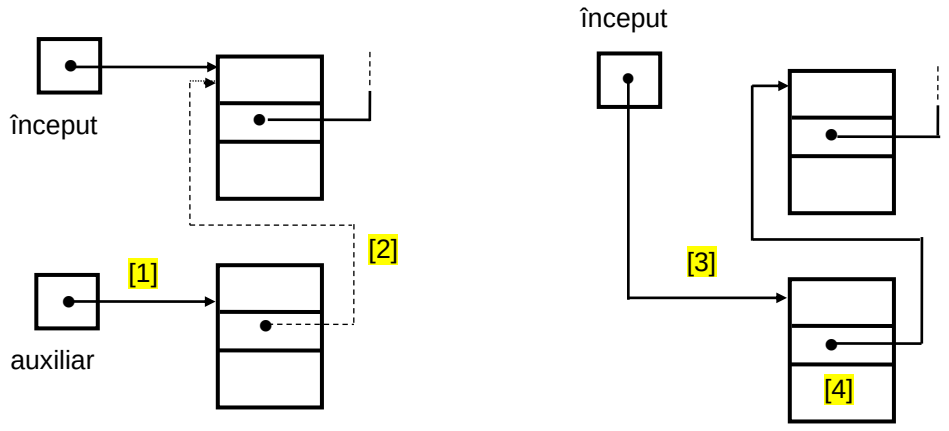

Fig.6.3.2.1.a. Inserţia unui nod la începutul unei liste înlănţuite

- Pe baza acestui fragment de program se prezintă în continuare, **crearea unei liste înlănţuite.**
    - Se porneşte 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 secvenţă s-a omis asignarea câmpurilor de informaţie [6.3.2.1.b].
    - Datorită faptului că inserţia noului nod are loc de fiecare dată la **începutul** listei, secvenţa creează lista în **ordinea inversă** a furnizării cheilor.

```
/*crearea unei liste înlănţuite*/

   tip_lista inceput;                        /*[6.3.2.1.b]*/
   tip_pointer_nod auxiliar;

   inceput= null;  /*se porneşte 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 doreşte crearea listei în **ordine naturală**, atunci este nevoie de o secvenţă care inserează un nod **la sfârşitul** unei liste.

- Această secvenţă de program se redactează mai simplu dacă se cunoaşte locaţia **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` = **`null`**.

- În practică această soluţie **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 prezenţa lui `ultim`, secvenţa de program care inserează un nod la sfârşitul unei liste liniare şi concomitent îl actualizează pe `ultim` este următoarea [6.3.2.1.c]:

```
-------------------------------------------------------------
/*inserţie la sfârşitul unei liste înlănţuite*/

  tip_pointer_nod ultim, auxiliar;       /*[6.3.2.1.c]*/

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

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

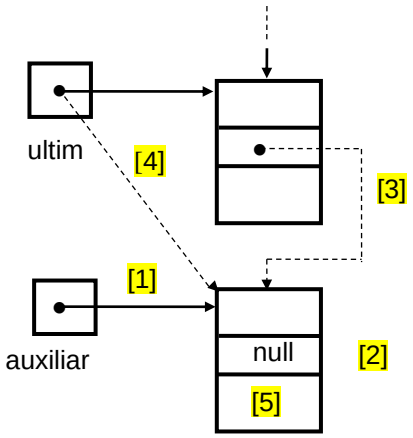

Fig.6.3.2.1.b. Inserţia unui nod la sfârşitul unei liste înlănţuite

- Referitor la secvenţa [6.3.2.1.c] se atrage atenţia 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 posibilităţi de a rezolva această problemă:

  - (1) Primul nod trebuie inserat printr-un alt procedeu spre exemplu prin **inserţie la începutul listei**.

- În continuare nodurile se pot adăuga conform secvenţei 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 şi ca atare `ultim->urm` există chiar şi 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 şi se creează un nou nod fanion.

- În continuare se descrie inserţia 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 condiţii **inserţia 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.

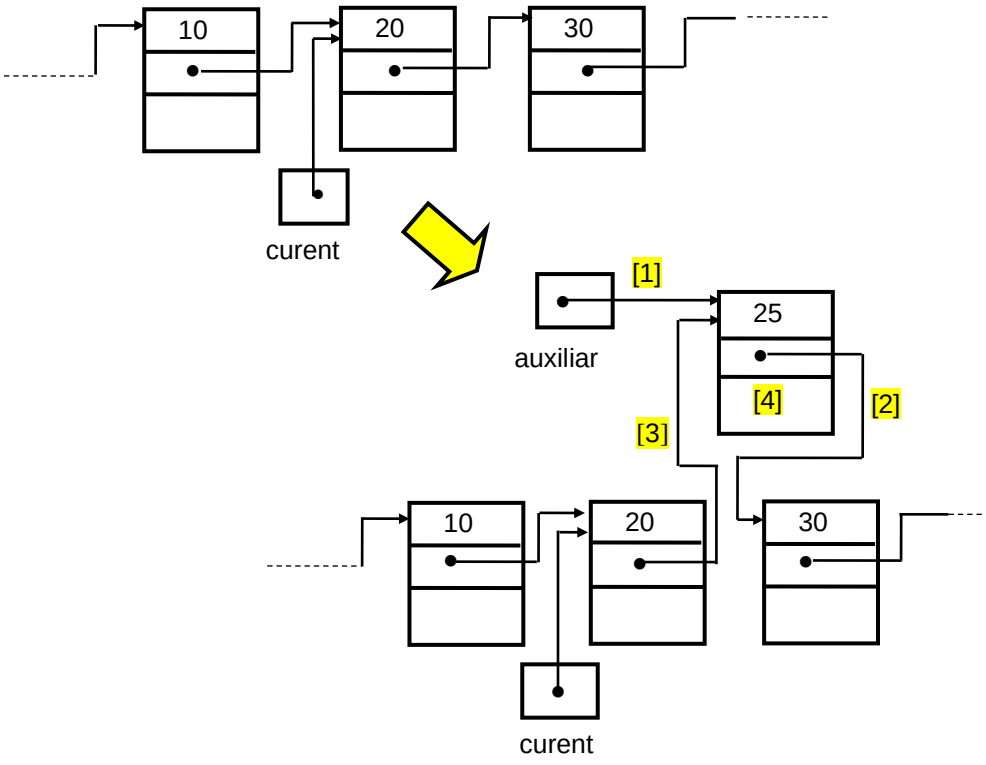

Fig.6.3.2.1.c. Inserţia unui nod nou după un nod precizat (`curent`)

- Secvenţa de cod care realizează această inserţie apare în [6.3.2.1.d].

```
-----------------------------------------------------------
/*inserţia unui nod nou după un nod precizat de indicatorul
curent*/

  tip_pointer_nod curent, auxiliar;      /*[6.3.2.1.d]*/
```

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

- Dacă se doreşte însă inserţia noului nod în lista liniară **înaintea unui nod indicat** de pointerul `curent`, apare o complicaţie 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 conţinutul nodului indicat de `curent`.
    - (3) Se inserează noul nod după nodul indicat de pointerul `curent`.
    - (4) Se creează câmpurile `cheie` şi `info` pentru noul nod şi se asignează cu ele câmpurile corespunzătoare ale vechiului nod indicat de pointerul `curent`.

- Secvenţa 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.

```
/*inserţia unui nod nou înaintea unui nod precizat de
indicatorul curent*/

  tip_pointer_nod curent, auxiliar;      /*[6.3.2.1.e]*/

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

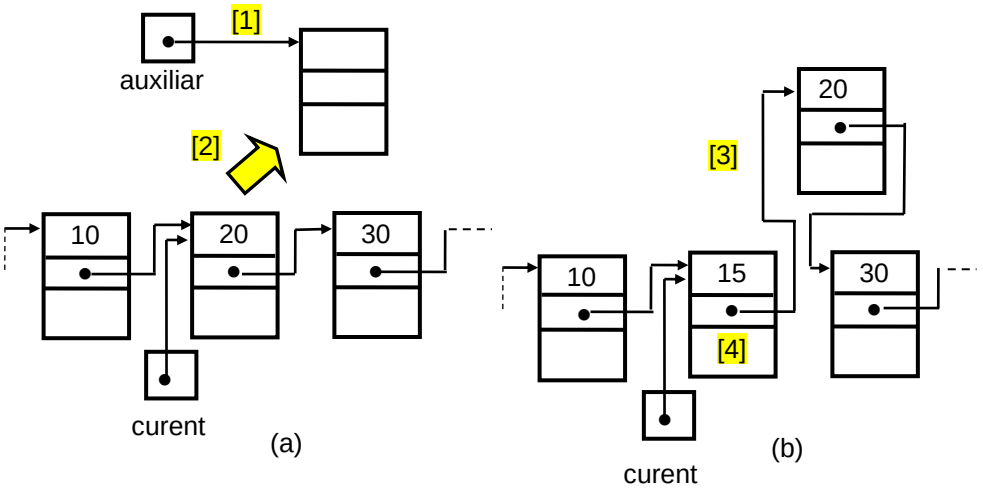

Fig.6.3.2.1.d. Inserţia unui nod nou în faţa unui nod indicat

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 înlănţuite şi se cere să **se suprime 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)*/

  tip_pointer_nod curent, auxiliar;      /*[6.3.2.2.a]*/

  [1] auxiliar= curent->urm;
  [2] curent->urm= auxiliar->urm;   /*performanţa O(1)*/
  [3] free(auxiliar);
-------------------------------------------------------------
```

- Efectul execuţiei aceste secvenţe de cod se poate urmări în figura 6.3.2.2.a.

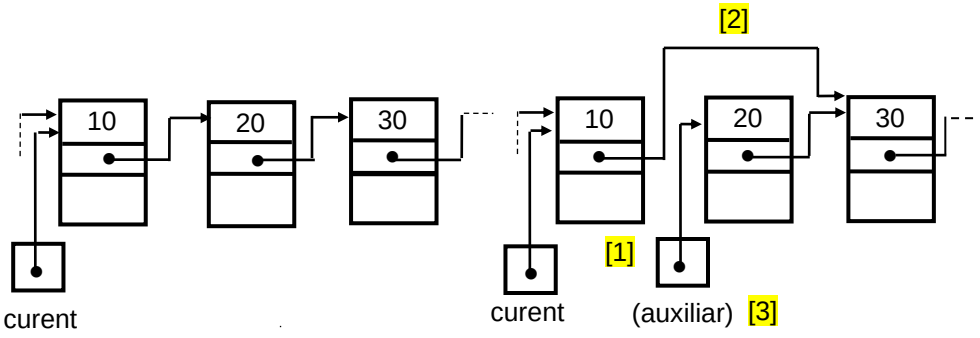

Fig.6.3.2.2.a. Tehnica suprimării succesorului nodului indicat de `curent`

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

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

   tip_pointer_nod curent;                /*[6.3.2.2.b]*/

   curent->urm= curent->urm->urm;     /*performanţa O(1)*/
-------------------------------------------------------------
```

- 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 condiţiile execuţiei secvenţei [6.3.2.2.b] se pierde.

- Revenind la problema suprimării unui nod, se consideră cazul în care se doreşte **suprimarea** nodului **indicat** de `curent`.

    - Aici apare aceeaşi dificultate semnalată în paragraful anterior, generată de imposibilitatea aflării simple a **adresei predecesorului nodului** indicat de pointerul `curent`.

- Soluţia se bazează pe aceeaşi **tehnică**:

    - (1) Se copiază conţinutul nodului succesor integral în nodul indicat de `curent`.

    - (2) Se suprimă nodul succesor.

- Aceasta se poate realiza printr-o singură instrucţie şi anume [6.3.2.2.c]:

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

    tip_pointer_nod curent;              /*[6.3.2.2.c]*/

    *curent= *curent->urm;           /*performanţa O(1)*/
-------------------------------------------------------------
```

- Ca şi înainte, această soluţie prezintă **dezavantajul** că pierde iremediabil zona de memorie ocupată iniţial de succesorul nodului indicat de pointerul `curent`.

- O soluţie care evită acest dezavantaj este cea prezentată în secvenţa [6.3.2.2.d]:

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

     tip_pointer_nod curent, auxiliar;   /*[6.3.2.2.d]*/

     auxiliar= curent->urm;
     *curent= *auxiliar;              /*performanţa O(1)*/
     free(auxiliar);
-------------------------------------------------------------
```

- 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` este diferit de **`null`.**

- Pentru a evita acest neajuns se pot utiliza alte modalităţi de implementare a listei înlănţuite, spre exemplu **tehnica nodului fanion**.

6.3.2.3. Traversarea unei liste înlănţuite. Căutarea unui nod într-o listă înlănţuită

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

```
-------------------------------------------------------------
/*traversarea unei liste înlănţuite*/

     tip_lista inceput;                    /*[6.3.2.3.a]*/
     tip_pointer_nod curent;

     curent= inceput;
     while (curent!=null)              /*performanţa O(n)*/
      {
        prelucrare(*curent);
        curent= curent->urm;
      }
-------------------------------------------------------------
/*traversarea unei liste înlănţuite în sens invers
  (varianta recursivă*/

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

- O operaţie 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**.

```
-------------------------------------------------------------
/*cautarea unui nod cu o cheie precizată x (varianta 1)*/

   tip_lista inceput;                    /*[6.3.2.3.c]*/
   tip_pointer_nod curent;

   curent= inceput;
   while((curent!=null) && (curent->cheie!=x))
```

```
    curent= curent->urm;
  if(curent!=null) /*nodul căutat este indicat de curent*/
    else  /*nodul căutat nu este in lista inceput*/
```

- Dacă acest fragment se termină cu `curent=null`, 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 instrucţiunii **`while`**, dacă lista **nu** conţine nici un nod cu cheia `x`, atunci în momentul în care `curent` devine **`null`**, nodul indicat de `curent` **nu** există.

    - În consecinţă, funcţie de implementare, se semnalează eroare, deşi expresia booleană completă este perfect determinată, ea fiind falsă din cauza primei subexpresii.

- Varianta propusă de secvenţa [6.3.2.3.d] a operaţiei de căutare este **corectă** în toate cazurile, ea utilizând o **variabilă booleană ajutătoare** notată cu `gasit`.

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

     tip_lista inceput;                    /*[6.3.2.3.d]*/
     tip_pointer_nod curent;
     boolean gasit;

     gasit= false;
     curent= inceput;
     while((curent!=NULL)&&~gasit)
       if(curent->cheie==x)
           gasit= true;
         else
           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 şi `curent=null`.

- Pornind de la cele prezentate în acest subparagraf, se pot concepe cu uşurinţă funcţiile şi procedurile care materializează **operatorii** aplicabili listelor implementate cu ajutorul pointerilor atât în varianta restrânsă cât şi în varianta extinsă.

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

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

- De asemenea, în anumite situaţii este mai avantajos pentru programator din punctul de vedere al **performanţei codului implementat** să **evite** utilizarea **pointerilor**.

- În astfel de cazuri, **pointerii** pot fi simulaţi cu ajutorul **cursorilor** care sunt valori întregi ce indică **poziţii** în **tablouri**.

- În cadrul acestui paragraf va fi abordată **implementarea listelor înlănţuite** cu ajutorul **cursorilor**, scop în care se definesc structurile de date din [6.3.3.a].

- În accepţiunea acestei implementări:

  - (1) Pentru toate listele ale căror noduri sunt de `tip_nod`, se crează un tablou `(Zona)` având elementele de `tip_element`.

  - (2) Fiecare element al tabloului conţine un câmp `nod_lista` de `tip_nod` şi un câmp `urm` de `tip_cursor` definit de regulă ca şi **subdomeniu** al tipului întreg. În limbajul C `tip_cursor` este identic cu `int`.

  - (3) `tip_lista` este în aceste condiţii identic cu `tip_cursor`, orice listă fiind **precizată** de fapt de un **cursor** în tabloul `Zona`.

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

enum {lung_max = 10};                          /*[6.3.3.a]*/

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

typdef ... tip_nod;

/*definirea tipului cursor*/
typedef int tip_cursor;

/*definirea structurii unui nod al listei*/
typedef struct tip_element
 {
   tip_nod nod_lista;
   tip_cursor urm;
 }tip_element;

/*definirea tipului lista*/
typedef tip_cursor tip_lista;

tip_element zona[lung_max]; /*zona de memorie pentru liste*/
tip_lista disponibil; /*lista disponibililor*/
tip_lista L,M; /*declarare liste*/
-------------------------------------------------------------
```

- Spre exemplu dacă `L` de `tip_lista` este un cursor care indică începutul unei liste atunci:

  - Valoarea lui `zona[L].nod_lista` reprezintă **primul** nod al listei `L`.

- `zona[L].urm` este indexul (cursorul) care indică cel de-al doilea element al listei `L`, ş.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 şi M = D, E care partajează tabloul `Zona` cu lungimea maximă 10.

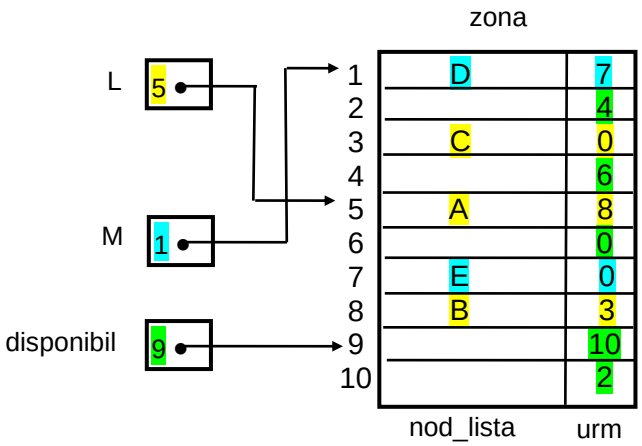

Fig.6.3.3.a. Implementarea listelor înlănţuite cu ajutorul cursorilor

- Se observă că:
  - Acele locaţii ale tabloului care **nu** apar în nici una din cele două liste sunt înlănţuite într-o altă listă numită `disponibil`.
  - Lista `disponibil` este utilizată fie:
    - Pentru a **furniza o nouă locaţie** în vederea realizării unei **inserţii**.
    - Pentru a **depozita o locaţie** a tabloului rezultată din **suprimarea** unui nod în vederea unei reutilizări ulterioare.
- 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 locaţie a listei `disponibil`.
    - (2) Se inserează această locaţie în poziţia dorită a listei `L`, actualizând valorile cursorilor implicaţi.
    - (3) Se asignează câmpul `nod_lista` al acestei locaţii cu valoarea lui `x`.
  - **Suprimarea** unui element `x` din lista `L` presupune:
    - (1) Suprimarea locaţiei care îl conţine pe `x` din lista `L`.

- (2) Inserţia acestei locaţii în capul listei `disponibil`.

- Atât **inserţia** cât şi **suprimarea** sunt de fapt cazuri speciale ale următoarei **situaţii**:

  - Fie două liste precizate prin cursorii `sursa` şi `destinatie`.
  - Fie `y` prima locaţie a listei indicate de `sursa`.
  - Se **suprimă** `y` din lista indicată de `sursa` şi se **înlănţuie** pe prima poziţie a listei indicate de `destinatie`. Acest lucru se realizează astfel:
    - (1) Se salvează valoarea cursorului `destinatie` în locaţia auxiliară `temp`.
    - (2) Se atribuie valoarea cursorului `sursa` cursorului `destinatie` 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 `destinatie`.

- Reprezentarea grafică a acestei acţiuni apare în fig.6.3.3.b unde sunt prezentate legăturile înainte (linie continuă) şi după (linie întreruptă) desfăşurarea ei.

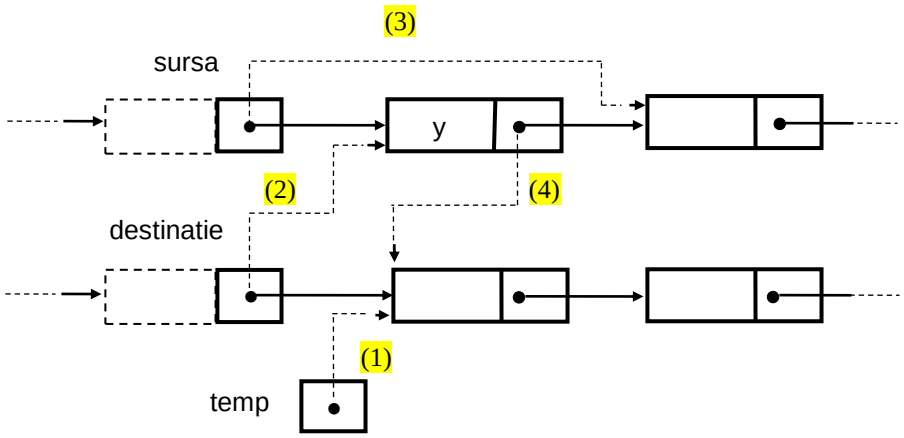

Fig.6.3.3.b. Mutarea unui nod dintr-o listă înlănţuită (`sursă`) în altă listă înlănţuită (`destinaţie`)

- În [6.3.3.b] apare codul care implementează funcţia `Muta`.

```
-------------------------------------------------------------
/*Liste înlănţuite implementate cu ajutorul cursorilor -
Operatorul de gestionare dinamică a memoriei*/

boolean muta(tip_cursor* sursa, tip_cursor* destinatie)
   /*mută elementul indicat de sursă în faţa celui
     indicat de destinaţie şi returnează true în caz de
     reuşită*/
  {
    tip_cursor temp;                          /*[6.3.3.b]*/
    boolean muta_result;
    if(*sursa==0)
        {
          printf("locaţia nu există");
          muta_result= false;
```

```
        }
      else                                /*performanţa O(1)*/
        {
          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 secvenţa [6.3.3.c] se prezintă:
  - Implementarea operatorilor ***insereaza*** şi ***suprima***.
  - Implementarea operatorului ***init*** care înlănţuie iniţial toate elementele tabloului `Zona` în lista indicată de `disponibil`.
    - Procedurile **omit** verificarea erorilor.
    - Se presupune existenţa funcţiei ***muta***.
    - Se precizează că variabila `inceput` indică **începutul listei curente**.

```
/*Liste înlănţuite implementate cu ajutorul cursorilor -
Operatorii insereaza, suprima şi init*/

void insereaza(tip_nod x, tip_cursor p,
               tip_cursor* inceput)        /*[6.3.3.c]*/
  {
    if(p==0){ /*se inserează pe prima poziţie*/
          if(muta(&disponibil,inceput))
            zona[*inceput-1].nodlista= x;
        }                                /*performanţa O(1)*/
      else /*se inserează într-o poziţie diferită de
             prima*/
        if (muta(&disponibil,&zona[p-1].urm))
          /*locaţia  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)                          /*performanţa O(1)*/
        muta(inceput,&disponibil);
      else
        muta(&zona[p-1].urm,&disponibil);
  } /*suprima*/

void init()
   /*iniţializează elementele zonei înlănţuindu-le în
    lista de disponibili*/
```

```
  {
    tip_cursor i;                        /*performanţa O(n)*/
    for (i=lung_max-1;i>=1;i--)
      zona[i-1].urm= i;
    disponibil= 1; zona[lung_max-1].urm= 0;
  } /*init*/
-------------------------------------------------------------
```

6.3.4. Implementarea listelor cu ajutorul referinţelor

- În **limbajele orientate pe obiecte** care **nu** definesc **tipul pointer**, implementarea structurilor de date recursive în general şi a listelor înlănţuite în mod special se poate realiza foarte elegant cu ajutorul **referinţelor**.

- Astfel, pentru a implementa o listă înlănţuită 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; //referinţa la nodul următor al listei

  //constructori                              //[6.3.4.a]
  Nod() {
    //crează un nod cu un element nul şi cu o referinţă
       nulă
    this(null,null);
  }

  public Nod(Object e, Nod n) {
    //crează un nod cu un anumit element e şi o anumită
       referinţă 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 referinţă la nodul de început al listei.
    - În mod opţional pot păstra şi alte informaţii referitoare la listă cum ar fi o referinţă la **ultimul** element al listei şi / sau **numărul** de noduri.
- În secvenţa [6.3.4.b] apare un fragment dintr-o astfel de clasă în care se prezintă structura de date listă ca atare precum şi unele dintre metodele care implementează operatorii specifici.

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

  public InitListaInlantuita() {  //Iniţializează lista
    inceput = null;
    dimensiune = 0;                          //[6.3.4.b]
  }

  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){ //Inserţie 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 excepţia în mod specific;
   obj = inceput.getElement();
   inceput = inceput.getUrm();
   dimensiune--;
   return obj;
```

```
  }

  //alte metode ...
}
```

6.3.5. Comparaţie 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) Operaţiile care se doresc a fi realizate.
    - (3) Frecvenţa cu care sunt invocaţi operatorii.
    - (4) Constrângerile de timp de acces, de memorie, de performanţă, ş.a.
- În orice caz se pot formula următoarele **observaţii**:
    - (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 referinţe**.
    - (2) **Aceiaşi operatori** pot avea performanţe diferite în implementări diferite.
        - Spre exemplu inserţia sau suprimarea unui nod precizat au o durată constantă la **implementarea înlănţuită** (O*(1)*), dar necesită o perioadă de timp proporţională 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)*) respectiv un timp care depinde de poziţia nodului în **implementarea bazată pe pointeri, referinţe** 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 spaţiul maxim solicitat, indiferent de dimensiunea reală a listei la un moment dat.
    - (4) **Implementarea înlănţuită** utilizează în fiecare moment spaţiul de memorie strict necesar lungimii curente a listei, dar necesită în plus spaţiul pentru înlănţuire în cadrul fiecărui nod.
- În funcţie de circumstanţe una sau alta dintre implementări poate fi mai mult sau mai puţin avantajoasă.

6.4. Aplicaţii ale listelor înlănţuite

6.4.1. Problema concordanţei

- **Formularea** problemei:
    - Se dă un **text** format dintr-o succesiune de **cuvinte**.
    - Se baleează textul şi **se depistează cuvintele**.
    - Pentru fiecare cuvânt se verifică dacă este sau nu la **prima apariţie**:
        - În caz că este la prima apariţie, **cuvântul se înregistrează**.
        - În caz că el a mai fost găsit, se incrementează un contor asociat cuvântului care memorează **numărul de apariţii**.
    - În final se dispune de **toate cuvintele distincte** din text şi de **numărul de apariţii** al fiecăruia.
- Se menţionează că această problemă este importantă, deoarece ea reflectă într-o formă simplificată una din activităţile pe care le realizează un **compilator** şi anume **construcţia şi exploatarea listei identificatorilor**.
- Programul **`Concordanta`** rezolvă această problemă utilizând drept suport o listă înlănţuită simplă, în următoarea manieră [6.4.1.a]:
    - Construieşte o listă înlănţuită conţinând cuvintele distincte ale unui text sursă.
    - Iniţial lista este vidă, ea urmând a fi completată pe parcursul parcurgerii textului.
    - Procesul de căutare în listă împreună cu inserţia 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 faţă numărul zero care precizează sfârşitul 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.

```
/*Concordanţa - varianta C*/

#include <stdio.h>                              /*[6.4.1.a]*/
#include <stdlib.h>

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

```
typedef struct tip_nod* tip_referinta;

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 inserţie*/
          {
            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 prelungeşte cu un **nod suplimentar** numit `fanion` (fig 6.4.1.a).

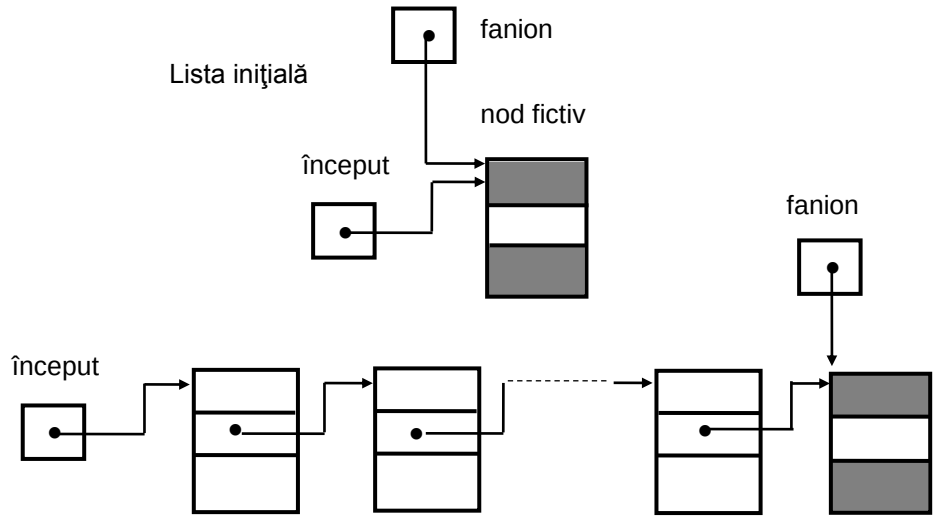

Fig.6.4.1.a. Implementarea unei liste înlănţuite utilizând tehnica nodului fanion

- 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` de `tip_referinta`;
    - Se modifică iniţializarea listei de cuvinte, indicată în cadrul programului prin [******], respectiv instrucţiunea `inceput= null` se înlocuieşte cu secvenţa:

```
inceput= (tip_nod*)malloc(sizeof(tip_nod));
fanion= inceput;
```

Prin aceasta lista de cuvinte conţine de la bun început un nod (cel fictiv).

- În aceste condiţii, procedura **`cauta1`** apare în secvenţa [6.4.1.b].

    - Faţă de varianta [6.4.1.a] condiţia din cadrul instrucţiei **`while`** este mai simplă, realizându-se un câştig simţitor de timp.

- Desigur trebuie modificată şi condiţia de test din procedura **Tiparire** astfel încât să reflecte noua situaţie.

```
-------------------------------------------------------------
/*Căutare în liste înlănţuite utilizând metoda fanionului*/

void cauta1(int x, tip_referinta* inceput) /*[6.4.1.b]*/
  {
    tip_referinta q;
    q= *inceput;
    fanion->cheie= x;  /*setare fanion*/
    while(((tip_nod*)q)->cheie!=x)
     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;
  } /*cauta1*/
-------------------------------------------------------------
```

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 şi sortează.
- În contextul problemei concordanţei, acest lucru se realizează simplu deoarece înainte de inserţia 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 poziţia 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 obţinerea listei sortate, dar procesul de căutare devine mai eficient.
    - Este important de observat faptul că la crearea unor **structuri tablou** sau **secvenţă nu** există posibilitatea simplă de a le obţine 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.
- **Inserţia** unui nod într-o **listă sortată** presupune inserţia unui nod **înaintea** celui indicat de pointerul cu care s-a realizat căutarea.

- O modalitate de rezolvare a unei astfel de situaţii a fost prezentată în paragraful [&6.3.2.1].

- În continuare se va descrie o altă tehnică de inserţie bazată pe utilizarea a **doi pointeri** `q1` şi `q2`, care indică tot timpul **două noduri consecutive ale listei**, conform figurii 6.4.2.a.

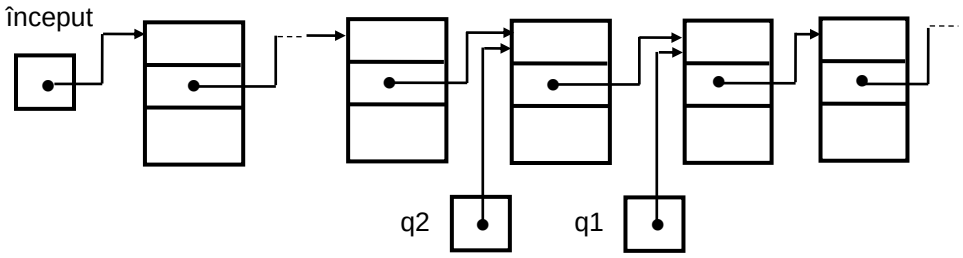

Fig.6.4.2.a. Traversarea unei liste înlănţuite utilizând doi pointeri.

- Se presupune că lista este explorată utilizând metoda nodului `fanion`. Cheia de inserat `x` se introduce iniţial în nodul `fanion`.

- Cei doi pointeri avansează simultan de-a lungul listei până când cheia nodului indicat de `q1` 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 `q1` devine egal cu fanionul.

    - Dacă în acest moment, cheia nodului indicat de `q1` este strict mai mare decât `x` sau `q1 = fanion`, atunci trebuie inserat un nou nod în listă între nodurile indicate de `q2` şi `q1`.

    - În caz contrar, s-a găsit cheia căutată şi trebuie incrementat contorul `q1->numar`.

- În implementarea acestui proces se va ţine cont de faptul că, funcţionarea sa corectă presupune existenţa iniţial în listă a **cel puţin două noduri**, deci cel puţin 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).

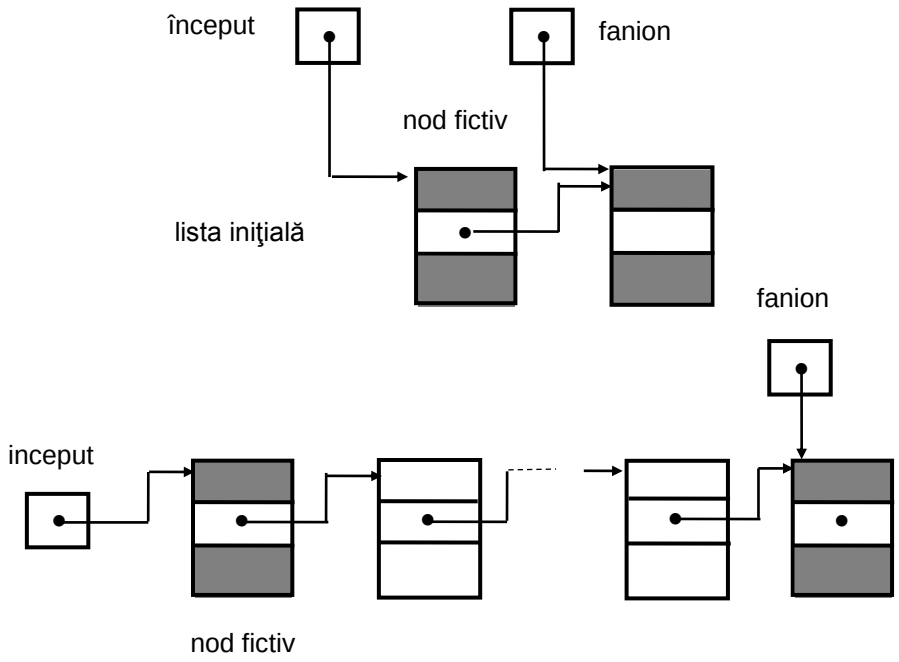

Fig.6.4.2.b. Implementarea unei liste utilizând tehnica celor două noduri fictive.

- În vederea realizării acestor deziderate programul **`Concordanta`** (secvenţa [6.4.1.a]) trebuie modificat după cum urmează:

  - (1) Se adaugă la partea de declaraţie a variabilelor, (locul indicat cu [*****]), declaraţia variabilei `fanion` de `tip_referinta`.

  - (2) Se înlocuieşte instrucţia `inceput=`**`null`** (indicată prin [******]) cu următoarea secvenţă de instrucţiuni care crează **lista vidă** specifică implementării prin **tehnica celor doi pointeri** [6.4.2.a]:

```
-------------------------------------------------------------
 /*iniţializarea listei - tehnica celor doi pointeri*/

 inceput= (tip_nod*)malloc(sizeof(tip_nod)); /*6.4.2.a*/
 fanion= (tip_nod*)malloc(sizeof(tip_nod));
 inceput->urmator= fanion;
-------------------------------------------------------------
```

  - (3) Se înlocuieşte procedura **`cauta`** cu **`cauta2`** secvenţa [6.4.2.b].

```
-------------------------------------------------------------
/*Cautare în liste înlanţuite utilizând tehnica celor doi
 pointeri*/

void cauta2(int x, tip_referinta inceput)  /*[6.4.2.b]*/
  {
    tip_referinta q1,q2,q3;
    q2= inceput;
    q1= ((tip_nod*)q2)->urmator;
    fanion->cheie= x;
    while(q1->cheie<x)
      {
```

```
        q2= q1;
        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 şi
             se inserează între q2^ şi 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** obţinut în urma **sortării** este destul de **limitat**.

    - El se manifestă **numai** în cazul **căutării unui nod** care **nu se găseşte în listă.**

        - Această operaţie necesită parcurgerea în medie a unei **jumătăţi** de listă în cazul listelor **sortate** în comparaţie cu parcurgerea **întregii** liste dacă aceasta **nu** este **sortată**.

    - La căutarea unui nod care **se găseşte 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 şir de variabile aleatoare cu distribuţii 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 acelaşi cuvânt se repetă de puţine 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 şi exploatarea **listei identificatorilor**.

    - În general la parcurgerea unui text sursă, compilatorul inserează în listă fiecare identificator declarat împreună cu o serie de informaţii necesare procesului de compilare.

    - Apariţiile ulterioare ale unor astfel de identificatori, presupun căutarea lor în listă, în vederea obţinerii informaţiilor necesare în procesul de generare a codului.

    - La părăsirea domeniului lor de existenţă, identificatorii sunt suprimaţi din lista indentificatorilor.

- În continuare, pornind de la contextul mai sus precizat, se va prezenta o altă **tehnică de căutare într-o listă înlănţuită** utilizabilă cu precădere la crearea şi **exploatarea listei identificatorilor** de către un compilator.

- Studiindu-se un număr mare de programe sursă considerate tipice, s-a făcut următoarea **constatare experimentală**:

    - Apariţiile unui **identificator** oarecare în textul sursă al unui program au tendinţa de a se "**îngrămădi**" în anumite locuri ale programului, în timp ce în restul textului, apariţia aceluiaşi identificator se produce cu o probabilitate mult mai redusă.

        - Acesta este aşa numitul **principiu al localizării**.

    - Conform acestui principiu, apariţia unui identificator oarecare, poate fi urmată curând, cu mare probabilitate, de una sau mai multe reapariţii.

- Pornind de la această constatare se poate concepe o **metodă** de construire a **listei identificatorilor** care să conducă la o ameliorare substanţială a procesului de căutare.

- Metoda, denumită "**căutare în listă cu reordonare**", constă în aceea că ori de câte ori un identificator se caută şi se găseşte în listă, el se "**mută**" la începutul listei, astfel încât la proxima apariţie 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 secvenţa [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*/

void cauta3(int x, tip_referinta* inceput) /*[6.4.3.a]*/
  {
    tip_referinta q1,q2,q3;
    q1= *inceput;
    ((tip_nod*)fanion)->cheie= x;  /*setare fanion*/
    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;
        } /*if*/
      else
        if(((tip_nod*)q1)->cheie==x)/*este pe primul nod*/
           ((tip_nod*)q1)->numar= ((tip_nod*)q1)->numar+1;
         else
           { /*căutarea*/
            do
```

```
                {
                  q2= q1;
                  q1= ((tip_nod*)q2)->urmator;
                 }while(!(((tip_nod*)q1)->cheie==x));
               if(q1==fanion)  /*nodul x nu există*/
                  { /*inserţie la începutul listei*/
                   q2= *inceput;
                   *inceput= (tip_nod*)malloc(sizeof(tip_nod));
                   ((tip_nod*)(*inceput))->cheie= x;
                   ((tip_nod*)(*inceput))->numar= 1;
                   ((tip_nod*)(*inceput))->urmator= q2;
                  } /*if*/
                 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;
                  } /*else*/
             } /*else*/
  } /*cauta3*/
-----------------------------------------------------------------
```

- Se observă că **reordonarea** presupune existenţa a cel puţin **două noduri în listă**. Acestea pot fi un nod real şi nodul fanion.

    - Deoarece în implementarea realizată s-a utilizat **tehnica nodului fanion**, în cadrul procedurii s-a prevăzut în mod explicit o secvenţă care tratează inserţia primului nod al listei.

    - În acest context, iniţializarea listei se face înlocuind instrucţiunea `inceput = null;` notată cu **`/*/`** în programul [6.4.1.a] cu secvenţa [6.4.3.b]:

```
-----------------------------------------------------------------
/*iniţializarea structurii de date listă*/

inceput= (tip_nod*)malloc(sizeof(tip_nod));  /*[6.4.3.b]*/
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 evidenţă un factor de ameliorare în favoarea celei din urmă cuprins între 1.37 şi 4.7 [Wi76].

    - Ameliorarea mai pronunţată 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 mulţimi elemente peste care a fost definită o relaţie de **ordonare parţială**, adică ordonarea este valabilă doar pentru anumite perechi de elemente, **nu** pentru toate.

- Aceasta este de fapt o situaţie reală ilustrată prin următoarele **exemple**:

  - (1) Într-un **dicţionar**, **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 **dicţionar** înseamnă aranjarea lor într-o astfel de ordine încât să **nu** existe **nici o referinţă în faţă** (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 execuţie 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 iniţializarea ori cărui subtask, toate subtaskurile care-l condiţionează să fie terminate.

  - (3) Într-un **program** anumite proceduri pot conţine apeluri la alte proceduri.

    - Dacă o procedură v este apelată de o procedură w, aceasta se va preciza prin notaţia $v \prec w$.

    - **Sortarea topologică** presupune un astfel de aranjament al declaraţiilor de proceduri încât să nu existe nici o referire înainte.

- În general, **ordonarea parţială** a unei **mulţimi** M presupune **existenţa** unei **relaţii** între unele dintre elementele lui M.

- Această **relaţie** este desemnată prin simbolul $\prec$, care înseamnă "precede" şi satisface următoarele trei proprietăţi:

  - (1) Dacă $x \prec y$ şi $y \prec z$ 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 mulţimea M a elementelor care urmează a fi sortate topologic este **finită**.

- În aceste condiţii o **relaţie de ordonare parţială** poate fi ilustrată printr-o **diagramă** sau **graf** în care **nodurile** desemnează elementele iar **săgeţile orientate** reprezintă relaţiile dintre ele.

  - Un exemplu de **relaţie de ordonare parţială** reprezentată grafic apare în fig.6.4.4.a.

- **Problema sortării topologice** este aceea de a transforma **ordinea parţială** într-o **ordine liniară.**

- Din punct de vedere grafic, aceasta implică rearanjarea nodurilor într-un şir astfel încât toate săgeţile să indice acelaşi sens, spre exemplu spre dreapta (fig.6.4.4.b).

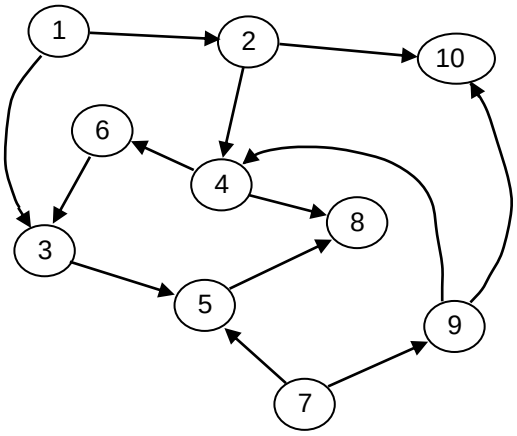

Fig.6.4.4.a. Mulţime de elemente parţial ordonată

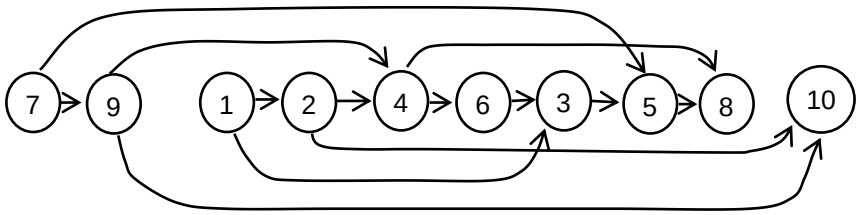

Fig.6.4.4.b. Ordonarea liniară a unei mulţimi parţial ordonate

- Proprietăţile (1) şi (3) ale relaţiei de ordonare parţială garantează faptul că **graful nu conţine bucle.**

    - Aceasta este de fapt **condiţia necesară şi suficientă** pentru ca **ordonarea liniară** a elementelor mulţimii, 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 puţin unul, altfel graful conţine o buclă.

    - (2) Elementul este extras din mulţimea M şi este plasat în lista ordonată liniar care se creează.

    - (3) Mulţimea rezultată rămâne parţial **ordonată** şi în consecinţă poate fi aplicat din nou acelaşi algoritm, până când ea devine vidă.

- Pentru a descrie acest algoritm mai riguros, trebuiesc precizate **structurile de date**, **reprezentarea** mulţimii M şi a **relaţiei de ordonare**.
    - Este evident faptul că alegerea acestor reprezentări este determinată de operaţiile care urmează să fie realizate, în mod particular de **operaţia de selecţie** 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 mulţimii M **nu** este cunoscut a priori, mulţimea iniţială este de regulă organizată ca o listă înlănţuită numită **lista principalilor**.
    - În consecinţă, fiecare element trebuie să mai conţină în plus, legătura la elementul următor al listei principalilor.
    - Pentru simplificare se va presupune că identificatorii elementelor mulţimii, respectiv cheile asociate, sunt numere întregi (nu neapărat consecutive) cuprinse între `1` şi `n`.
- În mod analog, mulţimea **succesorilor unui element** va fi reprezentată tot ca o listă înlănţuită.
    - Fiecare element al **listei succesorilor** este descris:
        - (1) Prin **identificatorul propriu** care este de fapt o referinţă în **lista principalilor**.
        - (2) Printr-o **legătură** la următorul element al **listei succesorilor** numită şi **lista secundarilor**.
- Nodurile **listei principale**, în care fiecare element al mulţimii M apare exact o singură dată, se vor numi **principali**.
- Nodurile corespunzătoare elementelor din **lista succesorilor**, se vor numi **secundari**.
- În aceste condiţii se pot defini următoarele **structuri de date** [6.4.4.a].

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

#include <stdlib.h>                          /*[6.4.4.a]*/

/*definire tip indicator nod în lista principalilor*/
typedef struct tip_nod_principal* tip_pointer_principali;

/*definire tip indicator nod în lista secundarilor*/
typedef struct tip_nod_secundar* tip_pointer_secundari;

typedef int tip_cheie;

/*definire structură nod în lista principalilor*/
```

```
typedef struct tip_nod_principal {
   tip_cheie cheie;
   int contor;  /*număr de predecesori*/
   tip_pointer_principali urm;
   tip_pointer_secundari secund;  /*lista secundarilor*/
} tip_nod_principal; /*nod în lista principalilor*/

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

- Se presupune că mulţimea M şi relaţiile sale de ordonare sunt furnizate iniţial ca o secvenţă 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.

$1 \prec 2$ $2 \prec 4$ $4 \prec 6$ $2 \prec 10$ $4 \prec 8$ $6 \prec 3$ $1 \prec 3$ [6.4.4.b]
$3 \prec 5$ $5 \prec 8$ $7 \prec 5$ $7 \prec 9$ $9 \prec 4$ $9 \prec 10$

- (1) Pentru început, programul de **sortare topologică** trebuie să citească **datele de intrare** şi să construiască **structurile de date** aferente.

    - Aceasta se realizează prin citirea succesivă a perechilor de elemente `x` şi `y` (`x` $\prec$ `y`).
    - Nodurile corespunzătoare sunt localizate prin căutare în **lista principalilor** şi dacă **nu** sunt găsite sunt **inserate** în această listă la **sfârşitul** ei.
        - Această sarcină este îndeplinită de funcţia **`caut`** care returnează pointerul nodului căutat.
    - Pointerii corespunzători celor două elemente `x` şi `y` în **lista principalilor** sunt notaţi cu `p` şi `q`.
    - În continuare în **lista secundarilor** lui `x` se adaugă prin inserţie în faţă, 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 numeşte **`faza initiala`** şi apare în secvenţa [6.4.4.c].

    - Fragmentul de program prezentat utilizează funcţia **`caut`**`(w)` care returnează pointerul nodului din lista principalilor care are cheia `w`.
    - Se presupune că secvenţa perechilor de chei furnizate la intrare se încheie cu o cifră 0.
    - Pentru setul de date iniţiale din [6.4.4.b], structura de date construită drept urmare a fazei iniţiale a algoritmului de sortare topologică apare în figura 6.4.4.c.

```
tip_pointer_principali inceput,sfarsit;    /*[6.4.4.c]*/
tip_pointer_principali p,q;
tip_pointer_secundari t;
tip_cheie x,y; /*chei*/
int z;  /*contor elemente procesate*/

/*sortarea topologică - faza iniţială*/

scanf("%i", &x);
/*iniţializarea listei principalilor*/
inceput=
(tip_nod_principal*)malloc(sizeof(tip_nod_principal));
sfarsit= inceput;
z= 0;  /*contor elemente procesate*/
while(x!=0)
  {
    scanf("%i", &y); /*citeşte nodul y*/
    p= caut(x); /*caută/inserează nodul x în principali*/
    q= caut(y); /*caută/inserează nodul y în principali*/
    /*crează un nou succesor*/
    t= (tip_nod_secundar*)malloc(sizeof(tip_nod_secundar));
    t->id= q; /*noul nod indică pe y (q)*/
    /*inserează y (q) în faţa listei succesorilor lui x
      (p)*/
    t->urm= p->secund;
    p->secund= t;
   /*incrementează numărul de predecesori ai lui y (q)*/
    q->contor= q->contor+1;
    scanf("%i", &x); /*citeşte următorul nod x*/
  } /*while*/
```

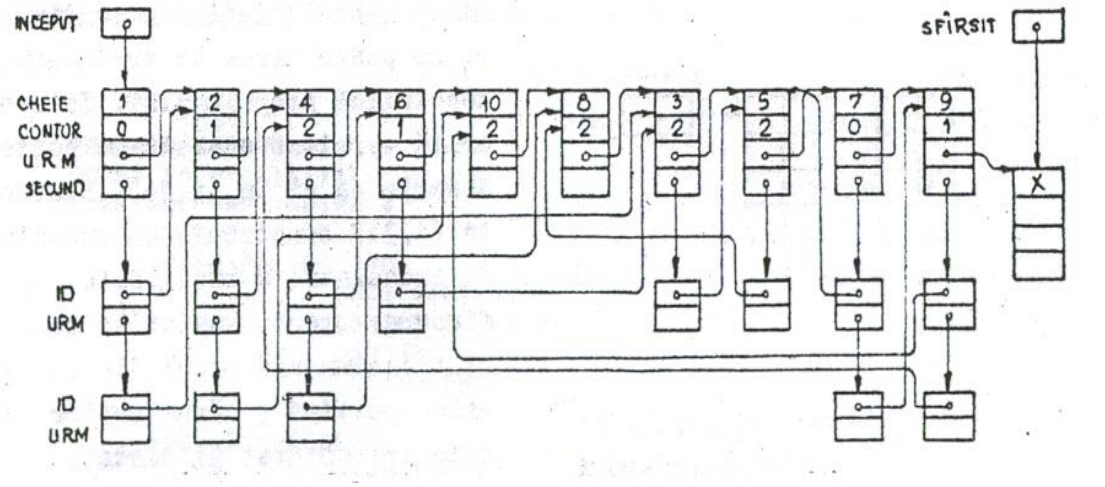

Fig.6.4.4.c. Structură de date pentru sortarea topologică

- (2) După ce **structura de date** a fost construită, urmează realizarea procesului de **sortare topologică** descris anterior.

    - **Sortarea topologică** constă în principiu în **selecţia repetată** a unui element cu contorul de predecesori nul.

- În consecinţă, pentru început se caută **nodurile cu zero predecesori** şi se înlănţuie într-o listă numită **lista principalilor cu zero predecesori.**
- Întrucât nu mai este nevoie de lista iniţială a principalilor, câmpul de înlănţuire `urm` va fi utilizat pentru a înlănţui nodurile cu zero predecesori în **lista principalilor cu zero predecesori.**

- Această operaţie prin care se înlocuieşte o listă printr-o altă listă apare deosebit de frecvent în procesul de prelucrare a listelor.

- Ea apare în detaliu în secvenţa [6.4.4.d] în care, din raţiuni de convenienţă noua listă este construită în sens invers prin inserţie în faţă.

```
-------------------------------------------------------------
/*căutarea principalilor cu zero predecesori*/

p= inceput; /*lista principalilor*/        /*[6.4.4.d]*/
inceput= null; /*lista principalilor cu zero predecesori*/
while(p!=sfarsit)
  {
    q= p; p= q->urm;
    if(q->contor==0)
      {     /*inserează nodul q la începutul noii liste*/
        q->urm= inceput; inceput= q;
      } /*if*/
  } /*while*/
-------------------------------------------------------------
```

- (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] şi poartă denumirea de **`faza de ieşire`.**

  - Ambele secvenţe constituie exemple de **traversare a listelor**.
  - În fiecare moment, variabila `p` desemnează nodul din **lista principalilor** al cărui contor trebuie decrementat şi testat.

```
-------------------------------------------------------------
/*faza de ieşire*/

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

while(t!=null)                                /*[6.4.4.f]*/
```

```
  {
    p= t->id; /*p este succesorul curent al lui q*/
    p->contor= p->contor-1; /*decrementează contorul de
                              predecesori ai lui p*/
    if(p->contor==0)
      {  /*inserează nodul p la începutul listei
           principalilor cu zero predecesori*/
        p->urm= q;
        q= p;
      } /*if*/
    t= t->urm; /*următorul succesor*/
  } /*while*/
```

- **Contorul** `z` a fost introdus pentru contoriza nodurile principale generate în **`faza initiala`** ([6.4.4.c]).
  - Acest contor este decrementat de fiecare dată când un element principal este scris în **`faza de iesire`**.
  - 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 mulţimea M **nu este parţial ordonată** şi în consecinţă **nu poate fi sortată topologic**.
- `Faza de iesire` este un exemplu de proces care gestionează o listă care pulsează, în care elementele sunt inserate şi suprimate într-o ordine impredictibilă.
- Codul integral al programului **`TopSort`** apare în secvenţa [6.4.4.g].

```
/*Programul TopSort*/

#include <stdio.h>                              /*[6.4.4.g]*/
#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;
   tip_pointer_secundari secund;
 } tip_nod_principal;

typedef struct tip_nod_secundar
 {
   tip_pointer_principali id;
   tip_pointer_secundari urm;
 } tip_nod_secundar;
```

```
tip_pointer_principali inceput,sfirsit;
tip_pointer_principali p,q;
tip_pointer_secundari t;
int z;  /*contor elemente*/
tip_cheie x,y;  /*chei*/

tip_pointer_principali caut(tip_cheie w)
/*furnizează referinţa la nodul principal cu cheia w; dacă
un astfel de nod nu există, este creat şi inserat în coada
listei*/
  {
    tip_pointer_principali h;
    tip_pointer_principali 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=
      (tip_nod_principal*)malloc(sizeof(tip_nod_principal));
        z= z+1; /*incrementează contorul de elemente*/
        h->contor= 0;
        h->secund= null; /*iniţializare lista secundari*/
        h->urm=sfirsit;
      } /*if*/
    caut_result= h;
    return caut_result;
  } /*caut*/

int main(int argc, const char* argv[])
{
  /*se iniţializează lista principalilor cu un nod fictiv*/
  inceput=
     (tip_nod_principal*)malloc(sizeof(tip_nod_principal));
  sfirsit= inceput;
  z= 0; /*iniţializare contor de elemente*/

  /*faza iniţială - construcţia listei principalilor*/
  scanf("%i", &x);
  while (x!=0)
    {
      scanf("%i", &y);
      printf("%i %i\n", x,y);
      p= caut(x);
      q= caut(y);
      t=
        (tip_nod_secundar*)malloc(sizeof(tip_nod_secundar));
      t->id= q;
      t->urm= p->secund;
      p->secund= t;
      q->contor= q->contor+1;
      scanf("%i", &x);
    }
  /*construcţia listei 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 - sortarea topologică*/
  q= inceput;
  while (q!=null){
      printf("%i", q->cheie);
      z= z-1; /*decrementează contorul de elemente*/
      t= q->secund;
      q= q->urm;
      while (t!=null){
          p= t->id;
          p->contor= p->contor-1;
          if (p->contor==0){   /*inserare nod p în lista
                       principalilor cu zero predecesori*/
              p->urm= q;
              q= p;
            }
          t= t->urm;
        }
    }
  if (z!=0)
    printf("Multimea nu este partial ordonata!\n");
  return 0;
} /*main*/
```

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 înlănţuite, stivele şi cozile**.

- De asemenea se prezintă **funcţia de asociere a memoriei** precum şi 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 posibilităţi de implementare.

6.5.1. Liste circulare

- **Listele circulare** sunt liste înlănţuite ale căror înlănţuiri se **închid**.

- În aceste condiţii se pierde noţiunea de început şi sfârşit, 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 **inserţia primului nod** în listă şi la **suprimarea ultimului nod**.

- O modalitate simplă de rezolvare a acestor situaţii este aceea de a utiliza un **nod fictiv** într-o manieră asemănătoare celei prezentate la listele obişnuite (**tehnica nodului fictiv** &6.3.2).

    - Această modalitate este ilustrată în figura 6.5.1.b.

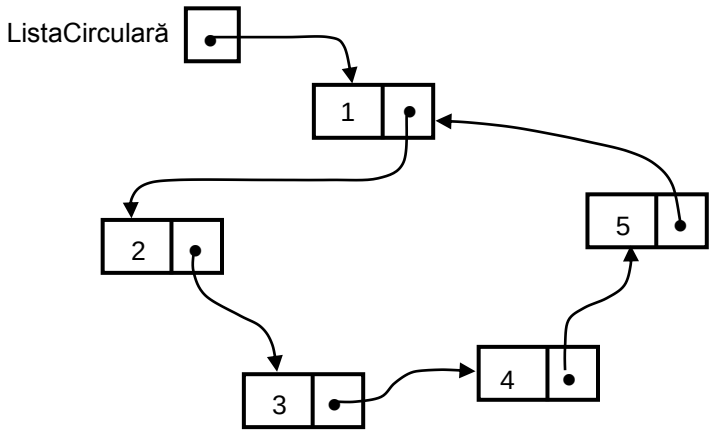

Fig.6.5.1.a. Listă circulară

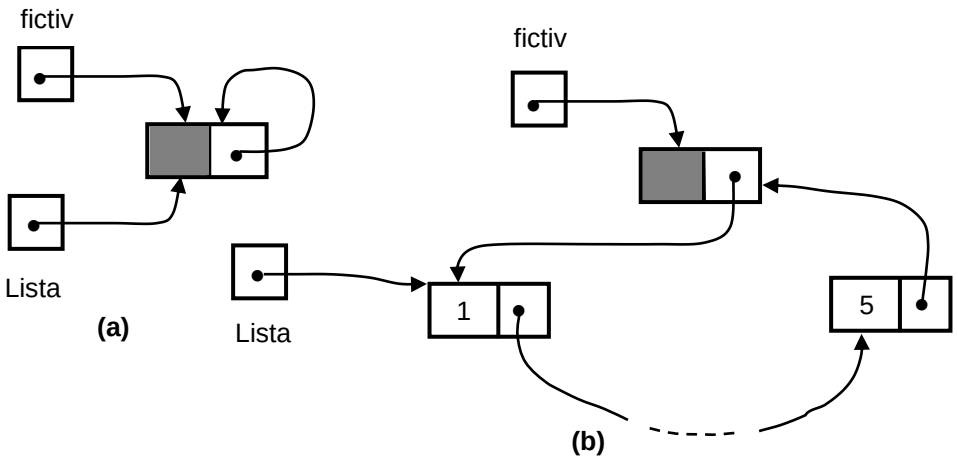

Fig.6.5.1.b. Listă circulară implementată prin tehnica nodului fictiv.
Listă vidă (a), listă normală (b)

6.5.2. Liste dublu înlănţuite

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

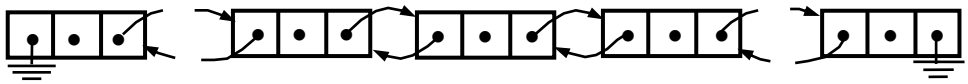

Fig.6.5.2.a. Listă dublu înlănţuită

    - Preţul care se plăteşte este:
        - (1) Prezenţa unui câmp suplimentar de tip pointer în fiecare nod.
        - (2) O oarecare creştere a complexităţii 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 secvenţa [6.5.2.a].

```
-------------------------------------------------------------
/*Lista dublu înlanţuită*/

typedef struct tip_nod* tip_pointer_nod;    /*[6.5.2.a]*/

typedef struct tip_nod
 {
   tip_element element;
   tip_pointer_nod anterior,urmator;
 } tipnod;

typedef tip_pointer_nod tip_lista_dublu_inlantuita;
-------------------------------------------------------------
```

- Pentru exemplificare se prezintă procedura de **suprimare** a elementului situat în poziţia `p` a unei liste dublu înlănţuite.

- În secvenţa [6.5.2.b] se prezintă maniera în care se realizează această acţiune, în accepţiunea faptului că nodul suprimat **nu** este nici **primul** nici **ultimul nod al listei**.
    - (1) Se localizează **nodul precedent** şi 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 spaţiul de memorie afectat lui poate fi reutilizat în regim de alocare dinamică a memoriei.

```
-------------------------------------------------------------
/*Liste dublu înlănţuite - suprimarea unui nod*/

void suprima(tip_pozitie* 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 înlănţuite**, derivate din tehnicile de implementare a listelor liniare.
    - Aceste tehnici simplifică implementarea operatorilor care prelucrează astfel de liste în mod deosebit în situaţii limită (listă vidă sau listă cu un singur nod).
- (1) O primă posibilitate o reprezintă **lista dublu înlănţuită cu două noduri fictive** (fig.6.5.2.b).
    - Cele două noduri fictive (`Fict1` şi `Fict2`) permit ca **inserţia** primului nod al listei respectiv **suprimarea** ultimului nod să se realizeze în manieră similară oricărui alt nod al listei.

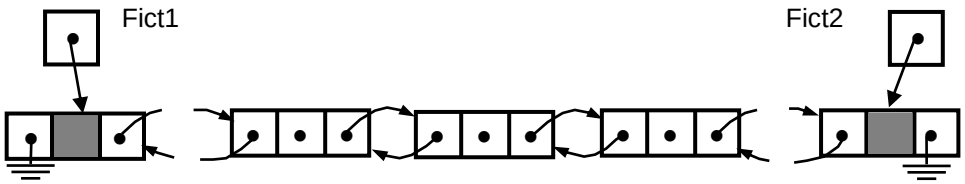

Fig.6.5.2.b. Listă dublu înlănţuită. Varianta cu două noduri fictive

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

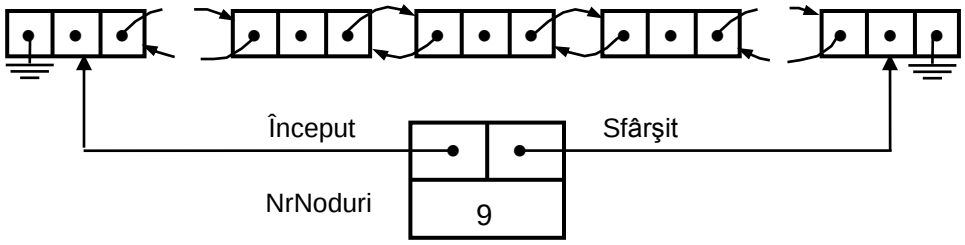

Fig.6.5.2.c. Listă dublu înlănţuită. Varianta cu indicatori la capete

- (3) Listele dublu înlănţuite pot fi implementate şi ca **liste circulare**.
    - În figurile 6.5.2.d respectiv 6.5.2.e apare o astfel de listă în două reprezentări grafice echivalente.

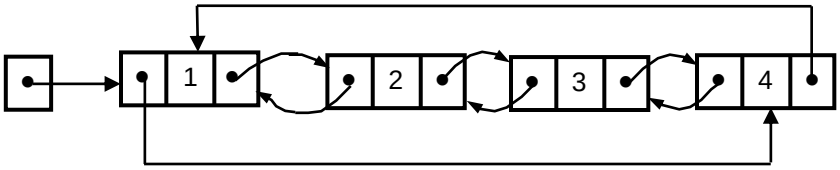

Fig.6.5.2.d. Listă dublu înlănţuită circulară

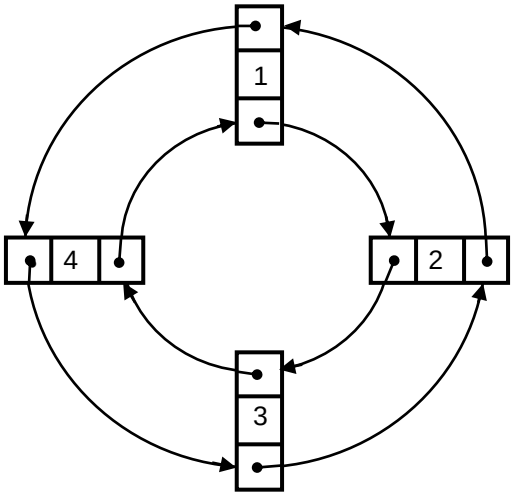

Fig.6.5.2.e. Listă dublu înlănţuită circulară

- (4) Este de asemenea posibil a se utiliza la implementarea listelor dublu înlănţuite 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 `urmator` pe primul (fig.6.5.2.f).
    - Când lista este vidă, ambele înlănţuiri indică chiar nodul fictiv.

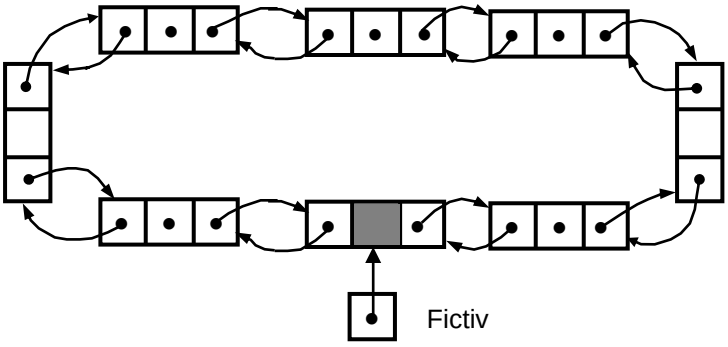

Fig.6.5.2.f. Listă dublu înlănţuită circulară. Varianta cu nod fictiv

6.5.3. Stive

- O **stivă** este un tip special de listă în care toate inserările şi suprimările se execută la un singur capăt care se numeşte **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 cărţi sau al unui vraf de farfurii pe o masă.

    - În mod evident cea mai convenabilă şi în acelaşi timp cea mai sigură manieră de a lua un obiect sau de a adăuga un altul este, din motive uşor de înţeles, aceea de a acţiona 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) Notaţiilor utilizate.
    - (3) Operatorilor definiţi pentru acest tip.

- Toate aceste elemente apar precizate în secvenţa [6.5.3.1.a].

```
-------------------------------------------------------------
TDA Stivă

Modelul matematic: o secvenţă finită de noduri. Toate
     nodurile aparţin unui acelaşi tip numit tip de bază. O
     stivă este de fapt o listă specială în care toate
     inserţiile şi toate suprimările se fac la un singur
     capăt care se numeşte vârful stivei.

Notaţii:
```

```
TipStiva s;
TipElement x;
boolean b;                                   [6.5.3.1.a]
```

Operatori:

1. ***Initializeaza***(*TipStiva s*) - face stiva *s* vidă.

2. *TipElement* ***VarfSt***(*TipStiva s*) - furnizează elementul din vârful stivei *s*.

3. ***Pop***(*TipStiva s*) - suprimă elementul din vârful stivei.

4. ***Push***(*TipElement x, TipStiva s*) - inserează elementul *x* în vârful stivei *s*. Vechiul vârf devine elementul următor ş.a.m.d.

5. *boolean* ***Stivid***(*TipStiva s*) - returnează valoarea adevărat dacă stiva *s* este vidă şi fals în caz contrar.

- În secvenţa [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)*/

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

/*Initializeaza(s:tip_stiva);*/  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 acelaşi timp şi un exemplu de **implementare ierarhică** a unui **tip de date abstract** care ilustrează:

  - Pe de o parte **flexibilitatea** şi **simplitatea** unei astfel de abordări.
  - Pe de altă parte, **invarianţa** ei în raport cu **nivelurile ierarhiei**.

- Cu alte cuvinte, **modificarea** implementării **TDA Listă nu** afectează sub nici o formă implementarea **TDA Stivă** în accepţiunea păstrării nemodificate a **prototipurilor operatorilor definiţi**.

- Utilizarea deosebit de frecventă şi cu mare eficienţă a **structurii de date stivă** în domeniul programării, a determinat **evoluţia** acesteia de la statutul de **structură de date avansată** spre cel de **structură fundamentală**.

- Această tendinţă s-a concretizat în **implementarea hardware** a acestei structuri în toate sistemele de calcul moderne şi în includerea operatorilor specifici tipului de date abstract stivă **în setul de instrucţii 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 şi pentru stive.

- În particular, reprezentarea stivelor ca **liste înlănţuite** nu ridică nici un fel de probleme, operatorii ***`push`*** şi ***`pop`*** operând doar cu **pointerul de început** şi cu primul nod al listei.

- În ceea ce priveşte implementarea **TDA Stivă** cu ajutorul **tablourilor**:

- Implementarea listelor bazată pe **structura tablou** prezentată în (&6.3.1) **nu** este cea mai propice.

    - **Explicaţia**: fiecare ***`push`*** şi fiecare ***`pop`*** necesită mutarea întregii stive, activitate care necesită un consum de timp proporţional cu numărul de elemente ale stivei.

- O utilizare mai eficientă a structurii tablou ţine cont de faptul că inserţiile şi suprimările se fac **numai** în **vârful stivei**.

    - Astfel se poate considera drept **bază a stivei** sfârşitul tabloului (indexul său cel mai mare), stiva crescând în sensul descreşterii indexului în tablou.

    - Un **indicator** numit `vârf` indică poziţia curentă a ultimului element al stivei (fig.6.5.3.2.a).

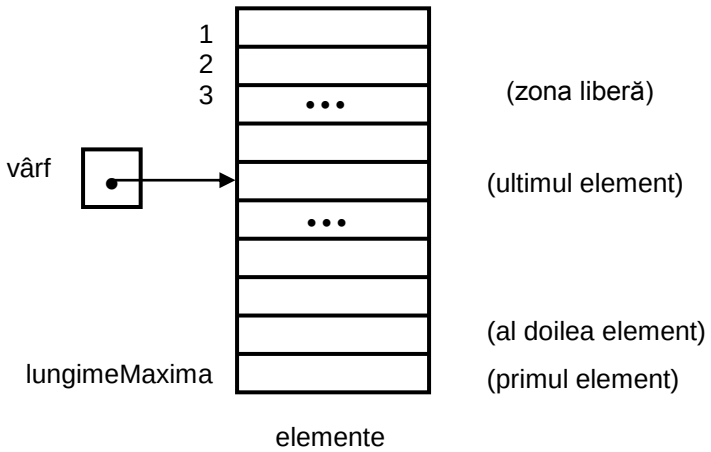

Fig.6.5.3.2.a. Implementarea TDA Stivă cu ajutorul structurii tablou

- Structura de date abstractă care se defineşte pentru această implementare este următoarea [6.5.3.2.a].

```
-----------------------------------------------------------------
/*Implementarea stivelor cu ajutorul tablourilor*/

enum { lungime_maxima = 50};                    /*[6.5.3.2.a]*/

typedef struct tip_stiva
{
  int varf;
  tip_element elemente[lungime_maxima];
} tip_stiva;
-----------------------------------------------------------------
```

- O instanţă a stivei constă din secvenţa `elemente[varf], elemente [varf+1], ...,elemente[lungime_maxima]`.
- **Stiva** este **vidă** dacă `varf = lungime_maxima+1`.
- **Operatorii** specifici acestei implementări sunt prezentaţi în [6.5.3.2.b].

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

#include <stdio.h>                              /*[6.5.3.2.b]*/

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
        varfst_result= s->elemente[s->varf-1];
    return varfst_result;
  }  /*varfst*/
```

```
void pop(tip_stiva* s, tip_element* x)
  {
    boolean er;
    if (stivid(*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 **funcţie** care **evaluează expresia** în cauză.
- În această situaţie se poate utiliza următoarea **tehnică**:
  - (1) Se parcurge expresia de la stânga la dreapta.
  - (2) Se execută operaţiile pe măsură ce se întâlnesc, înlocuind de fiecare dată grupul **[operand, operand, operator]** cu valoarea obţinută în urma evaluării.
- Se presupune că **forma corectă postfix** a expresiei aritmetice de evaluat este memorată în **lista** `PostFix`, ale cărei noduri conţin fie un operand fie un operator.
- Funcţia **`evaluare_postfix`** realizează **evaluarea expresiei** utilizând **stiva** `eval` în care memorează valori numerice aflate în aşteptarea execuţiei operaţiilor aritmetice necesare [6.5.3.3.a].

```
-------------------------------------------------------------
float evaluare_postfix (tip_lista postfix)
/*Evaluează o expresie postfix reprezentată ca şi o listă de
noduri. Se presupune ca expresia postfix este corectă. Se
utilizează ADT Stiva*/
```

```
{
  tip_stiva eval;                          /*[6.5.3.3.a]*/
  float nr_virf,nr_secund,raspuns;
  nod_postfix t;

  float evaluare_postfix_result;
  initializeaza(&eval);
  while *mai exista noduri în lista postfix
    {
      *citeste un nod al listei postfix în t;
      if (t este un operand)
          push(t,&eval); /*t este un operand*/
        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);
          } /*else*/
      *avanseaza la nodul urmator al listei postfix;
    } /*while*/
  evaluare_postfix_result= varfst(&eval);
  return evaluare_postfix_result;
}   /*evaluare_postfix*/
```

- **Exemplul 6.5.3.3.b.** Eliminarea recursivităţii.

- După cum s-a precizat în &5.2.4, recursivitatea poate fi **eliminată** practic în orice situaţie.

    - Eliminarea ei poate conduce la creşterea performanţei, însă algoritmul devine de regulă mai complicat şi mai greu de înţeles.

- Scopul acestui paragraf este acela de a prezenta un exemplu de **caz general** al eliminării recursivităţii, 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 soluţie **recursivă** şi una **nerecursivă** a unei versiuni simplificate a problemei clasice a "**sacului călătorului**" ("knapsack problem").

- Formularea problemei:

    - Fiind dată o **valoare** `gt` (greutatea totală) şi un **set de greutăţi** reprezentate prin întregi pozitivi, $a_1, a_2, \ldots, a_n$.

    - Se cere să se afle dacă greutăţile se pot selecta astfel încât **suma lor să fie exact** `gt`.

- Spre exemplu dacă `gt = 10` şi greutăţile sunt 7, 5, 4, 4, 1 atunci pot fi selectate a doua, a treia şi 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 ţine cont atât de greutatea obiectelor selectate cât şi de valoarea lor, a fost discutată şi prezentată în variantă recursivă.

- Algoritmul **recursiv** care rezolvă problema în varianta sa **simplă** apare în secvenţa [6.5.3.3.b].

```
-------------------------------------------------------------
/*Problema Knapsack - Problema sacului călătorului -
formularea recursivă*/

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

- Funcţia recursivă **`knapsack`** operează asupra unui **tablou** `g` de întregi care memorează **greutăţile obiectelor**.

    - Un apel al funcţiei **`knapsack`**`(s,i)` stabileşte dacă există un set de elemente cuprinse în domeniul `g[i], g[n]` a căror sumă este exact `s` şi în caz afirmativ le tipăreşte.

- **Modul de lucru** preconizat este următorul.

    - În primul rând funcţia verifică dacă:

        - (1) `gt = 0` caz în care soluţia problemei a fost găsită şi rezultatul este un succes.

        - (2) `gt < 0` sau `i > n`, caz în care nu s-a găsit o sumă egală cu `gt` sau, au fost parcurse toate greutăţile fără a se găsi o sumă egală cu `gt`.

- Dacă nici una din aceste situaţii **nu** este valabilă, atunci se apelează **`knapsack`**`(gt-g[i],i+1)` pentru a verifica dacă există o soluţie care îl **include** pe `g[i]`.
- Dacă există o astfel de soluţie, atunci se afişează `g[i]`.
- Dacă **nu** există o astfel de soluţie, se apelează **`knapsack`**`(gt,i+1)`, pentru a verifica dacă există o soluţie 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 şi gestionate de către programator.
  - Un **nod** al acestei stive conţine următoarele elemente (contextul apelului):
    - (1) Valorile curente ale parametrilor de apel ai procedurii.
    - (2) Valorile curente ale tuturor variabilelor locale ale procedurii.
    - (3) O indicaţie referitoare la adresa de retur, adică referitoare la **locul** în care revine controlul execuţiei în momentul în care apelul curent al instanţei procedurii se termină.
- În cazul procedurii **`knapsack`**, lucrurile se pot simplifica în baza următoarelor raţionamente.
- (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 candidaţi creşte cu 1.
  - În aceste condiţii, variabila `candidat` poate fi definită ca şi o **variabilă globală** care este incrementată cu 1 la fiecare introducere în **stivă** şi 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 funcţie este:
    - Fie situată într-o **altă procedură** care a apelat iniţial funcţia **`knapsack`**.
    - Fie este **adresa următoare** unuia dintre cele două apeluri recursive din interiorul funcţiei, însemnate cu [*] – (includerea candidatului) respectiv [**] – (excluderea candidatului) în secvenţa [6.5.3.3.b].
  - Cele trei situaţii pot fi modelate printr-o variabilă de tip `stare` care poate avea una din următoarele valori:
    - `extern` - indicând un apel din afara funcţiei **`knapsack`**.
    - `inclus` - indicând un apel recursiv din locul precizat de [*], apel care îl include pe `g[candidat]` în soluţie.
    - `exclus` - indicând un apel recursiv din locul precizat de [**], care îl exclude pe `g[candidat]`.

- Dacă se memorează valoarea variabilei de tip `stare` ca şi un indiciu pentru **adresa de revenire**, atunci `gt` poate fi tratată ca şi 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 soluţie, se utilizează variabila booleană globală `victorie`.
    - O dată pusă pe adevărat, `victorie` rămâne adevărată şi determină **golirea stivei** şi afişarea greutăţilor aflate în starea `inclus`.
- În noile condiţii, **stiva** ca atare va fi declarată ca o **listă de stări** [6.5.3.3.c]:

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

typedef enum {extern_, inclus, exclus} stare;
                                              /*[6.5.3.3.c]*/
tip_stiva ... /*o declaraţie potrivită de stivă având
              noduri de tip stare*/
-------------------------------------------------------------
```

- Având în vedere cele mai sus precizate, în secvenţa [6.5.3.3.d] apare **procedura nerecursivă `knapsack`** care:
    - Operează asupra unui tablou de greutăţi `g`.
    - Este concepută în termenii operatorilor definiţi asupra unei **structuri de date abstracte stivă.**
- Deşi această procedură poate fi mai rapidă decât funcţia **`knapsack`** recursivă:
    - Este specifică strict problemei în cauză.
    - Conţine în mod evident mai multe linii de cod.
    - Este mai complicată.
    - Este mai dificil de înţeles.
- Din aceste motive eliminarea recursivităţii se recomandă a fi utilizată atunci când factorul **viteză** este deosebit de important.

```
-------------------------------------------------------------
/*utilizează ADT Stiva*/

void knapsack_nerecursiv (int gt)          /*[6.5.3.3.d]*/
   {
     int candidat;
     boolean victorie;
     tip_stiva s;

     candidat= 1;
```

```
    victorie= false;
    initializeaza(&s);
    push(extern_,&s); /*initializare stivă cu g[1]*/
    do {
      if(victorie)
          {  /*golire stivă şi afişarea greutăţilor
                 incluse în soluţie*/
            if (varfst(&s)==inclus)
              printf("%i\n", g[candidat-1]);
            candidat= candidat-1;
            pop(&s, 0);
          }
        else
          if(gt==0)
              {  /*soluţie găsită*/
                victorie= true;
                candidat= candidat-1;
                pop(&s, 0);
              }
            else
              if((gt<0 && (varfst(&s)==extern_))) ||
                  (candidat>n))
                  {  /*nici o soluţie nu este posibilă
                         cu această alegere*/
                    candidat= candidat-1; pop(&s, 0);
                  }
                else /*nu există încă o soluţie; se
                  analizează 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
                              selecţie 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**) şi sunt suprimate la celălalt **(început**).

    - Cozile se mai numesc liste "**FIFO**" ("**first-in first-out**") adică liste de tip "**primul-venit-primul-servit**".

- Operaţiile care se execută asupra **cozii** sunt analoage celor care se realizează asupra **stivei** cu diferenţa că inserările se fac la **spatele** cozii şi **nu** la **începutul** ei şi că ele diferă ca şi terminologie.

6.5.4.1. Tipul de date abstract Coadă

- În acord cu abordările anterioare în secvenţa [6.5.4.1.a] se definesc **două** variante ale **TDA Coadă.**

- În secvenţa [6.5.4.1.b] se prezintă un exemplu de implementare a **TDA Coadă** bazat pe **TDA Listă**.

TDA Coadă

Modelul matematic: o secvenţă finită de noduri. Toate nodurile aparţin unui aceluiaşi tip numit **tip de bază.** O **coadă** este de fapt o listă specială în care toate inserţiile se fac la un capăt (spate) şi toate suprimările se fac la celălalt capăt (cap).

Notaţii:

```
TipCoada C;                                    [6.5.4.1.a]
TipElement x;
Boolean b;
```

Operatori (setul 1):

1. ***Initializeaza****(TipCoada C)* - face coada *C* vidă.

2. *TipElement* ***Cap****(TipCoada C)* - funcţie care returnează primul element al cozii *C*.

3. ***Adauga****(TipElement x, TipCoada C);* - inserează elementul *x* la spatele cozii *C*.

4. ***Scoate****(TipCoada C);* - suprimă primul element al lui *C*.

5. boolean ***Vid****(TipCoada C)* - este adevărat dacă şi numai dacă *C* este o coadă vidă.

Operatori (setul 2):

1. ***CreazaCoadaVida****(TipCoada C)* - face coada *C* vidă.

2. *boolean* ***CoadaVida****(TipCoada C)* - este adevărat dacă şi

numai dacă *C* este o coadă vidă.

3. *boolean* ***CoadaPlina****(TipCoada C* - este adevărat dacă şi numai dacă *C* este o coadă plină (operator dependent de implementare).

4. ***InCoada****(TipCoada C, TipElement x)* - inserează elementul *x* la spatele cozii *C* (***EnQueue***).

5. ***DinCoada****(TipCoada C, TipElement x)* - suprimă primul element al lui *C* şi îl returnează în *x* (***DeQueue***).

```
/*Exemplu de implementare a TDA Coada cu ajutorul TDA Lista
(setul 1 de operatori)*/

typedef tip_lista tip_coada;              /*[6.5.4.1.b]*/

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

/*initializeaza(C);*/          initializeaza(c);
/*cap(C);*/                    x= furnizeaza(primul(c),c);
/*cdauga(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 şi în cazul stivelor, orice **implementare** a listelor este valabilă şi pentru **cozi**.
- Pornind însă de la observaţia că inserţiile 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 sfârşit atunci când se doreşte adăugarea unui element, se va păstra un **pointer** la **ultimul element al cozii**.
  - De asemenea, ca şi la toate tipurile de liste, se va păstra şi pointerul la **primul element al listei** utilizat în execuţia comenzilor ***cap*** şi ***scoate***.
- În implementare se va utiliza **un nod fictiv** ca şi prim nod al cozii (tehnica **"nodului fictiv"**), caz în care pointerul de început va indica acest nod.
  - Această convenţie 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
structurii de date a unui nod al cozii*/

                                              /*[6.5.4.2.a]*/
typedef struct tip_nod* tip_referinta_nod;

typedef struct tip_nod
 {
   tip_element element;
   tip_referinta_nod urm;
 }  tip_nod;
-------------------------------------------------------------
```

- În aceste condiţii se poate defini o **structură de date coadă** care constă din **doi pointeri** indicând **începutul** respectiv **spatele** unei liste înlănţuite.
    - Primul nod al cozii este unul **fictiv** în care câmpul `element` este ignorat.
    - Această convenţie, după cum s-a menţionat mai înainte permite o reprezentare şi o manipulare mai simplă a cozii vide.
- Astfel `TipCoada` se poate defini după cum se prezintă în [6.5.4.2.b].

```
-------------------------------------------------------------
/* Definirea structurii de date tip_coada}*/

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

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

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

void initializeaza(tip_coada* c)            /*[6.5.4.2.c]*/
  /*creaza coada vida c*/
  {
    c->inceput = (tip_nod*)malloc(sizeof(tip_nod)); /*crează
                                                 nodul fictiv*/
    c->inceput->urm= null;
    c->spate= c->inceput; /*nodul fictiv este primul şi
                          ultimul nod al cozii*/
  }  /*initializeaza*/

boolean vid(tip_coada c)
  /*returnează true dacă coada este vidă, false altfel*/
  {
    boolean vid_result;
    if (c->inceput=c->spate)
        vid_result=true;
      else vid_result=false;
```

```
    return vid_result;
  }  /*vid*/

void cap(tip_coada c)
  /*returnează elementul din capul cozii, sau semnalează
    eroare dacă coada este vidă*/
  {
    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)
  /*adaugă un element la spatele cozii*/
  {
    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)
  /*suprimă primul element al cozii, sau semnalează eroare
    dacă coada este vidă*/
  {
    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 şi pentru **cozi**, dar **nu** este foarte eficientă.
    - Într-adevăr, indicatorul la ultimul element al listei, permite implementarea **simplă**, într-un număr fix de paşi, a operaţiei ***adauga***.
    - Execuţia operaţiei ***scoate*** presupune însă **mutarea** întregii cozi cu o poziţie în tablou, operaţie care necesită un timp O*(n)*, dacă coada are lungimea `n`.

- Pentru a depăşi acest dezavantaj se poate utiliza **o structură de tablou circular** în care prima poziţie urmează ultimei, după cum rezultă din figura 6.5.4.3.

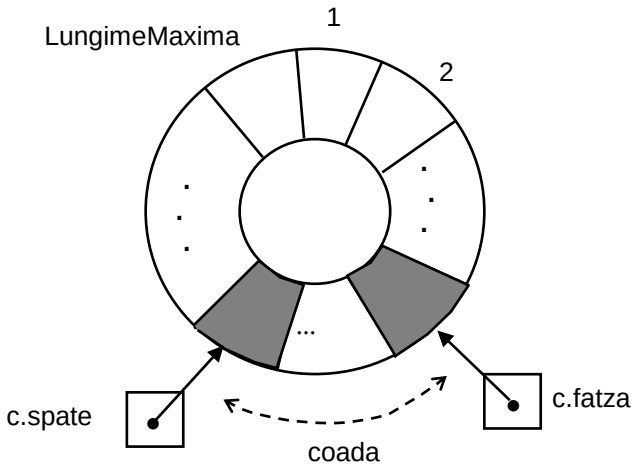

Fig.6.5.4.3. Implementarea unei cozi cu ajutorul unui tablou circular

  - Coada se găseşte undeva în jurul cercului, în **poziţii consecutive**, având **spatele** undeva înaintea **feţei** la parcurgerea cercului în sensul acelor de ceasornic.
  - **Adăugarea** unui element presupune mişcarea indicatorului `c.spate` cu o poziţie în sensul acelor de ceasornic şi introducerea elementului în această poziţie.
  - **Scoaterea** unui element din coadă, presupune simpla mutare a indicatorului `c.fatza` cu o poziţie în sensul rotirii acelor de ceasornic.
    - Astfel coada **se roteşte** în sensul rotirii acelor de ceasornic după cum se **adaugă** sau se **scot** elemente din ea.
- În aceste condiţii atât procedura ***`adauga`*** cât şi ***`scoate`*** pot fi redactate astfel încât să presupună un **număr fix de paşi de execuţie**, 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** şi a **celei pline**.
- Presupunând că structura **coadă** reprezentată în figura 6.5.4.3 este **plină** (conţine `LungimeMaxima` elemente), indicatorul `c.spate` va indica **poziţia 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` şi `c.spate` indică aceeaşi poziţie.
- Dacă se scoate **singurul** element, `c.fatza` avansează cu o poziţie în faţă (în sensul acelor de ceasornic) indicând **coada vidă**.
- Se observă însă că această situaţie este **identică** cea anterioară care indică coada plină, adică `c.fatza` se află cu o poziţie în faţa lui `c.spate`, la parcurgerea în sensul arcelor de ceasornic (trigonometric negativ) a cozii.

- În vederea rezolvării acestei situaţii, în tablou se vor introduce **doar** `Lungime_maxima-1` elemente, deşi în tablou există `Lungime_maxima` poziţii.
  - 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 poziţie.
  - Evident avansările se realizează în sensul acelor de ceasornic al parcurgerii cozii.
- În continuare se prezintă implementarea **operatorilor** definiţi 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*/

typedef struct tip_coada                    /*[6.5.4.3.a]*/
 {
   tip_element elemente[Lungime_maxima];
   int fatza,spate;
 }  tip_coada;
-------------------------------------------------------------
```

- Secvenţa de instrucţii corespunzătoare apare în [6.5.4.3.b].
- Funcţia ***avanseaza***`(c)` furnizează poziţia următoare celei curente în tabloul circular.

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

 int avanseaza(int i)                       /*[6.5.4.3.b]*/
     {
       int avanseaza_result;
       avanseaza_result= (i+1)%Lungime_maxima;
       return avanseaza_result;
     } /*avanseaza*/

   void initializeaza(tip_coada* c)
     {
        c->fatza= 0;
        c->spate= Lungime_maxima-1;
     } /*initializeaza*/

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

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

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

Problemele legate de sesizarea **cozii vide** şi 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 `Lungime_maxima` coadă plină.

- Structura de date corespunzătoare acestei abordări apare în secvenţa [6.5.4.3.c].

- Procedurile care implementează operatorii specifici în aceste circumstanţe pot fi dezvoltate cu uşurinţă în baza exemplelor prezentate anterior.

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

const {Lungime_maxima = 100};            /*[6.5.4.3.c]*/

typedef ... tip_element;

typedef tip_element tip_tablou[Lungime_maxima];

typedef struct tip_coada
 {
   int fatza,spate;
   int contor;
   tip_tablou elemente;
 }  tip_coada;

tip_coada c;
tip_element x;
-------------------------------------------------------------
```

6.5.4.4. Aplicaţie. Conversia infix-postfix

- Aplicaţia de faţă realizează conversia unei **expresii infix** în **reprezentare postfix** utilizând o implementare **nerecursivă**.
    - Ambele expresii sunt reprezentate prin intermediul unor **liste înlănţuite**.
- 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** `CoadaPost` care înregistrează forma postfix a expresiei care se construieşte.
- Fiecărui **operator** din expresia infix i se asociază o **valoare de precedenţă**:
    - Operatorii **`'*'`** şi **`'/'`** au cea mai mare precedenţă.
    - Operatorii **`'+'`** şi **`'-'`** precedenţa următoare.
    - În vederea simplificării proiectării algoritmului, în mod artificial se acordă parantezei stânga **`'('`** prioritatea cea mai scăzută.
- Codul efectiv al procedurii apare în secvenţa [6.5.4.4.a].
    - După cum se observă transferurile se realizează din `StivaOp` în `CoadaPost`, care în final va putea fi utilizată ca şi intrare a procedurii de evaluare.

```
-------------------------------------------------------------
/*Conversie din format Infix în format Postfix - se
utilizează ADT Stiva şi ADT Coadă*/
```

```
tip_stiva stiva_op;                          /*[6.5.4.4.a]*/

static void transfer(tip_stiva* s, tip_coada* q)
 /*Transfera vârful stivei s la sfârsitul cozii q. Se
    presupune ca atomii din s şi q aparţin aceluiaşi tip*/
{
     adauga(varfst(s),q);
     pop(s, 0);
} /*transfer*/

void conversie_in_postfix(tip_lista infix, tip_coada*
coada_post)
/*Converteşte expresia infix din lista infix în forma
postfix memorată în coada_post*/

{
    tip_stiva stiva_op;
    initializeaza(&stiva_op);
    initializeaza(coada_post);
    while(mai exista noduri în lista infix)
      {
       *t este primul nod din lista infix;
/*[1]*/if(t este un operand) /*t este un operand*/
          adauga(t, coada_post);
        else /*t este un operator*/
/*[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*/
               } /*if*/
              else
               {
/*[5]*/           while(precedenţa lui t ≤ precedenţa
                                       lui varfSt(stiva_op))
                      transfer(&stiva_op, coada_post);
                  push(t, &stiva_op);
               } /*else*/
        *şterge nodul t;
      } /*while*/
/*[6]*/
    while(!stivid(stiva_op))
      transfer(&stiva_op,coada_post); /*transferă
                                        operatorii rămaşi*/
} /*conversie_in_postfix*/
-------------------------------------------------------------
```

- Procedura **`ConversieInPostfix`**, ilustrează o mică parte a activităţii pe care **compilatorul** sau **interpretorul** o realizează în momentul translatării expresiilor aritmetice în cod executabil.

    - De fapt sunt eludate fazele de analiză sintactică şi semantică ale expresiei precum şi natura efectivă a operanzilor (constante, variabile etc.).

- **Varianta recursivă** a aceleaşi aplicaţii a fost prezentată la exemple de algorimi recursivi.

6.5.5. Cozi bazate pe prioritate

- **Coada bazată pe prioritate** (**"priority queue"**) este structura de date abstractă care permite **inserţia** unui element şi **suprimarea celui mai prioritar** element.

- O astfel de structură diferă atât faţă de structura **coadă** (din care se suprimă **primul** venit, deci **cel mai vechi**) cât şi faţă de **stivă** (din care se suprimă **ultimul** venit, deci **cel mai nou**).

- De fapt **cozile bazate pe prioritate** pot fi concepute ca şi structuri care **generalizează cozile** şi **stivele**.

    - **Cozile** şi **stivele** pot fi implementate prin cozi bazate pe prioritate atribuind **priorităţi corespunzătoare**.

- **Aplicaţiile** 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 **priorităţi**, definirea acesteia apare în [6.5.5.1.a].

TDA Coadă bazată pe prioritate

Modelul matematic: o secvenţă finită de noduri. Toate nodurile aparţin unui aceluiaşi 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 inserţia normală şi suprimarea doar a celui mai prioritar nod.

Notaţii:

```
TipCoadaBazataPePrioritate q;          [6.5.5.1.a]
```

```
TipElement x;
boolean b;
```

Operatori:

1. ***Initializează****(TipCoadaBazataPePrioritate q)* - face coada *q* vidă.

2. ***Inserează****(TipElement x, CoadaBazataPePrioritate q)* - inserţia unui nou element *x* în coada *q*.

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

4. *TipElement* ***Inlocuieşte****(CoadaBazataPePrioritate q, TipElement x)* - înlocuieşte cel mai prioritar element al cozii *q* cu elementul *x*, mai puţin în situaţia în care noul element este cel mai prioritar element. Returnează cel mai prioritar element.

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

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

7. *boolean* ***Vid****(TipCoadaBazataPePrioritate q)* - operator care returnează **true** dacă şi numai dacă *q* este o coadă vidă.

- Operatorul ***Inlocuieşte*** constă dintr-o **inserţie** urmată de **suprimarea** celui mai prioritar element.
    - Este o operaţie diferită de succesiunea suprimare-inserţie deoarece necesită creşterea pentru moment a dimensiunii cozii cu un element.
    - Acest operator se defineşte separat deoarece în anumite implementări poate fi conceput foarte eficient.
- În mod analog operatorul ***Schimbă*** poate fi implementat ca şi o **suprimare**, urmată de o **inserţie**, iar generarea unei cozi ca şi 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 performanţe diferite pentru operatorii specifici.
- În continuare vor fi prezentate unele dintre aceste posibilităţi.

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 secvenţa [6.5.5.2.a] apare structura de date corespunzătoare acestei abordări iar în secvenţa [6.5.5.2.b] implementarea operatorilor ***Insereaza*** şi ***Extrage***.

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

typedef struct tip_element                    /*[6.5.5.2.a]*/
 {
   tip_prioritate prioritate;
   tip_info info;
 }  tip_element;

typedef struct tip_coada_bazata_pe_prioritate
 {
   tip_element elemente[dim_max];
   int nr_elemente;
 } tip_coada_bazata_pe_prioritate;
-------------------------------------------------------------
/*Implementarea cozilor bazate pe prioritate cu tablouri -
operatorii insereaza şi extrage*/
                                              /*[6.5.5.2.b]*/
void insereaza(tip_element x, tip_coada_bazata_pe_prioritate
q)
  {      /*performanţa O(1)*/
    q.nr_elemente= q.nr_elemente+1;
    q.elemente[q.nr_elemente]= x;
  } /*insereaza*/

tip_element extrage(tip_coada_bazata_pe_prioritate q)
  {
    int j,max;  /*performanţa O(n)*/
    tip_element extrage_result;

    max= 0;
    for(j=0;j<q.nr_elemente;j++)
      if(q.elemente[j-1].prioritate >
              q.elemente[max-1].prioritate) max= j;
    extrage_result= q.elemente[max];
    q.elemente[max]= q.elemente[q.nr_elemente-1];
    q.nr_elemente= q.nr_elemente-1;
    return extrage_result;
  } /*extrage*/
-------------------------------------------------------------
```

- Pentru **inserţie** se incrementează `nr_elemente` şi se adaugă elementul de inserat pe ultima poziţie a tabloului `q.elemente`, operaţie care este constantă în timp (O*(1)*).

- Pentru **extragere** se baleează întreg tabloul găsindu-se cel mai mare element, apoi se introduce ultimul element pe poziţia celui care a fost extras şi se decrementează `nr_elemente`. Operatorul necesită o regie O*(n)*.

- Implementarea lui ***inlocuieste*** este similară, ea presupunând o inserţie urmată de o extragere.

- Redactarea celorlalţi operatori în acest context nu ridică nici un fel de probleme.

- În implementarea **cozilor bazate pe prioritate** se pot utiliza şi **tablouri ordonate**.

  - Elementele cozii sunt păstrate în tablou în **ordinea crescătoare a priorităţilor**.

  - In aceste condiţii, operatorul ***extrage***, returnează **ultimul element** `q.elemente[nr_elemente-1]` şi decrementează `nr_elemente`, operaţie ce durează un interval constant de timp.

  - Operatorul ***insereaza*** presupune mutarea spre dreapta a elementelor mai mari ca şi elementul de inserat, operaţie care durează O*(n)*.

- Operatorii care se referă la **cozi bazate pe prioritate** pot fi utilizaţi în implementarea unor **algoritmi de sortare**. Spre exemplu, dacă:

  - (1) Se aplică în mod **repetat** operatorul ***insereaza*** pentru a introduce elemente într-o **coadă bazată pe priorităţi.**

  - (2) Se **extrag** pe rând elementele cu operatorul ***extrage*** până la golirea cozii.

  - (3) Se obţine **secvenţa sortată** a elementelor în **ordinea descrescătoare** a priorităţii lor.

- Dacă se utilizează o **coadă bazată pe prioritate** reprezentată ca un **tablou neordonat** se obţine **sortarea prin selecţie**.

- Dacă se utilizează o **coadă bazată pe prioritate** reprezentată ca un **tablou ordonat**, se obţine **sortarea prin inserţie**.

6.5.5.3. Implementarea cozilor bazate pe prioritate cu ajutorul listelor înlănţuite

- În implementarea cozilor bazate pe prioritate pot fi utilizate şi **listele înlănţuite** în variantă **ordonată** sau **neordonată**.

- Această abordare **nu** modifică principial implementarea operatorilor specifici.

  - Ea face însă posibilă realizarea mai **eficientă** a **fazelor de inserţie şi suprimare** a elementelor cozii datorită flexibilităţii listelor înlănţuite.

- În continuare se prezintă un **exemplu** de realizare a operaţiei ***extrage***, utilizând pentru implementarea cozilor bazate pe prioritate **listele înlănţuite neordonate**.

- Se utilizează următoarele structuri de date [6.5.5.3.a].

```
-------------------------------------------------------------
/*Implementarea cozilor bazate pe prioritate cu liste
înlănţuite neordonate - structuri de date*/

typedef int tip_element;                    /*[6.5.5.3.a]*/

typedef struct tip_nod* tip_referinta_nod;

typedef struct tip_nod {
  tip_element element;
  tip_referinta_nod urm;
} tip_nod;

typedef tip_referinta_nod coada_bazata_pe_prioritate;
-------------------------------------------------------------
```

- Precizări:
  - În implementare se utilizează o variantă a **tehnicii celor doi pointeri** utilizând un singur pointer (`curent`) într-o exploatare de tip **"look ahead"**.
  - Primul nod al listei este un nod **fictiv**, el nu conţine nici un element având poziţionat numai câmpul `urm` (tehnica **"nodului fictiv"**) [6.5.5.3.b].

```
-------------------------------------------------------------
/*Implementarea cozilor bazate pe prioritate cu liste
înlănţuite neordonate - operatorul extrage*/

#include <stdlib.h>                         /*[6.5.5.3.b]*/

tip_referinta_nod extrage(coada_bazata_pe_prioritate* q)
  {
    tip_referinta_nod curent;/*nodul din faţa 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");
      else
       {
         mare= (*q)->urm->element; /*q indică 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 reţine pointerul
           nodului ce precede nodul cu cheia cea mai mare */
               mare= curent->urm->element;
             } /*if*/
            curent= curent->urm;
```

```
          } /*while*/
        extrage_result= anterior->urm; /*poziţionează
                        pointerul pe cel mai mare element*/
        curent= anterior->urm;
        anterior->urm= anterior->urm->urm;
      } /*else*/
    return extrage_result;
  } /*extrage*/
```

- Implementarea celorlalte funcţii 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 situaţii sunt mai utile decât cele bazate pe modele sofisticate.

    - Astfel, implementarea bazată pe **liste neordonate** e potrivită în situaţiile în care se fac multe inserţii şi mai puţine extrageri.
    - În schimb, implementarea bazată pe **liste ordonate** este potrivită dacă priorităţile elementelor care se inserează au tendinţa 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 parţial ordonat** reprezentat printr-un **tablou liniar**, ale cărui elemente satisfac condiţiile ansamblului (&3.2.5).

    - În consecinţă, cea mai mare (prioritară) cheie este poziţionată întotdeauna pe **prima poziţie** a tabloului care materializează **ansamblul**.

- Algoritmii care operează asupra structurilor de date **ansamblu** necesită în cel mai defavorabil caz O $(\log_2 n)$ paşi.

- **Ansamblele** pot fi utilizate în implementarea **cozilor bazate pe prioritate**.

- Există algoritmi care prelucrează ansamblul de **sus în jos**, alţii 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 (`Dim_max`) .
    - (3) **Dimensiunea curentă** a tabloului este păstrată în variabila `nr_elemente` care face parte din definiţia **cozii bazate pe prioritate** [6.5.5.4.a].

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

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

typedef struct tip_coada_bazata_pe_prioritate
 {
   tip_element ansamblu[Dim_max];
   int nr_elemente;
 }  tip_coada_bazata_pe_prioritate;
```

- Pentru **construcţia** 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 şi 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 ***down_heap***.
    - O altă posibilitate de realizare a **inserţiei** o reprezintă **extinderea** spre **dreapta** a ansamblului, prin introducerea elementului de inserat pe **ultima** sa poziţie.
    - Această operaţie poate viola însă **regulile ansamblului** dacă **noul element** introdus are prioritatea mai mare ca şi părintele său.
        - În această situaţie 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 şi **prioritatea părintelui său**, sau a ajuns pe **prima** poziţie a ansamblului.
- Procedura ***up_heap*** prezentată în secvenţa [6.5.5.4.b] implementează aceast operator respectiv avansul elementului nou introdus de pe **ultima poziţie 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 prelungeşte spre stânga cu poziţia `0` care **nu aparţine ansamblului** dar care este utilizată pe post de **fanion** în procesul de ascensiune.
- Alături de procedura ***up_heap*** apare şi procedura care implementează inserţia propriu-zisă (***insertie***).

```
/*Implementarea cozilor bazate pe prioritate cu ajutorul
ansamblelor - operatorii up_heap şi insereaza*/

                                              /*[6.5.5.4.b]*/
void up_heap(tip_coada_bazata_pe_prioritate q, int k)
 {
   /*elementul de inserat se găseşte în ansamblu pe
```

```
      poziţia k*/
   tip_element v;
   v= q.ansamblu[k];  /*v memorează elementul de inserat*/
   /*poziţia 0 se utilizează pe post de fanion*/
   q.ansamblu[0].prioritate=(/*cea mai mare prioritate*/)+1;
   while(q.ansamblu[k/2].prioritate<=v.prioritate)
                              /*testare prioritate părinte*/
      {
        q.ansamblu[k]= q.ansamblu[k/2];
        k= k/2;
      }
   q.ansamblu[k]= v;
 } /*up_heap*/

void insereaza(tip_element x,
               tip_coada_bazata_pe_prioritate q)
{
     tip_element v;
     q.nr_elemente= q.nr_elemente+1;
     q.ansamblu[q.nr_elemente]= x;
     up_heap(q,q.nr_elemente);
} /*insereaza*/
```

- Dacă în operatorul ***up_heap***, `(k/2)` se înlocuieşte cu `(k-1)` se obţine în esenţă **sortarea prin inserţie.**
    - În acest caz, găsirea locului în care se inserează noul element se realizează verificând şi deplasând **secvenţial** elementele cu câte o poziţie spre dreapta.
- În procedura ***up_heap*** deplasarea se face **nu** liniar (secvenţial) ci din **nivel în nivel** de-a lungul ansamblului.
- La fel ca la sortarea prin inserţie, **interschimbarea nu** este totală, `v` fiind implicat mereu în această operaţie.
- Poziţia `0` a ansamblului `q.elemente[0]` se utilizează pe post de **fanion** care se asignează iniţial 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 concordanţă cu definiţia ansamblului.

- Operatorul ***extrage*** presupune:
    - (1) Extragerea elementului cel mai prioritar (situat pe poziţia `q.elemente[1]`).
    - (2) Introducerea lui `q.elemente[q.nr_elemente]` (ultimul element al ansmblului) pe prima poziţie.
    - (3) Decrementarea numărului de elemente (`q.nr_elemente`).
    - (4) Deplasarea primului element de **sus în jos**, spre baza ansamblului, până la locul potrivit.

- Deplasarea de sus în jos în ansamblu de la poziţia `k` spre baza acestuia este materializată de operatorul ***down_heap*** [6.5.5.4.c].

```
-------------------------------------------------------------
/*Implementarea cozilor bazate pe prioritate cu ajutorul
ansamblelor - operatorul down_heap*/
                                              /*[6.5.5.4.c]*/
void down_heap(tip_coada_bazata_pe_prioritate q, int k)
  {
    tip_indice1 j;
    tip_element v;
    boolean ret;
    v= q.ansamblu[k];
    ret= false;
    while((k<q.nr_elemente/2)&&(!ret))
        {
          j= k+k; /*avans pe nivelul următor*/
          if(j<q.nr_elemente) /*selecţie cel mai prioritat
                                fiu*/
            if(q.ansamblu[j].prioritate<
               q.ansamblu[j+1].prioritate)  j= j+1;
          if(v.prioritate>=q.ansamblu[j].prioritate)
              ret= true;
            else
              {
                q.ansamblu[k]= q.ansamblu[j];
                k= j;
              } /*else*/
        } /*while*/
    q.ansamblu[k]= v; /*inserţia noului element*/
  } /*down_heap*/
-------------------------------------------------------------
```

  - Deplasarea în ansamblu se realizează interschimbând elementul de pe poziţia curentă `k` cu cel mai prioritar dintre fiii săi şi 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 şi în situaţia anterioară **nu** este necesară interschimbarea completă întrucât `v` este implicat tot timpul.
  - Bucla **`while`** este prevăzută cu două ieşiri:
    - Una corelată cu atingerea **bazei ansamblului** `(k>q.nr_elemente/2)`.
    - A doua corelată cu **găsirea poziţiei** elementului de inserat în interiorul ansamblului (variabila booleană `ret`).

- Cu aceste precizări, implementarea operatorilor ***extrage*** şi ***inlocuieste*** este imediată [6.5.5.4.d].

```
-------------------------------------------------------------
/*Implementarea cozilor bazate pe prioritate cu ajutorul
ansamblelor - operatorii Extrage şi Inlocuieste*/
```

```
                                                    /*[6.5.5.4.d]*/
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;
    down_heap(q,1);
    return extrage_result;
  } /*extrage*/

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

- În cazul operatorului ***inlocuieste*** se utilizează şi poziţia `q.elemente[0]` ai cărei fii sunt poziţiile `0` (ea însăşi) şi `1`.
    - Astfel dacă `x` este mai prioritar decât oricare element al ansamblului, ansamblul rămâne nemodificat, altfel `x` este deplasat în ansamblu.
    - În toate situaţiile este returnat `q.elemente[0]` în calitate de cel mai prioritar element.
- Operatorii ***suprima*** şi ***schimba*** pot fi implementaţi utilizând combinaţii simple ale metodelor de mai sus.
    - Spre exemplu dacă prioritatea elementului situat pe poziţia `k` este ridicată atunci poate fi apelată procedura ***up_heap***`(q,k)`, iar dacă prioritatea sa este coborâtă, procedura ***down_heap***`(q,k)` rezolvă situaţia.

6.5.5.5. Tipul de date abstract ansamblu

- Se reiterează observaţia 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 consecinţă, ne propunem să definim un **TDA Ansamblu**.
- **TDA Ansamblu** constă din **modelul matematic** descris de un **arbore binar parţial ordonat** peste care se definesc **operatorii** de construcţie (extensie) ai ansamblului ***up_heap***, şi ***down_heap***, operatorul de extragere ***extrage,*** operatorul ***dimensiune_ansamblu*** precum şi un operator ***initializează_ansamblu*** care creează ansamblul vid.

- În sinteză o posibilă variantă de **TDA Ansamblu** apare în secvenţa [6.5.5.5.a] .

TDA Ansamblu

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

Notaţii:

```
tip_ansamblu a;                                  [6.5.5.5.a]
tip_element x;
int n;
tip_poziţie p;
```

Operatori:

1. ***initializează_ansamblu****(tip_ansamblu a)* - operator care iniţializeză pe *a* ca ansamblu vid, şi face dimensiunea ansamblului egală cu 0.

2. ***up_heap****(tip_ansamblu a, tip_element x, tip_poziţie p)* - extensia spre dreapta a ansamblului *a*. Operator care introduce elementul *x* pe poziţia *p* (ultima sa poziţie) şi îl deplasează în ansamblu de jos în sus până la poziţia specifică priorităţii, sau în vârful acestuia dacă prioritatea lui *x* este maximă. Incrementeză dimensiunea ansamblului.

3. ***down_heap****(tip_ansamblu a, tip_element x, tip_poziţie p)* – extensia spre stânga a ansamblului *a*. Operator care introduce elementul *x* în poziţia *p* a ansamblului (pe prima sa poziţie) şi îl deplasează în ansamblu de sus în jos până la poziţia specifică priorităţii, sau până la baza acestuia dacă prioritatea lui *x* este minimă. Incrementeză dimensiunea ansamblului.

4. *tip_element* ***extrage****(tip_ansamblu a)* – operator care extrage şi furnizează 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 extracţiei.

5. *int* ***dimensiune_ansamblu****(tip_ansamblu a)* – operator care furnizează dimensiunea curentă a ansamblului *a*.

- După cum s-a precizat, operatorii care acţionează asupra **TDA Ansamblu** se încadrează cu toţii în performanţa $O(log_2 n)$ unde n este dimensiunea ansamblului.

- Excepţie fac operatorii ***iniţializează_ansamblu*** şi ***dimensiune_ansamblu*** care sunt O*(1)*.

- Un exemplu de definire şi implementare a unui **ansamblu** conceput ca şi un arbore **binar parţial ordonat**, reprezentat printr-o **structură tablou** s-a realizat în capitolul 3, subcapitolul (§ 3.2.5).

- Modalităţi de implementare a unora dintre operatorii specifici au fost prezentate atât în subcapitolul mai sus menţionat (§3.2.5) cât şi pe parcursul acestui subcapitol (vezi secvenţele [6.5.5.4.b] şi [6.5.5.4.c]).

- Funcţie de aplicaţie pot fi definiţi şi alţi operatori pentru **TDA Ansamblu**, cum ar fi spre exemplu operatorii referitori la verificarea sau schimbarea **priorităţii elementelor**. Un astfel de exemplu va fi prezentat în volmul II al acestui manual în partea referitoare la grafuri.

6.6. Structura de date multilistă

- Se numeşte **multilistă**, o structură de date ale cărei noduri conţin **mai multe câmpuri de înlănţuire.**

    - Cu alte cuvinte, un nod al unei astfel de structuri poate aparţine în acelaşi timp la **mai multe liste înlănţuite simple**.

- În literatura de specialitate termenii consacraţi 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 secvenţa [6.6.a] se prezintă un **exemplu de definire** a unei astfel de structuri.

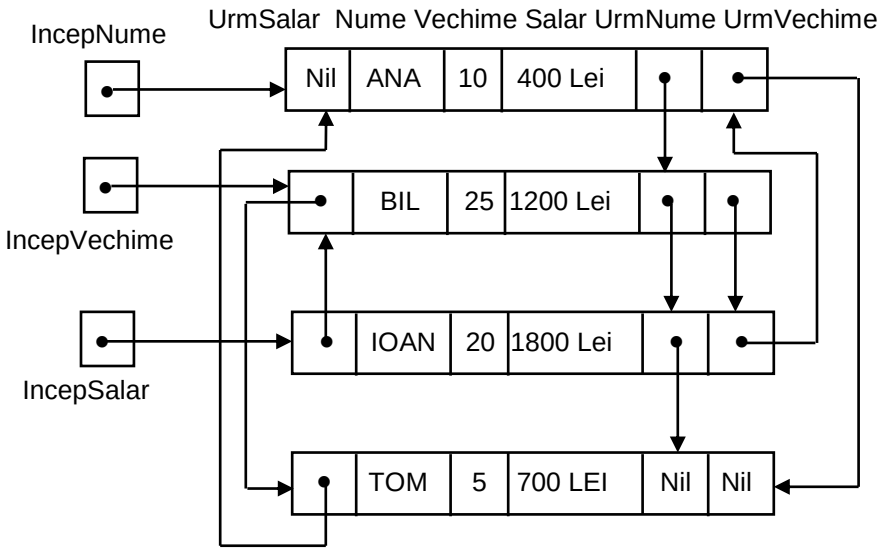

Fig.6.6.a. Exemplu de structură de date multilistă

```
-------------------------------------------------------------
/*Structura multilistă - definirea structurilor de date*/

typedef char* tip_nume;                          /*[6.6.a]*/

typedef struct tip_info1
 {
    tip_nume nume;
    int vechime;
    float salar;
 } tip_info1;

typedef struct tip_nod* tip_referinta_nod;

typedef struct tip_nod
 {
    tip_info1 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.
    - Prezenţa mai **multor înlănţuiri** într-un acelaşi nod, respectiv **apartenenţa simultană** a aceluiaşi nod la mai multe liste, asigură acestei structuri de date o **flexibilitate** deosebită.
- Acest avantaj, coroborat cu o manipulare relativ **facilă** specifică structurilor înlănţuite este exploatat la implementarea **bazelor de date**, cu precădere a celor relaţionale.
- 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 conţin de regulă un **număr redus** de elemente restul fiind poziţionate de obicei pe zero.
        - Din acest motiv memorarea lor în forma obişnuită a tablourilor bidimensionale presupune o **risipă** mare de memorie.
    - Spre a evita această risipă se poate utiliza **structura multilistă** din secvenţa [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.**
- **Înlănţuirile la proximul element valid** din acelaşi rând respectiv la proximul element valid situat pe aceeaşi coloană.

```
-------------------------------------------------------------
/*Aplicaţie la structura multilistă - memorarea matricilor
rare utilizând structurile multilistă*/


typedef struct tip_nod* tip_referinta_nod;    /*[6.6.b]*/

typedef struct tip_nod
 {
   int rand;
   int coloana;
   tip_info info;
   tip_referinta_nod dreapta,jos;
 } tip_nod;

tip_referinta_nod matrice;
-------------------------------------------------------------
```

6.7. Liste generalizate

- Singura structură de date definită în limbajul de programare **LISP** este **lista**.
  - După cum se cunoaşte limbajul LISP este destinat **manipulării simbolurilor** fiind utilizat cu precădere în domeniul **Inteligenţei Artificiale**.
- Unitatea structurală fundamentală definită în **LISP** este **atomul** care este conceput ca un şir de caractere şi / sau cifre.
- În acest context, o **listă** este o mulţime de paranteze conţinând orice număr de **atomi** şi **liste**. Spre exemplu:
  - `B=(b)` reprezintă în LISP o listă care conţine un singur element.
  - `L =(a,b,(c,d,e),f)` reprezintă o listă cu o structură mai complexă, care conţine o sublistă.
- Atomii şi listele sunt memorate în LISP cu ajutorul unei structuri de date speciale numită **listă generalizată**.
- Nodurile unei **liste generalizate** conţin trei câmpuri:
  - (1) `Atom` - câmp boolean.
  - (2) Un câmp dependent de valoarea câmpului `Atom` .
    - Dacă `Atom` este **`TRUE`** atunci acest câmp se numeste `info` şi memorează o informaţie adică un **atom**.
    - Dacă `Atom` este **`FALSE`** câmpul se numeste `lista` şi memorează o **referinţă la o listă**;

- (3) `Urm` – câmp de înlănţuire.

- În figura 6.7.a apar reprezentările grafice ale atomului `'a'` şi ale listelor `B` şi `L` definite anterior, în accepţiunea acestei definiţii.

- Pentru o mai bună înţelegere a acestui concept, în secvenţa [6.7.a] apare specificarea PASCAL respectiv C a structurii de date `ListăGeneralizată`.

  - Din cauza restricţiei de implementare a articolului cu variante în aceste limbaje, implementare care impune plasarea variantelor la sfârşitul articolului, câmpul de înlănţuire `Urm` apare în declaraţia nodului la începutul acestuia.

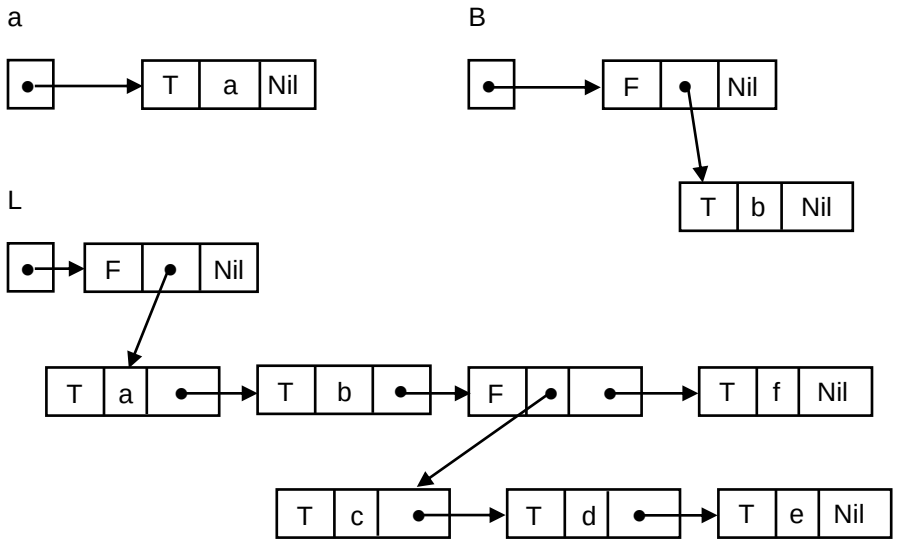

Fig.6.7.a. Exemple de liste generalizate

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

TYPE TipListaGeneralizata = ^TipNod;            {[6.7.a]}

     TipNod = RECORD
       Urm: TipListaGeneralizat;
       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;                       /*[6.7.a]*/
#define true (1)
#define false (0)
```

```
typedef struct tip_nod* tip_lista_generalizata;

typedef struct tip_nod
 {
   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 uzanţelor acestui manual precizarea modelului matematic, a notaţiilor şi a operatorilor specifici.

- Acest lucru este realizat în [6.7.1.a].

TDA Listă Generalizată

Modelul matematic: o secvenţă finită de noduri. Toate nodurile aparţin unui aceluiaşi tip numit tip de bază. Fiecare nod conţine trei câmpuri:(1) un câmp boolean – Atom;(2) un câmp dependent de valoarea câmpului Atom: dacă Atom este TRUE atunci acest câmp se numeste info şi memorează o informaţie adică un atom, dacă Atom este FALSE câmpul se numeste lista şi memorează o referinţă la o listă;(3) un câmp de înlănţuire – Urm.

Notaţii:

lista: TipListaGeneralizata; [6.7.1.a]

Operatori:

1. *TipListaGeneralizata* ***car*** *(TipListaGeneralizata lista)* – operator care returnează **primul element al listei** (care poate fi atom sau listă).

2. *TipListaGeneralizata* ***cdr*** *(TipListaGeneralizata lista)* – operator care returnează **restul listei**, adică ceea ce rămâne după înlăturarea primului element.

3. *TipListaGeneralizata* ***cons*** *(TipListaGeneralizata M,N)* – operator care realizează construcţia unei liste în care M este primul element iar N restul listei.

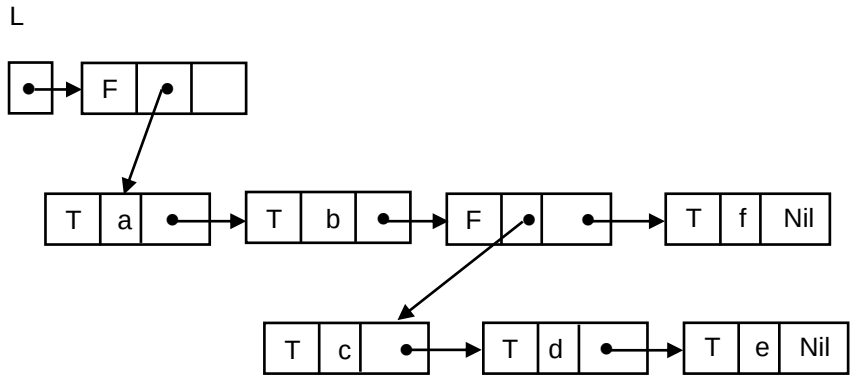

Fig.6.7.1.a. Exemplu de listă generalizată

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

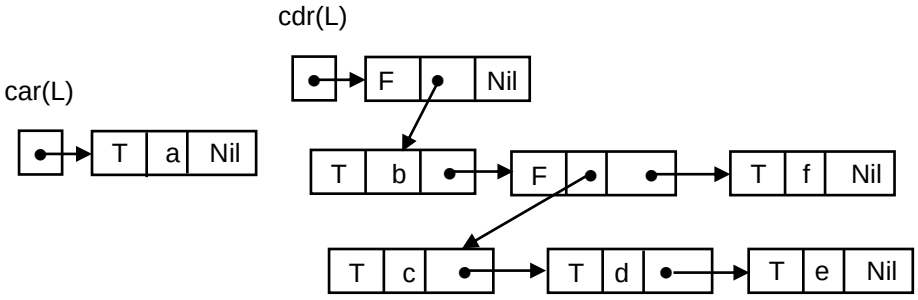

Fig.6.7.1.b. Operatorii `car` şi `cdr` aplicaţi asupra listei generalizate `L` din fig.6.7.1.a

- În mod asemănător, pentru listele generalizate `M=(g,h)` şi `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. 6.7.1.c.

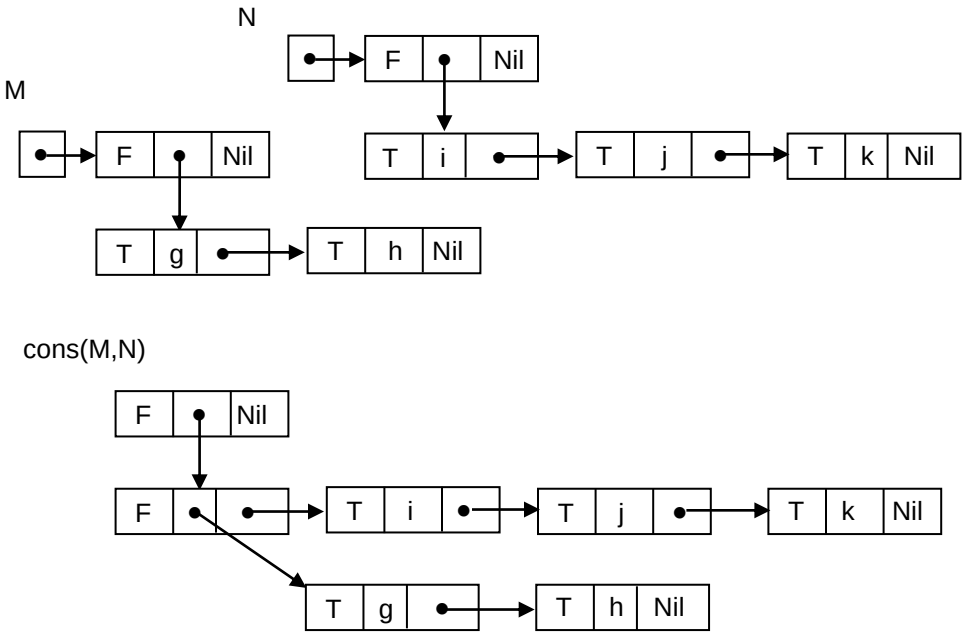

Fig.6.7.1.c. Exemplu de utilizare a operatorului `cons`

6.8. Tipul de date abstract asociere

- **Mapping**-ul sau **asocierea memoriei** este o **funcţie** definită pe mulţimea elementelor unui tip denumit **tip domeniu** cu valori în mulţimea elementelor unui alt tip (eventual acelaşi) numit **tip valoare** (codomeniu).
    - De fapt o **asociere** `M`, pune în **corespondenţă** un element `V` aparţinând **tipului valoare** cu un element `D` al **tipului domeniu** prin **relaţia** `M(D)=V`.
- Anumite asocieri, ca spre exemplu `PATRAT(i)=i`^{2}, pot fi implementate simplu cu ajutorul unor **funcţii** care precizează **expresia aritmetică de calcul**, sau prin alte mijloace simple, de determinare a lui `M(D)` pornind de la `D`.
- În multe situaţii 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` aparţinând lui `TipDomeniu` se pot defini următorii operatori:
    - (1) Un operator ***`Initializează`*** care realizează iniţializarea domeniului valorilor cu asocierea vidă, adică cea a cărui 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 şi furnizează valoarea lui `M(D)`.
        - (c) În caz negativ **returnează** valoarea fals.
- Aceşti operatori sunt precizaţi sintetic în secvenţa [6.8.1.a] care descrie **TDA Asociere**.

`TDA Asociere`

Modelul matematic: o funcţie (algebrică) definită pe mulţimea elementelor unui tip denumit *TipDomeniu* şi cu valori în mulţimea elementelor unui alt tip denumit *TipCodomeniu*.

Notaţii:

```
TipAsociere M;                                    [6.8.1.a]
TipDomeniu D;
TipCodomeniu V;
Boolean b;
```

Operatori:

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

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

3. *Boolean* ***Determina****(TipAsociere M, TipDomeniu D, TipCodomeniu V)* - operator care returnează valoarea adevărat şi 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 situaţii **tipul domeniu** al funcţiei de asociere este un **tip elementar** care poate fi utilizat ca şi 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 secvenţa [6.8.2.a].

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

enum {prima_valoare = 30,                    /*[6.8.2.a]*/
      ultima_valoare = 100};

typedef int tip_domeniu;

typedef int tip_codomeniu;

typedef tip_codomeniu tip_asociere[ultima_valoare-
                                   prima_valoare+1];
```

- Presupunând în continuare că *'nedefinit'* este o constantă a lui `tip_codomeniu`, operatorii aferenţi **TDA Asociere** apar definiţi în secvenţa [6.8.2.b].

```
-------------------------------------------------------------
/*TDA Asociere - implementare cu ajutorul tablourilor -
operatorii Initializeaza, Atribuie şi Determina*/

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

boolean determina(tip_asociere m, tip_domeniu d,
                  tip_codomeniu* v)
  {
    boolean determina_result;
    if(m[d-prima_valoare]= nedefinit)
        determina_result= false;
      else
        {
          *v= m[d-prima_valoare];
          determina_result= true;
        } /*else*/
    return determina_result;
  } /*determina*/
-------------------------------------------------------------
```

6.8.3. Implementarea TDA Asociere cu ajutorul listelor înlănţuite

- Există multe posibilităţi de a implementa funcţiile de asociere cu domenii finite.
    - Astfel tabelele `HASH` (§7.) reprezintă în anumite situaţii o soluţie excelentă. Ele vor fi studiate în capitolul următor.
- În paragraful de faţă se va descrie o altă posibilitate.
    - Orice structură **asociere**, poate fi reprezentată ca o listă de perechi (D_1,V_1), $(D_2,V_2),\ldots,(D_k,V_k)$, unde
        - $D_1,D_2,\ldots,D_k$ sunt elementele ale **domeniului**.
        - V_i sunt valori ale **codomeniului** pe care asocierea le pune în corespondenţă cu D_i pentru $i=1,2\ldots,k$.
    - 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 şi o **listă înlănţuită**, implementată în oricare din manierele cunoscute, în care `TipElement` are următoarea structură [6.8.3.a].

```
/*TDA Asociere - implementare cu ajutorul listelor înlănţuite - structuri de date*/

typedef struct tip_element                    /*[6.8.3.a]*/
 {
    tip_domeniu argument;
    tip_codomeniu valoare;
 } tip_element;
```

- Cei trei operatori care gestionează **structura asociere**, descrişi în termenii operatorilor definiţi asupra **TDA Lista** apar în secvenţa [6.8.3.b].

```
/*TDA Asociere - implementare bazată pe TDA Listă (varianta restrânsă) - operatorii initializeaza, atribuie şi determina*/

void initializeaza(tip_asociere m)            /*[6.8.3.b]*/
  {
    /*identică cu cea de la liste*/
  } /*initializeaza*/

void atribuie(tip_asociere m,tip_domeniu d,tip_codomeniu 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));  /*inserează (D,V) în listă*/
  } /*atribuie*/

boolean determina(tip_asociere m, tip_domeniu d,
                  tip_codomeniu* 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*/
-----------------------------------------------------------------
```