6. Liste

6.1. Structura de date listă

- În cadrul structurilor avansate de date, **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 şi 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 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**, care se reprezintă de regulă astfel [6.1.a]:

```
------------------------------------------------------------
     a1, a2,...,an                                [6.1.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 **ordonate** liniar funcţie de **poziţia** lor în cadrul listei.
    - 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 $i=2,3,4, ...,n$.
    - De regulă se spune că nodul a_i se află pe poziţia i.
- Este de asemenea convenabil să se postuleze existenţa **poziţiei** următoare **ultimului** element al listei.
    - În această idee se introduce funcţia `FIN(L):TipPozitie` 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 de maniera de implementare a listelor.
- În continuare se prezintă **două** seturi reprezentative de operatori care acţionează asupra listelor, unul restrâns şi altul extins.

6.2.1. Set de operatori restrâns

- Pentru a defini **setul restrâns** de operatori:
    - Se notează cu `L` o listă ale cărei noduri aparţin tipului de bază `TipNod`.
    - `x:TipNod` este un obiect al acestui tip (deci un nod al listei).
    - `p` este o variabilă de `TipPozitie`. Tipul poziţie este dependent de implementare (indice, pointer, cursor, etc.) [AH85].
- În termenii formalismului utilizat în cadrul acestui manual, **TDA Listă** - varianta restrânsă apare în [6.2.1.a].

```
------------------------------------------------------------
TDA Listă
(varianta restrânsă)

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

Notaţii:
      L: TipLista;
      p: TipPozitie;
      x: TipNod.                                  [6.2.1.a]

Operatori:
      1.Fin(L:TipLista):TipPozitie; - operator care
         returnează poziţia următoare ultimului nod al
         listei, adică poziţia următoare sfârşitului ei.
         În cazul listei vide Fin(L)=0;
      2.Insereaza(L:TipLista,x:TipNod,p:TipPozitie); -
         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
         a1,a2,...,an devine a1,a2,...,ap-1,
         x,ap,...an. Dacă p este Fin(L) atunci lista
         devine a1, a2,...an,x. Dacă p>Fin(L) rezultatul
         este nedefinit;
      3.Cauta(x:TipNod,L:TipLista):TipPozitie; - 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.Furnizeaza(p:TipPozite,L:TipLista):TipNod; -
         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****(p:TipPozite,L:TipLista);* - 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. ***Urmator****(p:TipPozite,L:TipLista):TipPozitie;* ***Anterior****(p:TipPozite,L:TipLista):TipPozitie;* - operatori care returnează poziţia următoare, respectiv poziţia anterioară poziţiei *p* în cadrul listei. Dacă *p* este ultima poziţie în *L* atunci ***Urmator****(p,l)=Fin(L)*. *Urmator* nu este definit pentru *p=Fin(L)*, iar *Anterior* pentru *p=1*.

7. ***Initializeaza****(L:TipLista):TipPozitie;*- operator care face lista *L* vidă şi returnează poziţia *Fin(L)=0*.

8. ***Primul****(L:TipLista):TipPozitie;* - returnează valoarea primei poziţii în lista *L*. Dacă *L* este vidă, poziţia returnată este *Fin(L)=0*.

9. ***TraverseazaLista****(L:TipLista,ProcesareNod(...):PROCEDURE);* - 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].

```
/* Eliminarea nodurilor duplicat din cadrul unei liste */

typedef struct{
  int nrcurent;
  char* nume;
  char* adresa;
}tipnod;                                      /*[6.2.1.b]*/
void elimina(tiplista *l)
   /*procedura suprima duplicatele nodurilor din lista*/
{
  tippozitie p,q;  /*p este pozitia curenta*/
                     /*q este utilizat în cautare*/
  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);
    }
}
```

- 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. 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].
- În principiu structura listei avute în vedere se bazează pe înlănţuiri [SH90].

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

 Operatori:
      1.CreazaListaVida(incLista: TipLista); - variabila
         incLista devine nil.
      2.ListaVida(incLista:TipLista):boolean; - operator
         care returnează TRUE dacă lista este vidă
         respectiv FALSE altfel.
      3.Primul(incLista:TipLista,curent:TipIndicatorNod; -
```

```
   operator care face ca variabila curent să indice
   primul nod al listei precizată de incLista.
4.Ultimul(curent: TipIndicatorNod): boolean; -
   operator care returnează TRUE dacă curent
   indică ultimul element al listei.
5.InserInceput(incLista: TipLista,pnod:
   TipIndicatorNod); - inserează la inceputul
   listei incLista nodul indicat de pnod.
6.InserDupa(curent,pnod: TipIndicatorNod); -
   inserează nodul indicat de pnod după nodul
   indicat de curent. Se presupune că curent
   indică un nod din listă.
7.InserInFatza(curent,pnod: TipIndicatorNod); -
   insertie în faţa nodului curent.
8.SuprimaPrimul(incLista: TipLista); - suprimă
   primul nod al listei incLista.
9.SuprimaUrm(curent: TipIndicatorNod); - suprimă
   nodul următor celui indicat de curent.
10.SuprimaCurent(curent: TipIndicatorNod);
11.Urmatorul(curent: TipIndicatorNod); - curent se
   poziţionează pe următorul nod al listei. Dacă
   curent indică ultimul nod al listei el va
   deveni nil.
12.Anterior(curent: TipIndicatorNod); - curent se
   poziţionează pe nodul anterior celui curent.
13.MemoreazaInfo(curent: TipIndicatorNod,info:
   TipInfo); - atribuie nodului indicat de curent
   informaţia info.
14.MemoreazaLeg(curent,p: TipIndicatorNod); -
   atribuie câmpului urm (de legatură) al nodului
   indicat de curent valoarea p.
15.FurnizeazaInfo(curent:TipIndicatorNod):TipInfo;
   - returnează partea de informaţie a nodului
   indicat de curent.
16.FurnizeazaUrm(curent: TipIndicatorNod):
   TipIndicator Nod; - returnează legatura
   nodului curent (valoarea câmpului urm).
17.TraverseazaLista(incLista:TipLista,ProcesareNod
  (...):PROCEDURE); - parcurge nodurile listei L
  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 acestor structuri 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 servi la reprezentarea **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, `TipLista` se defineşte ca un articol cu două câmpuri.
    - (1) Primul câmp este un **tablou** numit `noduri`, cu elemente de `TipNod`,
        - Lungimea acestui tablou este astfel aleasă de către programator încât să fie suficientă pentru a putea păstra cea mai mare dimensiune de listă ce poate apare în respectiva aplicaţie.
    - (2) Cel de-al doilea cîmp este un întreg (`ultim`) care indică în tablou poziţia **ultimului** nod al listei.
    - Cel de-al `i`-lea nod al listei se găseşte în cel de-al `i`-lea element al tabloului, pentru $1 \leq i \leq ultim$ (fig.6.3.1).
    - **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`.

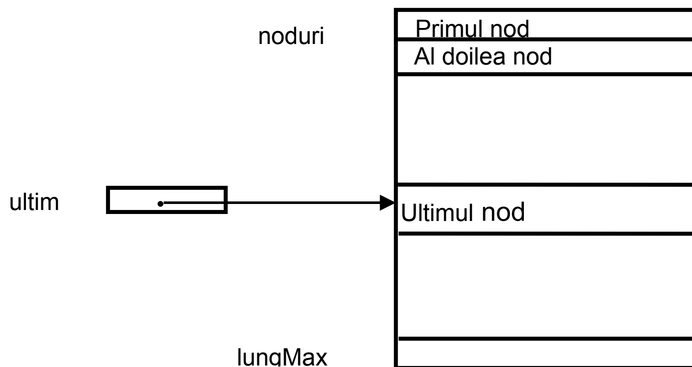

Fig.6.3.1. Implementarea listelor cu ajutorul structurii tablou.

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

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

#define lungmax = 100
typedef struct{                          /*[6.3.1.a]*/
   tipnod noduri[lungmax];
   tipindice ultim;
} tiplista;
typedef tipindice tippozitie;
------------------------------------------------------------
```

- 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 */

#include <stdlib.h>

#define lungmax 100
#define n 30

typedef int tipindice;

typedef struct{
  int nrcurent;
  char* nume;
  char* adresa;
} tipnod;

typedef struct{
   tipnod noduri[lungmax];
   tipindice ultim;
} tiplista;
typedef tipindice tippozitie;

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

boolean reccmp(tipnod, tipnod);
```

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

void insereaza(tiplista* l, tipnod x,
               tippozitie p, boolean* er)
  /*plaseaza pe x in pozitia p a listei; performanta O(n)*/
  {
    tippozitie q;                     /*[6.3.1.b]*/
    *er= false;
    if (l->ultim>=lungmax){
          *er= true;
          printf("lista este plina");
        }
      else
        if ((p>l->ultim+1) || (p<1)){
              *er= true;
              printf("pozitia nu exista");
            }
          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(tippozitie p, tiplista* l, boolean er)
  /*extrage elementul din pozitia p a listei*/
  {
    tippozitie q;                       /*performanta 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*/

tippozitie cauta(tipnod x, tiplista l)
  /*returneaza pozitia lui x in lista*/
  {
    tippozitie q;
    boolean gasit;
    tippozitie cauta_result;
    q= 1; gasit= false;          /*performanta 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(tipnod x, tipnod y)
{
 if ((x.nrcurent == y.nrcurent) && !(strncmp(x.nume,y.nume,n))
       && !(strncmp(x.adresa,y.adresa,n)))
   return true;
 else
   return false;
}
----------------------------------------------------------------
```

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

6.3.2. Implementarea listelor cu ajutorul pointerilor

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

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

typedef int tipinfo;
typedef struct tipnod* tippointernod;          /*[6.3.2.a]*/
typedef struct {
```

```
  int cheie;
  tippointernod urm;
  tipinfo info;
} tipnod;
typedef tippointernod tiplista;
-----------------------------------------------------------------
```

* După cum se observă, în cazul definirii s-au pus în evidenţă trei câmpuri:
    * O **cheie** care serveşte la identificarea nodului,
    * Un **pointer** de înlănţuire la nodul următor
    * 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.

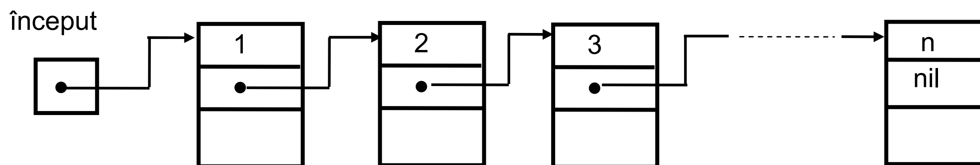

Fig.6.3.2.a. Exemplu de listă liniară

* În secvenţa [6.3.2.a] se observă că o listă liniară poate fi definită ca şi o **structură recursivă** avînd o componentă de tip **identic** cu cel al structurii complete.
* Caracteristica unei astfel de structuri rezidă în prezenţa unei **singure** înlănţuiri.
* În continuare, în cadrul acestui paragraf se prezintă câteva **tehnici** referitoare la implementarea listelor liniare ca şi structuri recursive.
    * (1) Există posibilitatea de a înlocui pointerul `inceput` care indică începutul listei, cu o variabilă de `tipNod` având câmpurile `cheie` şi `info` neasignate, iar câmpul `urm` indicând **primul** nod al listei.
        * Utilizarea acestui nod de început, cunoscută sub denumirea de **tehnica nodului fictiv** simplifică în anumite situaţii prelucrarea listelor înlănţuite (fig.6.3.2.b).

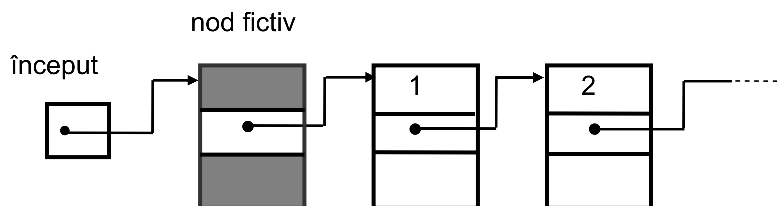

Fig.6.3.2.b. Implementarea listelor prin tehnica nodului fictiv

* (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).

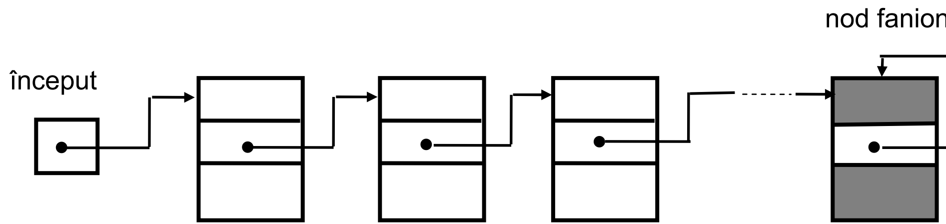

Fig.6.3.2.c. Implementarea listelor prin tehnica nodului fanion

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

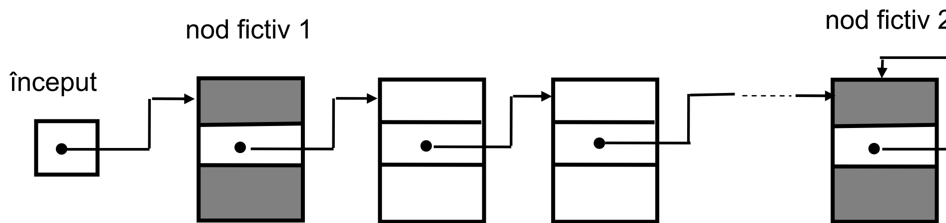

Fig.6.3.2.d. Implementarea listelor cu ajutorul tehnicii celor două noduri fictive

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

6.3.2.1. Tehnici de inserție a nodurilor şi de creeare a listelor înlănțuite

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

```
------------------------------------------------------------------
/*insertie la începutul listei - */
                                                    /*[6.3.2.1.a]*/
  /*[1]*/ auxiliar = (tipnod*)malloc(sizeof(tipnod));
  /*[2]*/ auxiliar->urm= inceput;      /*performanta O(1)*/
  /*[3]*/ inceput= auxiliar;
  /*[4]*/ inceput->info= 0;
------------------------------------------------------------------
```

* În figura 6.3.2.1.a se prezintă grafic maniera în care se desfăşoară o astfel de inserție.

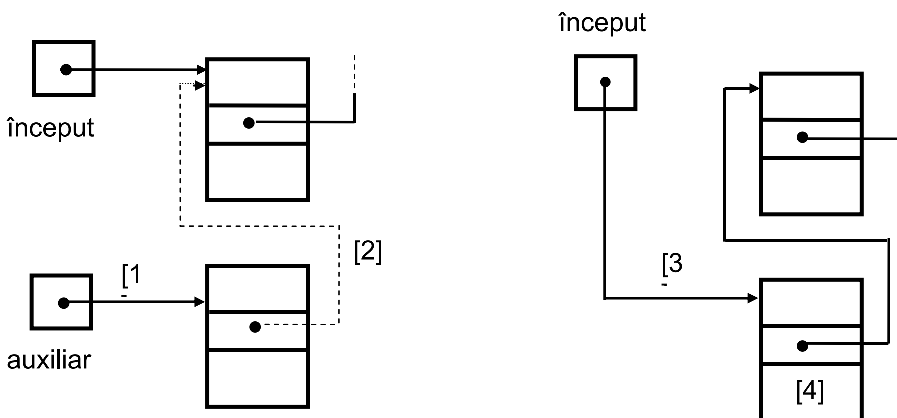

Fig.6.3.2.1.a. Inserția unui nod la începutul unei liste înlănțuite

- Pe baza acestui fragment de program se prezintă în continuare, **crearea** unei liste înlănțuite.
    - Se pornește cu o listă vidă în care se inserează câte un nod la începutul listei până când numărul nodurilor devine egal cu un număr dat `n`.
    - În secvență s-a omis asignarea câmpurilor de informație [6.3.2.1.b]:

```
------------------------------------------------------------
/*creearea unei liste înlantuite*/
                                                 /*[6.3.2.1.b]*/
   inceput= NULL;  /*se porneste cu lista vida*/
   while (n>0){
      auxiliar = (tipnod*)malloc(sizeof(tipnod));
      auxiliar->urm= inceput;
      inceput= auxiliar;
      auxiliar->cheie= n;
      n=n-1;
     }
------------------------------------------------------------
```

- Datorită faptului că inserția noului nod are loc de fiecare dată la **începutul** listei, secvența de mai sus creează lista în ordinea inversă a furnizării cheilor.
- Dacă se dorește crearea listei în **ordine naturală**, atunci este nevoie de o secvență care inserează un nod **la sfârşitul** unei liste.
- Această secvență de program se redactează mai simplu dacă se cunoaşte locația **ultimului** nod al listei.
    - Teoretic lucrul acesta **nu** prezintă nici o dificultate, deoarece se poate parcurge lista de la începutul ei (indicat prin `inceput`) până la detectarea nodului care are câmpul `urm=`**`nil`**.
    - În practică această soluție **nu** este convenabilă, deoarece parcurgerea întregii liste este **ineficientă**.
    - Se preferă să se lucreze cu o **variabilă** pointer ajutătoare `ultim` care indică mereu ultimul nod al listei, după cum `inceput` indică mereu primul nod.
- În prezența lui `ultim`, secvența de program care inserează un nod la sfârşitul unei liste liniare şi concomitent îl actualizează pe `ultim` este următoarea [6.3.2.1.c]:

```
------------------------------------------------------------
/*inserţie la sfârşitul listei*/
                                                    /*[6.3.2.1.c]*/
  /*[1]*/ auxiliar = (tipnod*)malloc(sizeof(tipnod));
  /*[2]*/ auxiliar->urm = NULL;           /*performanta O(1)*/
  /*[3]*/ ultim->urm = auxiliar;
  /*[4]*/ ultim = auxiliar;
  /*[5]*/ ultim->info = 1;
------------------------------------------------------------
```

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

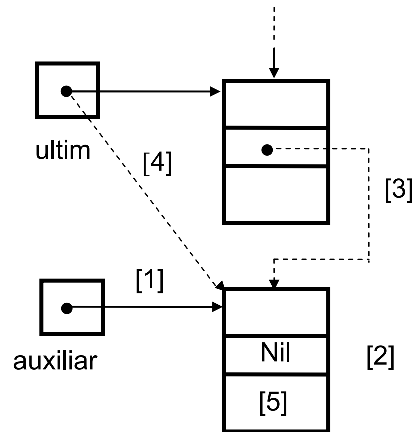

Fig.6.3.2.1.b. Inserția unui nod la sfârşitul unei liste înlănțuite

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

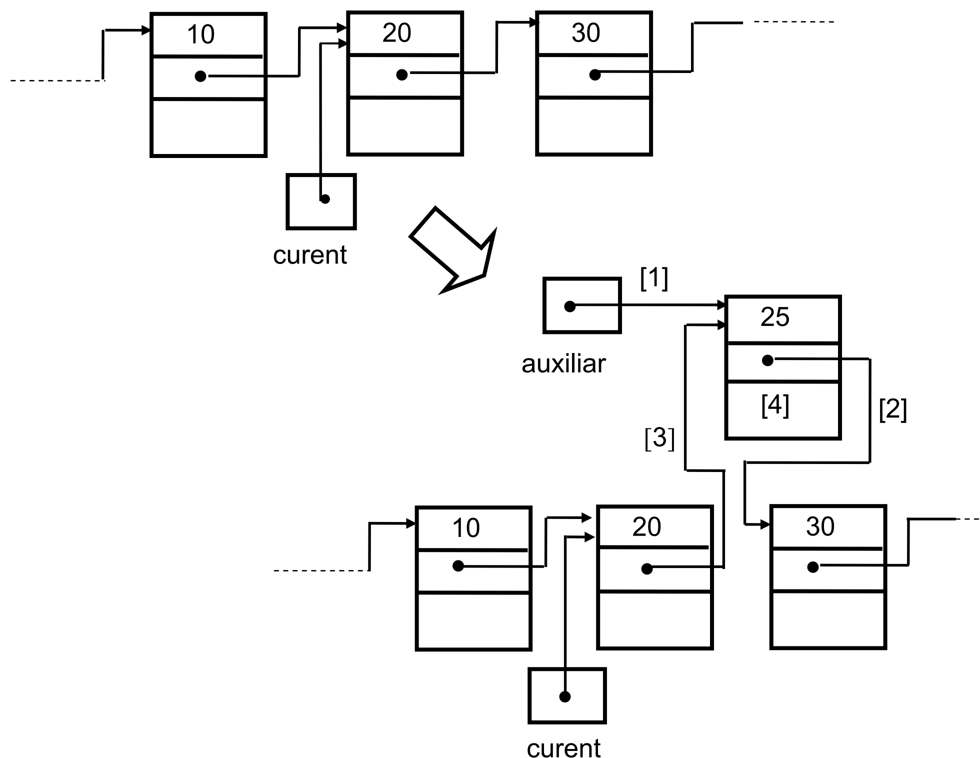

Fig.6.3.2.1.c. Inserţia unui nod nou după un nod precizat (`curent`)

* Secvenţa de program care realizează această inserţie apare în [6.3.2.1.d].

```
------------------------------------------------------------------
/*inserţia unui nod nou după un nod precizat de indicatorul
curent*/
                                              /*[6.3.2.1.d]*/
  /*[1]*/ auxiliar = (tipnod*)malloc(sizeof(tipnod));
  /*[2]*/ auxiliar->urm= curent->urm;    /*performanţa O(1)*/
  /*[3]*/ curent->urm= auxiliar;
  /*[4]*/ auxiliar->info= 2;
------------------------------------------------------------------
```

* 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 inserează un nod nou după nodul indicat de pointerul `curent`
    * 2) Se asignează acest nod cu conţinutul nodului indicat de `curent`
    * (3) 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 program 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*/
                                                        /*[6.3.2.1.e]*/
  /*[1]*/ auxiliar = (tipnod*)malloc(sizeof(tipnod));
  /*[2]*/ *auxiliar= *curent;            /*performanta O(1)*/
  /*[3]*/ curent->urm= auxiliar;
  /*[4]*/ curent->info= ...;
-----------------------------------------------------------------
```

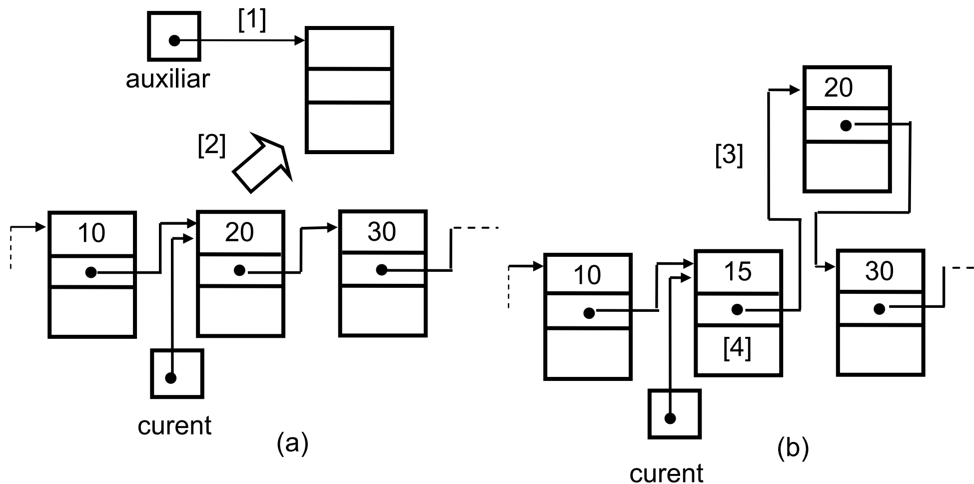

Fig.6.3.2.1.d. Inserţia unui nod nou în faţa unui nod indicat

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`.
- Aceasta se poate realiza prin intermediul fragmentului de program [6.3.2.2.a] în care `auxiliar` este o variabilă pointer ajutătoare.

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

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

- Efectul execuţiei aceste secvenţe de program se poate urmări în figura 6.3.2.2.a.
- Se observă că secvenţa de program 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)*/        /*performantaO(1)*/

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

- Utilizarea pointerului `auxiliar` are însă avantajul că prin intermediul lui, programatorul poate avea acces ulterior la nodul suprimat din listă în vederea disponibilizării zonei de memorie alocate lui, zonă care în condiţiile execuţiei secvenţei [6.3.2.2.b] se pierde.

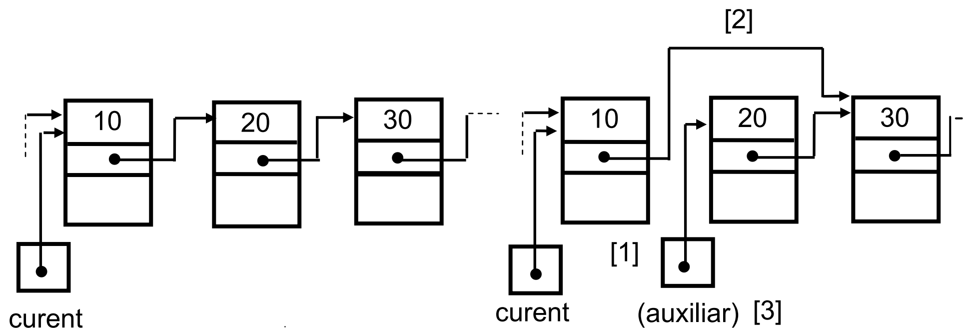

Fig.6.3.2.2.a. Tehnica suprimării succesorului nodului indicat de `curent`

- 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 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)*/                    /*performanta O(1)*/

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

- 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)*/

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

- Se remarcă însă faptul că ambele tehnici de suprimare se pot aplica **numai** dacă nodul indicat de `curent` **nu** este ultimul nod al listei, respectiv numai dacă `curent^.urm` ≠ **`nil`**.
- Pentru a evita acest neajuns se pot utiliza alte modalităţi de implementare a listei înlănţuite spre exemplu, cu nod fictiv sau cu nod fanion.

6.3.2.3. Traversarea unei liste înlănțuite

- Prin **traversarea** unei liste se înțelege executarea unei anumite operații asupra tuturor nodurilor listei.
  - Fie pointerul `p` 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)` operația amintită, a cărei natură nu se precizează.
- În aceste condiții fragmentul de program [6.3.2.3.a] reprezintă **traversarea** în **sens direct** a listei înlănțuite iar fragmentul [6.3.2.3.b] reprezintă traversarea în sens invers.

```
------------------------------------------------------------
/*traversarea unei liste înlantuite*/          /*[6.3.2.3.a]*/

        auxiliar= inceput;
        while (auxiliar!=NULL){
          prelucrare(*auxiliar);
          auxiliar= auxiliar->urm;
       }
------------------------------------------------------------
/*traversarea unei liste înlantuite în sens invers
  (varianta recursiva*/

void traversareinversa(tiplista curent)
  {
    if (curent!=NULL)                            /*[6.3.2.3.b]*/
      {
        traversareinversa(curent->urm);
        prelucrare(*curent);
      }
  }
------------------------------------------------------------
```

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

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

auxiliar= inceput;                              /*[6.3.2.3.c]*/
   while ((auxiliar!=NULL) && (auxiliar->cheie!=x))
     auxiliar= auxiliar->urm;
   if (auxiliar!=NULL) /*nodul cautat este indicat de auxiliar*/
------------------------------------------------------------
```

- Dacă acest fragment se termină cu `auxiliar=`**`nil`**, atunci **nu** s-a găsit nici un nod cu cheia `x`, altfel nodul indicat de `auxiliar` este primul nod având această cheie.
- În legătură cu acest fragment de program trebuie subliniat faptul că în majoritatea compilatoarelor el trebuie considerat **incorect**.
  - Într-adevăr la evaluarea expresiei booleene din cadrul instrucțiunii **`WHILE`**, dacă lista **nu** conține nici un nod cu cheia `x`, atunci în momentul în care `auxiliar` devine **`nil`**, nodul indicat de `auxiliar` **nu** există.
  - În consecință, funcție de implementare, se semnalează eroare, deşi expresia booleană completă este perfect determinată, ea fiind falsă din cauza primei subexpresii.

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

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

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

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

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

6.4. Aplicații ale listelor înlănțuite

6.4.1. Problema concordanței

- Formularea problemei:
    - Se dă un text format dintr-o succesiune de cuvinte,
    - Se baleează textul şi se depistează cuvintele.
    - Pentru fiecare cuvânt se verifică dacă este sau nu la prima apariție.
    - În caz că este la prima apariție, cuvântul se înregistrează,
    - În caz că el a mai fost găsit, se incrementează un contor asociat cuvântului care memorează numărul de apariții.
    - În final se dispune de toate cuvintele distincte din text şi de numărul de apariții al fiecăruia.
- Se menționează că această problemă este importantă, deoarece ea reflectă într-o formă simplificată una din activitățile pe care le realizează un **compilator** şi anume construcția **listei identificatorilor**.
- Programul `Concordanta` [6.4.1.a]:
    - Construieşte o listă înlănțuită conținând cuvintele distincte ale unui text sursă.
    - Inițial lista este vidă, ea urmând a fi completată pe parcursul parcurgerii textului.
    - Procesul de căutare în listă împreună cu inserția sau incrementarea contorului este realizat de procedura `Cauta`.
    - Pentru simplificare, se presupune că "textul" este de fapt o succesiune de numere întregi pozitive care reprezintă "cuvintele".
    - Cuvintele se citesc de la tastatură ele terminându-se cu un cuvânt fictiv, în cazul de față numărul zero care precizează 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.

```
/*Concordanta - varianta C*/

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

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


typedef struct tipnod* tipreferinta;          /*[6.4.1.a]*/
typedef struct {
                int cheie;
                int numar;
                tipreferinta urmator;
               }tipnod;

int cuv;
tipreferinta inceput;         /*#*/

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

  void tiparire(tipreferinta q)
    {
      tipreferinta r;
      r= q;
      while (r!=NULL)
        {
          printf("%i%i\n", ((tipnod*)r)->cheie,((tipnod*)r)->
                                 numar);
          r= ((tipnod*)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).

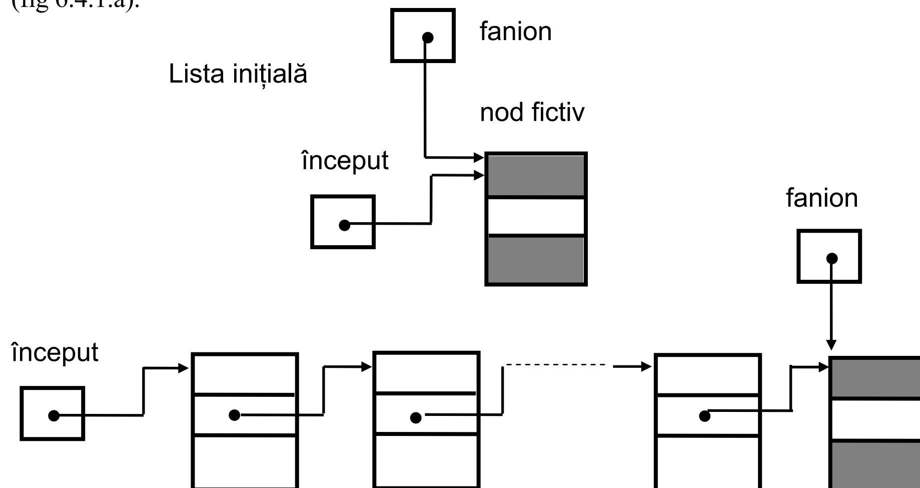

Fig.6.4.1.a. Implementarea unei liste înlănțuite utilizând tehnica nodului fanion

- Tehnica de căutare este similară celei utilizate în cazul tablourilor liniare (&1.4.2.1).
- Pentru aplicarea procedurii de căutare optimizate `Cauta1`, în programul `Concordanta` [6.4.1.a] trebuiesc efectuate două **modificări**:
  - La declararea variabilelor, în locul indicat prin [#], se adaugă declararea nodului fanion: `tipReferinta fanion`;
  - Se modifică inițializarea listei de cuvinte indicată în cadrul programului prin [##], respectiv instrucțiunea `inceput=NULL` se înlocuieşte cu secvența `*inceput = (tipnod*)malloc(sizeof(tipnod)), fanion= inceput.` Prin aceasta lista de cuvinte conține de la bun început un nod (cel fictiv).
- În aceste condiții, procedura `Cauta1` apare în secvența [6.4.1.b].
  - Față de varianta [6.4.1.a] condiția din cadrul instrucției `WHILE` este mai simplă, realizându-se un câştig simțitor de timp.

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

```
------------------------------------------------------------
/*Cautare în liste înlantuite utilizând metoda  "fanionului"*/

void cauta1(int x, tipreferinta* inceput)
  {
    tipreferinta q;
    q= *inceput;
    fanion->cheie= x;
    while (((tipnod*)q)->cheie!=x)                  /*[6.4.1.b]*/
      q= ((tipnod*)q)->urmator;
    if (q==fanion){   /*elementul nu s-a gasit*/
          q= *inceput;
          *inceput = (tipnod*)malloc(sizeof(tipnod))
         ((tipnod*)(*inceput))->cheie= x;
         ((tipnod*)(*inceput))->numar= 1;
         ((tipnod*)(*inceput))->urmator= q;
        }
      else /*s-a gasit*/
        ((tipnod*)q)->numar= ((tipnod*)q)->numar+1;
  }   /*cauta1*/
------------------------------------------------------------
```

6.4.2. Crearea unei liste ordonate. Tehnica celor doi pointeri

- În cazul 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 fişier **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 pointer.
- 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.

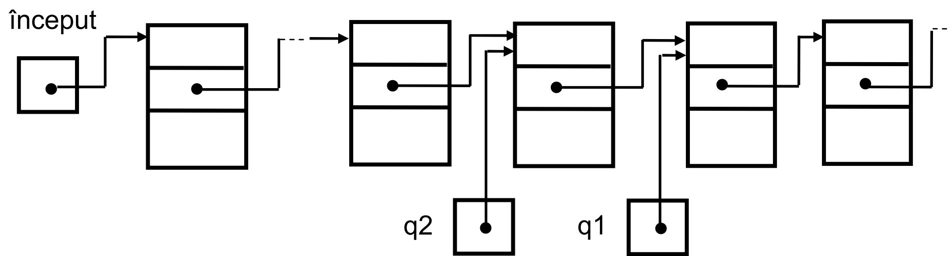

Fig.6.4.2.a. Traversarea unei liste înlănțuite utilizând doi pointeri.

- Se presupune că lista este explorată utilizând metoda nodului `fanion`. Cheia de inserat `x` se introduce inițial în nodul `fanion`
- Cei doi pointeri avansează simultan de-a lungul listei până când cheia nodului indicat de `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 ține cont de faptul că, funcționarea sa corectă presupune existența inițial în listă a **cel puțin** două noduri, deci cel puțin un nod în afară de cel indicat de fanion.
- Din acest motiv lista se va implementa utilizând tehnica celor **două noduri fictive** (fig.6.4.2.b).
- În vederea realizării acestor deziderate programul `Concordanta` (secvența [6.4.1.a]) trebuie modificat după cum urmează;
    - (1) Se adaugă la partea de declarație a variabilelor, (locul indicat cu [#]), declarația `tipreferinta fanion`;
    - (2) Se înlocuieşte instrucția `inceput=NULL` (indicată prin [##]) cu următoarea secvență de instrucțiuni care crează lista vidă specifică implementării prin tehnica celor doi pointeri [6.4.2.a]:

```
----------------------------------------------------------------
*inceput =(tipnod*)malloc(sizeof(tipnod));
*fanion = (tipnod*)malloc(sizeof(tipnod));            [6.4.2.a]
 inceput->urmator= fanion;
----------------------------------------------------------------
```

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

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

void cauta2(int x, tipreferinta inceput)
  {
    tipreferinta q1,q2,q3;
    q2= inceput;
    q1= ((tipnod*)q2)->urmator;
    fanion->cheie= x;
```

```
  while (q1->cheie<x)
    {
      q2= q1;                                 /*[6.4.2.b]*/
      q1= ((tipnod*)q2)->urmator;
    }
  if ((((tipnod*)q1)->cheie==x) && (q1!=fanion))
      ((tipnod*)q1)->numar= ((tipnod*)q1)->numar+1;
    else
      {      /*se creeaza un nou nod indicat de q3 si
               se insereaza între q2^ si q1^*/
        q3 = (tipnod*)malloc(sizeof(tipnod));
        ((tipnod*)q3)->cheie= x;
        ((tipnod*)q3)->numar= 1;
        ((tipnod*)q3)->urmator= q1;
        ((tipnod*)q2)->urmator= q3;
      }
}
```

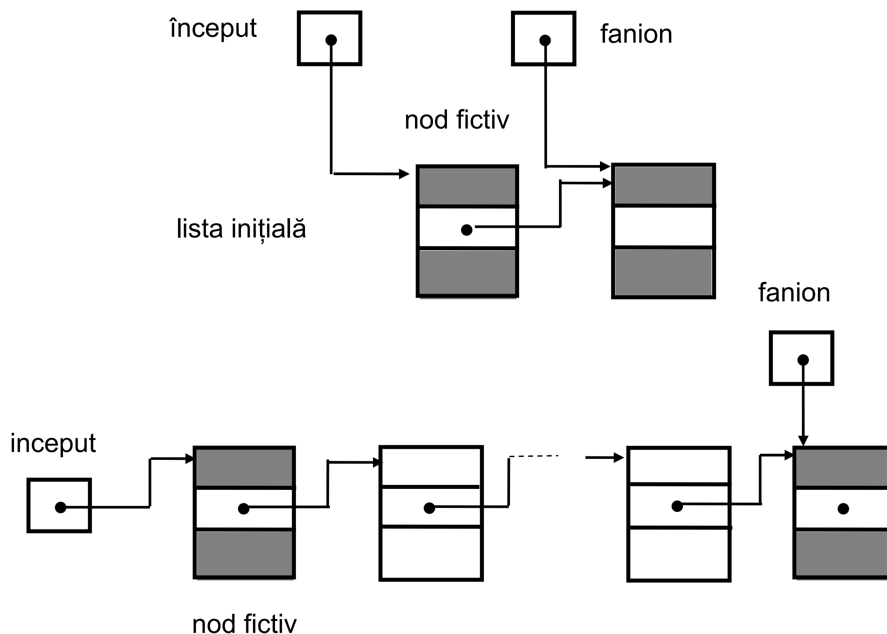

Fig.6.4.2.b. Implementarea unei liste utilizând tehnica celor două noduri fictive.

- Aceste modificări conduc la crearea listei **sortate**.
- Trebuie însă observat faptul că **beneficiul** obținut în urma sortării este destul de **limitat**.
    - El se manifestă **numai** în cazul căutării unui nod care **nu** se găseşte în listă;
    - Această operație necesită parcurgerea în medie a unei **jumătăți** de listă în cazul listelor **sortate** în comparație cu parcurgerea **întregii** liste dacă aceasta **nu** este **sortată**.
    - La căutarea unui nod care se 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 distribuții identice.

* În definitiv, cu toate că sortarea listei practic **nu** costă nimic, se recomandă a fi utilizată numai în cazul unor texte cu multe cuvinte distincte în care acelaşi cuvânt se repetă de puţine ori.

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).

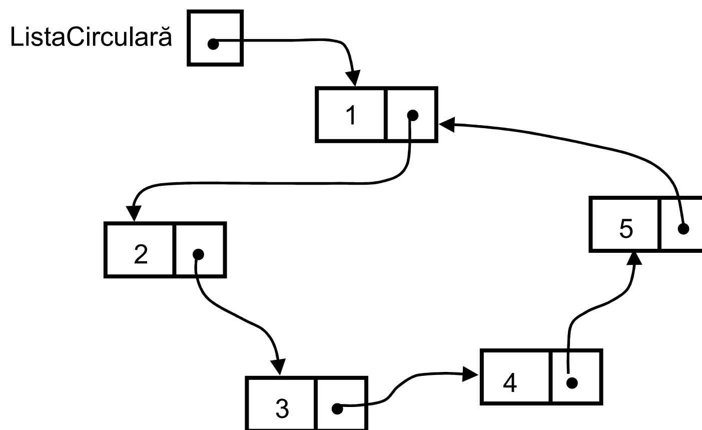

Fig.6.5.1.a. Listă circulară

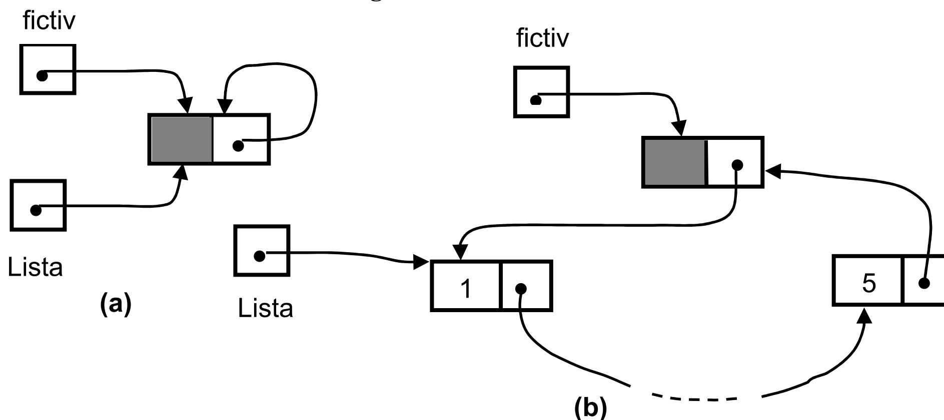

Fig.6.5.1.b. Listă circulară implementată prin tehnica nodului fictiv. Listă vidă (a), normală (b)

* O modalitate simplă de rezolvare a acestor situații 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.
* Listele circulare ridică unele probleme referitoare la inserția **primului** nod în listă şi la suprimarea ultimului nod.

6.5.2. Liste dublu înlănțuite

* Unele aplicații necesită traversarea listelor în ambele sensuri.
* Cu alte cuvinte fiind dat un element oarecare al listei trebuie determinat cu rapiditate atât succesorul cât şi predecesorul acestuia.
* Maniera cea mai rapidă de a realiza acest lucru este aceea de a memora în fiecare nod al listei referințele "**înainte**" şi "**înapoi**"
    * Această abordare conduce la structura **listă dublu înlănțuită**. (fig.6.5.2.a).
    * Prețul care se plăteşte este:
        * (1) Prezența unui câmp suplimentar de tip pointer în fiecare nod
        * (2) O oarecare creştere a complexității procedurilor care implementează operatorii de bază care prelucrează astfel de liste.

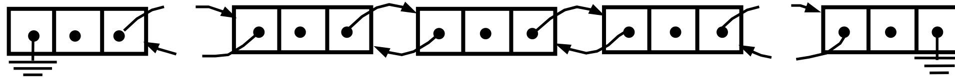

Fig.6.5.2.a. Listă dublu înlănțuită

* Dacă implementarea acestor liste se realizează cu pointeri se pot defini tipurile de date din secvența [6.5.2.a].

```
------------------------------------------------------------
/*Lista dublu înlantuita*/

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

* Pentru exemplificare se prezintă procedura de **suprimare** a elementului situat în poziția `p` a unei liste dublu înlănțuite.
* În secvența [6.5.2.b] se prezintă maniera în care se realizează această acțiune, în accepțiunea faptului că nodul suprimat **nu** este nici primul **nici** ultimul nod al listei.
    * (1) Se localizează nodul **precedent** 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(tippozitie* p)
  {                                                      /*[6.5.2.b]*/
    if ((*p)->anterior!=NULL)  /*nu este primul nod*/
      (*p)->anterior->urmator= (*p)->urmator;
    if ((*p)->urmator!=NULL)  /*nu este ultimul nod*/
      (*p)->urmator->anterior= (*p)->anterior;
  }   /*Suprima*/
------------------------------------------------------------------
```

* În practica programării, se pot utiliza diferite tehnici de implementare a listelor dublu 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.

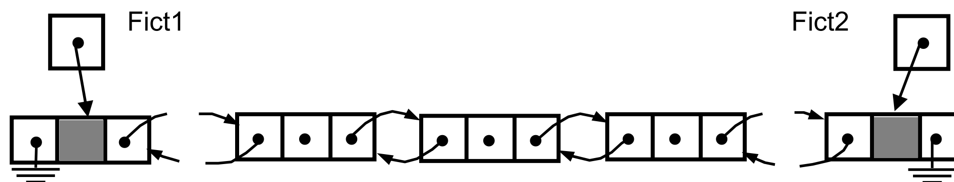

Fig.6.5.2.b. Listă dublu înlănţuită. Varianta cu două noduri fictive

* (2) O altă variantă de implementare se bazează pe utilizarea:
    * (1) De **indicatori pentru cele două capete ale listei**
    * (2) Unui **contor** de noduri pentru sesizarea situaţiilor limită (fig.6.5.2.c).

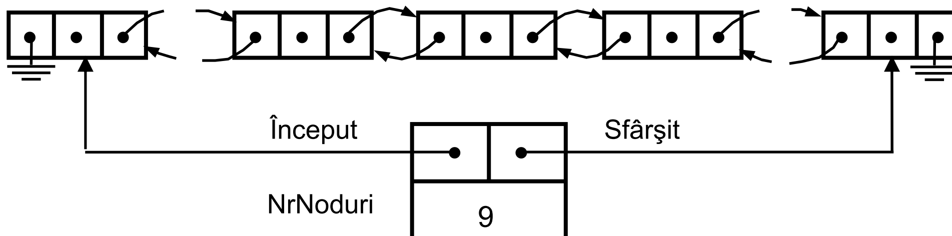

Fig.6.5.2.c. Listă dublu înlănţuită. Varianta cu indicatori la capete

* 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.

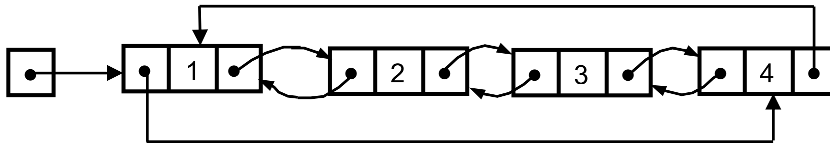

Fig.6.5.2.d. Listă dublu înlănțuită circulară

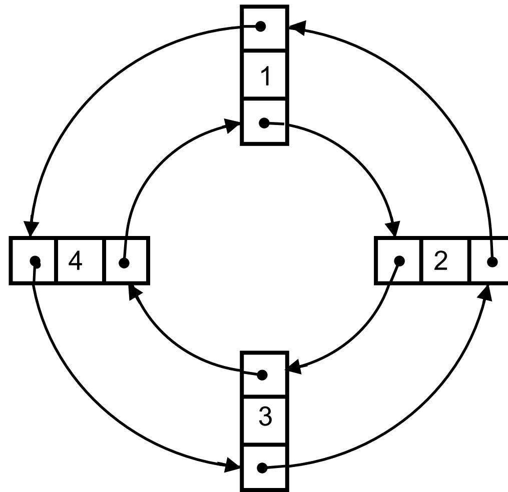

Fig.6.5.2.e. Listă dublu înlănțuită circulară

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

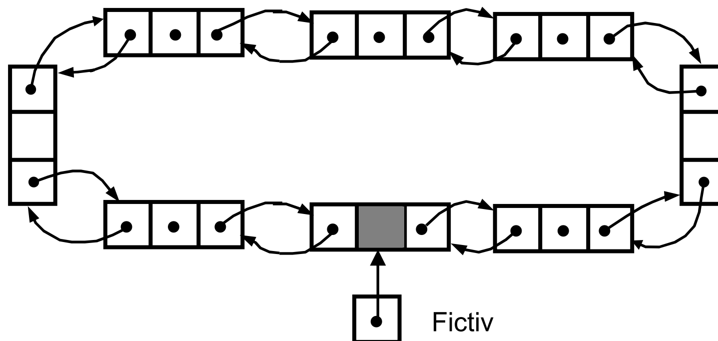

Fig.6.5.2.f. Listă dublu înlănțuită circulară. Varianta cu nod fictiv

6.5.3. Stive

- O **stivă** este un tip special de listă în care toate inserările 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 maniera cea mai convenabilă şi în acelaşi timp cea mai sigură 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 în [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:
      s: TipStiva;
      x: TipElement.                          [6.5.3.1.a]

Operatori:
      1.Initializeaza(s:TipStiva){:TipPozitie}; - face
         stiva s vidă.
      2.VarfSt(s:TipStiva):TipElement; - furnizează
         elementul din vârful stivei s.
      3.Pop(s:TipStiva); - suprimă elementul din vârful
         stivei.
      4.Push(x:TipElement,s:TipStiva); - inserează
         elementul x în vârful stivei s. Vechiul vârf
         devine elementul următor ş.a.m.d.
      5.Stivid(s:TipStiva):boolean; - 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.
    * 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.

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

typedef tiplista tipstiva;
tipstiva s;
tippozitie p;                          /*[6.5.3.1.b]*/
tipnod x;
boolean b;

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

- 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 tipul 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 inserările şi suprimările se fac **numai** în vârf.
    - Astfel se poate considera drept **bază** a stivei sfârşitul tabloului (indexul său cel mai mare), stiva crescând în sensul descreşterii indexului în tablou.
    - Un cursor numit `vârf` indică poziţia curentă a ultimului element al stivei (fig.6.5.3.2).
- Structura de date abstractă care se defineşte pentru această implementare este următoarea [6.5.3.2.a].

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

enum { lungimemaxima = 50};
typedef struct tipstiva {
  int varf;                                 /*[6.5.3.2.a]*/
  tipelement elemente[lungimemaxima];
} tipstiva;

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

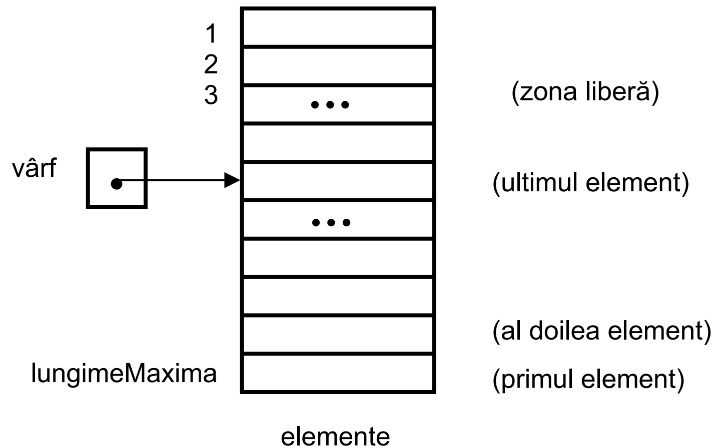

Fig.6.5.3.2. Implementarea TDA Stivă cu ajutorul structurii tablou

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

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

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

void initializeaza(tipstiva* s)
  {
    s->varf= lungimemaxima+1;
  }   /*Initializeaza*/

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

tipelement varfst(tipstiva* s)
  {
    boolean er;
    tipelement varfst_result;
    if (stivid(*s)){
          er= true;
          printf("stiva este vida");
        }
      else                                   /*[6.5.3.2.b]*/
```

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

void pop(tipstiva* s, tipelement* 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(tipelement x, tipstiva* 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*/
----------------------------------------------------------------
```

6.5.4. Cozi

- Cozile sunt o altă categorie specială de liste în care elementele sunt inserate la un capăt (**spate**) şi sunt suprimate la celălalt (**cap**).
- 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 la celălalt capăt (cap).

Notaţii:

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

Operatori (setul 1):

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

Operatori (setul 2):

1. ***CreazaCoadaVida***(*C: TipCoada*); - face coada *C* vidă.
2. ***CoadaVida***(*C: TipCoada*)*: boolean;* - este adevărat dacă şi numai dacă *C* este o coadă vidă.
3. ***CoadaPlina***(*C: TipCoada*)*: boolean;* - este adevărat dacă şi numai dacă *C* este o coadă plină (operator dependent de implementare).
2. ***InCoada***(*C: TipCoada x: TipElement*); - inserează elementul *x* la spatele cozii *C* (***EnQueue***).
3. ***DinCoada***(*C: TipCoada, x: TipElement*); - suprimă primul element al lui *C* (***DeQueue***).

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

typedef tiplista tipcoada;
tipcoada c;
tippozitie p;
tipelement x;                               /*[6.5.4.1.b]*/
boolean b;

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

6.5.4.2. Implementarea cozilor cu ajutorul pointerilor

- Ca şi în cazul stivelor, orice **implementare** a listelor este valabilă şi pentru cozi.
- Pornind însă de la observaţia că inserările 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 poate 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
structurilor de date}

#include <stdlib.h>
typedef struct tipnod* tipreferintanod;
typedef struct tipnod {
  tipelement element;                    /*[6.5.4.2.a]*/
  tipreferintanod urm;
} tipnod;
----------------------------------------------------------------
```

- În aceste condiţii se poate defini o structură de tip 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].

```
----------------------------------------------------------------
/* Implementarea cozilor cu ajutorul pointerilor - definirea lui
TipCoad */

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

- 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(tipcoada* c)
  {
    c->inceput = (tipnod*)malloc(sizeof(tipnod)); /*creaza nodul
fictiv*/
    c->inceput->urm= NULL;
    c->spate= c->inceput; /*nodul fictiv este primul si
                          ultimul nod al cozii*/
  }   /*Initializeaza*/

boolean vid(tipcoada c)
  {
    boolean vid_result;
    if (c.inceput=c.spate)
```

```
        vid_result=true;
        else vid_result=false;
    return vid_result;
  }   /*Vid*/

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

void adauga(tipelement x, tipcoada* c)
  {
    c->spate->urm = (tipnod*)malloc(sizeof(tipnod)); /*se adauga
                                 o noua celula la spatele
cozii*/
    c->spate= c->spate->urm;
    c->spate->element= x;
    c->spate->urm= NULL;
  }   /*Adauga*/

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

6.5.4.3. Implementarea cozilor cu ajutorul tablourilor circulare

- Reprezentarea listelor cu ajutorul tablourilor, prezentată în &6.3.1, **poate** fi utilizată şi pentru cozi, dar **nu** este foarte eficientă.
    - Într-adevăr, pointerul 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.

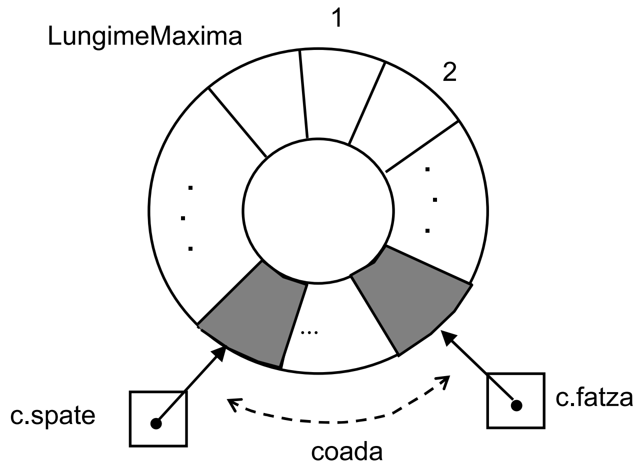

Fig.6.5.4.3. Implementarea circulară a unei cozi

- Coada se găseşte undeva în jurul cercului, în **poziţii consecutive**, având spatele undeva înaintea feţei la parcurgerea cercului în sensul acelor de ceasornic.
- **Înlănţuirea** unui element presupune mişcarea pointerului `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 pointerului `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.
- În cazul implementării ce apare în continuare, pointerul `C.fatza` indică primul element al cozii, iar pointerul `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 coada reprezentată în figura 6.5.4.3 este **plină** (conţine `LungimeMaxima` elemente), pointerul `C.spate` va indica poziţia **adiacentă** lui `C.fatza` în sens trigonometric pozitiv.
- 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.spate` se află cu o poziţie în faţa lui `C.fatza,` la parcurgerea în sens trigonometric pozitiv a cozii.
- În vederea rezolvării acestei situaţii, în tablou se vor introduce **doar** `LungimeMaxima-1` elemente, deşi în tablou există `LungimeMaxima` 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 trigonometric pozitiv al parcurgerii cozii.

- În continuare se prezintă implementarea operatorilor definiți peste o structură de date de tip coadă definită după cum urmează [6.5.4.3.a]:

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

typedef struct tipcoada {                          /*[6.5.4.3.a]*/
  tipelement elemente[lungimemaxima];
  tipindice fatza,spate;
} tipcoada;
------------------------------------------------------------------
```

- Secvența de instrucții corespunzătoare apare în [6.5.4.3.b].

- Funcția `Avanseaza(C)` furnizează poziția următoare celei curente în coadă.

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

 tipindice avanseaza(tipindice* i)
     {
       tipindice avanseaza_result;
       avanseaza_result= (*i % lungimemaxima)+1;
       return avanseaza_result;
     }   /*Avanseaza*/

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

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

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

   void adauga(tipelement x, tipcoada* c)
```

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

  void scoate(tipcoada* 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 `DimMax` 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 dezvoltaţi cu uşurinţă în baza exemplelor prezentate anterior.

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

enum { dimmax = 100,
      dimcoada = dimmax+1};

typedef int tipelement;
typedef unsigned char tipindex;
                                            /*[6.5.4.3.c]*/
typedef unsigned char tipcontor;

typedef tipelement tiptablou[dimmax+1];
typedef struct tipcoada {
  tipindex fatza,spate;
  tipcontor contor;
  tiptablou elemente;
} tipcoada;

tipcoada c;
tipelement x;
```

6.5.5. TDA Coadă bazată pe prioritate

- Coada bazată pe prioritate (**"priority queue"**) este structura de date abstractă care permite inserția unui element şi suprimarea **celui mai prioritar** element.
- O astfel de structură diferă atât față de structura **coadă** (din care se suprimă **primul** venit, deci **cel mai vechi**) cât şi față 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 utilizând **priorități corespunzătoare**.
- **Aplicațiile** acestor cozi sunt: sistemele de simulare (unde cheile pot corespunde unor evenimente "de timp" ce trebuie să decurgă în ordine), planificarea job-urilor, traversări speciale ale unor structuri de date, etc.
- Considerând coada bazată pe prioritate drept o structură abstractă de date ale cărei elemente sunt articole cu chei (sau priorități), definirea acesteia apare în secvența [6.5.5.a].

TDA Coadă bazată pe prioritate

Modelul matematic: o secvență finită de noduri. Toate nodurile aparțin unui aceluiaşi tip numit **tip de bază.** O **coadă bazată pe prioritate** este de fapt o listă specială în care se permite inserția normală şi suprimarea doar a celui mai prioritar element.

Notații:

q: TipCoadaBazataPePrioritate; [6.5.5.a]
x: TipElement;

Operatori:

1. ***Initializează****(q: TipCoadaBazataPePrioritate);* - face coada *q* vidă.
2. ***Inserează****(x: TipElement, q: CoadaBazataPePrioritate);* - inserția unui nou element *x* în coada *q*.
3. ***Extrage****(q: CoadaBazataPePrioritate): TipElement;* - extrage cel mai prioritar element al cozii *q*.
4. ***Inlocuieşte****(q: CoadaBazataPePrioritate, x: TipElement): TipElement;* - înlocuieşte cel mai prioritar element al cozii *q* cu elementul *x*, mai puțin în situația în care noul element este cel mai prioritar element. Returnează cel mai prioritar element.
5. ***Schimbă****(q: TipCoadaBazataPePrioritate), x: TipElement; p: TipPrioritate);* - schimbă prioritatea elementului *x* al cozii *q* şi îi conferă valoarea *p*.
6. ***Suprimă****(q: TipCoadaBazataPePrioritate, x: TipElement);* - suprimă elementul *x* din coada *q*.
7. ***Vid****(q: TipCoadaBazataPePrioritate): boolean;* - 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 inserţii.
- Cozile bazate pe prioritate pot fi în general implementate în diferite moduri unele bazate pe structuri simple altele pe structuri avansate, fiecare presupunând însă performanţe diferite pentru operatorii specifici.
- În continuare vor fi prezentate unele dintre aceste posibilităţi.

6.5.5.1. 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 [6.5.5.1.a] apare structura de date corespunzătoare acestei abordări iar în secvenţa [6.5.5.1.b] implementarea operatorilor `Insereaza` şi `Extrage`.

```
----------------------------------------------------------------
/* Implementarea cozilor bazate pe prioritate cu tablouri -
operatorii Insereaza şi Extage */

typedef struct tipelement {
  tipprioritate prioritate;
  tipinfo info;                          /*[6.5.5.1.a]*/
} tipelement;
typedef struct tipcoadabazatapeprioritate {
  tipelement elemente[dimmax];
  unsigned char nrelemente;
} tipcoadabazatapeprioritate;
----------------------------------------------------------------
void insereaza(tipelement x,
               tipcoadabazatapeprioritate q)
  {      /*O1*/
    q.nrelemente= q.nrelemente+1;
    q.elemente[q.nrelemente-1]= x;
  }   /*Insereaza*/
tipelement extrage(tipcoadabazatapeprioritate q)
  {
    unsigned char j,max;/*O(n)*/
      /*[6.5.5.1.b]*/
    tipelement extrage_result;
    max= 1;
    for( j=1; j <= q.nrelemente; j ++)
      if (q.elemente[j-1].prioritate >
             q.elemente[max-1].prioritate)    max= j;
    extrage_result= q.elemente[max-1];
    q.elemente[max-1]= q.elemente[q.nrelemente-1];
    q.nrelemente= q.nrelemente-1;
    return extrage_result;
  }   /*Extrage*/
----------------------------------------------------------------
```

        - Pentru **inserție** se incrementează `nrElemente` şi se adaugă elementul de inserat tabloului `q.elemente`, operație 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 poziția celui care a fost extras şi se decrementează `nrElemente`. Operația necesită o regie O*(n)*.
        - Implementarea lui `Inlocuieste` este similară, ea presupunând o inserție urmată de o extragere.
        - Redactarea celorlalți operatori în acest context nu ridică nici un fel de probleme.
- În implementarea cozilor bazate pe prioritate se pot utiliza şi **tablouri ordonate**.
    - Elementele cozii sunt păstrate în tablou în ordinea crescătoare a priorităților.
    - In aceste condiții, operatorul `Extrage`, returnează ultimul element `q.elemente[nrElemente]` şi decrementează `nrElemente`, operație ce durează un interval constant de timp.
    - Operatorul `Insereaza` presupune mutarea spre dreapta a elementelor mai mari ca şi elementul de inserat, operație care durează O*(n)*.
- Operațiile care se referă la cozi bazate pe prioritate pot fi utilizate în implementarea unor algoritmi de **sortare**.
    - Spre exemplu utilizând în mod **repetat** operația `Insereaza` pentru a crea o coadă bazată pe priorități
    - Iar apoi **extrăgând** pe rând elementele până la golirea cozii se obține secvența sortată a elementelor în ordinea descrescătoare a priorității lor.
    - Dacă se utilizează o coadă bazată pe priorități reprezentată ca un tablou neordonat se obține sortarea prin **selecție**;
    - Dacă se utilizează o coadă bazată pe priorități reprezentată ca un tablou ordonat, se obține sortarea prin **inserție**.

6.5.5.2. Implementarea cozilor bazate pe prioritate cu ajutorul listelor înlănțuite

- În implementarea cozilor bazate pe prioritate pot fi utilizate şi **listele înlănțuite** în variantă **ordonată** sau **neordonată**.
- Această abordare **nu** modifică principial implementarea operatorilor specifici, dar face posibilă realizarea mai **eficientă** a fazelor de inserție şi suprimare a elementelor cozii datorită flexibilității listelor înlănțuite.
- În continuare se prezintă un **exemplu** de realizare a operației `Extrage`, utilizând pentru implementarea cozilor bazate pe prioritate listele înlănțuite neordonate.
- Se utilizează următoarele structuri de date [6.5.5.2.a].

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

typedef int tipelement;
typedef struct tipnod* tipreferintanod;
typedef struct tipnod {
  tipelement element;                   /*[6.5.5.2.a]*/
  tipreferintanod urm;
} tipnod;
typedef tipreferintanod coadabazatapeprioritate;
--------------------------------------------------------------
```

- 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 conține nici un element având poziționat numai câmpul `urm` (tehnica **“nodului fictiv”**)[6.5.5.2.b].

```
------------------------------------------------------------------
/* Implementarea operatorului Extrage*/

#include <stdlib.h>
tipreferintanod extrage(coadabazatapeprioritate* q){
    tipreferintanod curent;  /*nodul din fata celui baleat*/
    tipelement mare;   /*cel mai mare element curent*/
    tipreferintanod anterior;  /*nodul anterior celui mai mare*/
    tipreferintanod extrage_result;
    if (vid(*q))
      printf("coada este vida");                /*[6.5.5.2.b]*/
    else{
        mare= (*q)->urm->element; /*q indica nodul fictiv*/
        anterior= *q; curent= (*q)->urm;
        while (curent->urm != NULL){/*compara valoarea lui mare
                     cu valoarea urmatoare elementului curent*/
           if (curent->urm->element > mare){
              anterior= curent;  /*anterior retine pointerul
              nodului ce precede nodul cu cheia cea mai mare */
              mare= curent->urm->element;
           }   /*IF*/
           curent= curent->urm;
        }   /*WHILE*/
        extrage_result= anterior->urm; /*pozitioneaza pointerul
                                      pe cel mai mare element*/
        curent= anterior->urm;
        anterior->urm= anterior->urm->urm;
    }   /*ELSE*/
    return extrage_result;
  }   /*Extrage*/
------------------------------------------------------------------
```

- Implementarea celorlalte funcții referitoare la cozile bazate pe prioritate utilizând liste neordonate nu ridică nici un fel de probleme.
- Aceste implementări **simple** în multe situații sunt mai utile decât cele bazate pe modele sofisticate.
    - Astfel implementarea bazată pe **liste neordonate** e potrivită în situațiile în care se fac multe inserții şi mai puține extrageri,
    - În schimb, implementarea bazată pe **liste ordonate** este potrivită dacă prioritățile elementelor care se inserează au tendința de a fi apropiate ca valoare de prioritatea maximă.

6.5.5.3. Implementarea cozilor bazate pe prioritate cu ajutorul ansamblelor

- Un **ansamblu** este un **arbore binar parțial ordonat** reprezentat printr-un **tablou**, ale cărui elemente satisfac condițiile ansamblului (&3.2.5).
    - În particular cea mai mare (prioritară) cheie este aşezată întotdeauna în prima poziție a tabloului care materializează ansamblul.
- Algoritmii care operează asupra structurilor de date de tip **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 îl prelucrează de jos în sus.
- Pentru simplificare, se presupune că elementele cozii (ansamblului) :
  - (1) Sunt formate numai din **prioritate**
  - (2) Sunt memorate într-un **tablou** cu dimensiunea maximă precizată (`DimMax`), a cărui dimensiune curentă este păstrată în variabila `nrElemente` care face parte din definiţia cozii [6.5.5.3.a].

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

typedef struct tipelement {
  tipprioritate prioritate;
  tipinfo info;                        /*[6.5.5.3.a]*/
} tipelement;
typedef unsigned char tipindice1;

typedef struct tipcoadabazatapeprioritate {
  tipelement ansamblu[dimmax];
  unsigned char nrelemente;
} tipcoadabazatapeprioritate;
----------------------------------------------------------------
```

- Pentru **construcţia** unui ansamblu se utilizează de regulă operatorul `Insereaza` .
  - O 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), situaţie care necesită **avansul** elementului în ansamblu prin **interschimbare** cu părintele său.
  - Acest proces se repetă până când elementul devine mai **mic** ca şi părintele său, sau a ajuns pe **prima** poziţie a ansamblului.
- Procedura `UpHeap` prezentată în secvenţa [6.5.5.3.b] implementează această metodă respectiv avansul elementului nou introdus de pe ultima poziţie a ansamblului, de jos în sus în ansamblu până la locul potrivit
- Această metodă opusă procedurii `Deplasare` utilizată la sortarea "heap sort" (&3.2.5).
- În aceeaşi secvenţă apare şi procedura care implementează inserţia propriu-zisă.

```
----------------------------------------------------------------
/* Implementarea cozilor bazate pe prioritate cu ajutorul
ansamblelor - operatorii UpHeap şi Insereaza */

int reccmp(tipelement x,tipelement y){
   return x.prioritate-y.prioritate;}

void upheap(tipcoadabazatapeprioritate q,
            tipindice1 k){
   tipelement v;
   v= q.ansamblu[k-1];
   q.ansamblu[0].prioritate=(1/*cea mai mare prioritate*/)+1;
   while (q.ansamblu[k / 2].prioritate <= v.prioritate)   /*look
                                                         ahead*/
        {
        q.ansamblu[k]= q.ansamblu[k / 2];
```

```
        k= k / 2;
      }
    q.ansamblu[k]= v;                          /*[6.5.5.3.b]*/
  }   /*UpHeap*/

void insertie(tipelement x,
              tipcoadabazatapeprioritate q){
     tipelement v;
     q.nrelemente= q.nrelemente+1;
     q.ansamblu[q.nrelemente]= v;
     upheap(q,q.nrelemente);
  }   /*Insertie*/
-----------------------------------------------------------------
```

- Dacă în operatorul `UpHeap`, `(k DIV 2)` se înlocuieşte cu `(k-1)` se obţine în esenţă **sortarea prin inserţie** la care găsirea locului în care se inserează noul element se realizează verificând şi deplasând **secvenţial** elementele cu câte o poziţie spre dreapta.
- În procedura `UpHeap` deplasarea se face **nu** liniar 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.
- `q.elemente[0]` se utilizează pe post de **fanion** care se asignează iniţial cu o prioritate mai **mare** decât oricărui alt element.

- Operatorul `Inlocuieste` presupune înlocuirea celui mai prioritar element (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.nrElemente]` (ultimul element al ansmblului) pe prima poziţie
  - (3) Deplasarea primului element de sus în jos, spre baza ansamblului, până la locul potrivit
  - (4) Decrementarea numărului de elemente (`q.nrElemente`).
- Deplasarea de sus în jos în ansamblu de la poziţia `k` spre baza acestuia este materializată de procedura `DownHeap` [6.5.5.3.c].

```
-----------------------------------------------------------------
/* Implementarea cozilor bazate pe prioritate cu ajutorul
ansamblelor - operatorul DownHeap*/

void downheap(tipcoadabazatapeprioritate q,
              tipindice1 k){
    tipindice1 j;
    tipelement v;
    boolean ret;
    v= q.ansamblu[k];                          /*[6.5.5.3.c]*/
    ret= false;
    while((k < q.nrelemente / 2) && (! ret))
        {
          j= k+k;
          if (j < q.nrelemente)
          if (q.ansamblu[j].prioritate <
```

```
            q.ansamblu[j+1].prioritate)  j= j+1;
          if (reccmp(v, q.ansamblu[j]) >= 0)
            ret= true;
            else
           {
               q.ansamblu[k]= q.ansamblu[j]; k= j;
                }
}   /*WHILE*/
    q.ansamblu[k]= v;
  }   /*DownHeap*/
----------------------------------------------------------------
```

- Deplasarea în ansamblu se realizează interschimbând elementul de pe poziția curentă `k` cu cel mai prioritar dintre fiii săi şi avansând 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 situația 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.nrElemente` **`DIV`** `2)`
  - A doua corelată cu găsirea poziției elementului de inserat în interiorul ansamblului (variabila booleană `ret`).
- Cu aceste precizări, implementarea operatorilor `Extrage` şi `Inlocuieste` este imediată [6.5.5.3.d].

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

tipelement extrage(tipcoadabazatapeprioritate q)
  {
    tipelement extrage_result;
    extrage_result= q.ansamblu[1];
    q.ansamblu[1]= q.ansamblu[q.nrelemente];
    q.nrelemente= q.nrelemente-1;
    downheap(q,1);
    return extrage_result;
  }   /*Extrage*/                            /*[6.5.5.3.d]*/

tipelement inlocuieste(tipcoadabazatapeprioritate q,
                       tipelement x)
  {
    tipelement inlocuieste_result;
    q.ansamblu[0]= x;
    downheap(q,0);
    inlocuieste_result= q.ansamblu[0];
    return inlocuieste_result;
  }   /*Inlocuieste*/

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

- În cazul operatorului `Inlocuieste` se utilizează şi poziția `q.elemente[0]` ai cărei fii sunt pozițiile `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 `UpHeap(q,k)`, iar dacă prioritatea sa este coborâtă, procedura `DownHeap(q,k)` rezolvă situaţia.

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 o reprezentare grafică a unei structuri multilistă iar în secvenţa [6.6.a] se prezintă un exemplu de definire a unei astfel de structuri

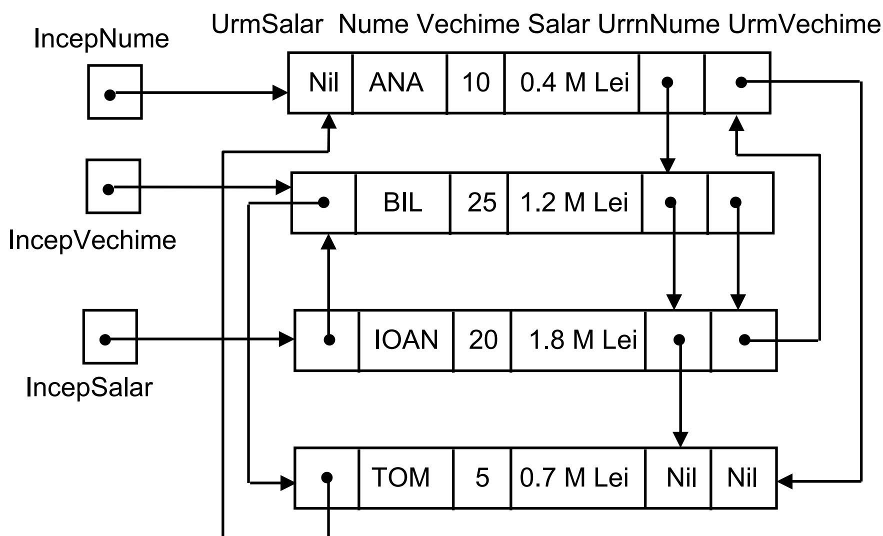

Fig.6.6.a. Exemplu de structură de date multilistă

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

typedef char* tipnume;
typedef struct tipinfo1 {
    tipnume nume;
    int vechime;
    float salar;                                    /*[6.6.a]*/
    } tipinfo1;
typedef struct tipnod* tipreferintanod;
```

```
typedef struct tipnod {
    tipinfo1 informatie;
    tipreferintanod urmsalar,urmnume,urmvechime;
   }tipnod;
tipreferintanod incepnume,incepvechime,incepsalarii;
----------------------------------------------------------------
```

- Avantajul utilizării unor astfel de structuri este evident.
    - Prezența mai **multor înlănțuiri** într-un acelaşi nod,
    - Respectiv **apartenența simultană** a aceluiaşi nod la mai multe liste
    - Asigură acestei structuri de date o **flexibilitate** deosebită,
    - Avantaj care coroborat cu o manipulare relativ **facilă** specifică structurilor înlănțuite,
    - Este exploatat cu precădere la implementarea **bazelor de date**.

6.7. Rezumat

- O **listă** este o **structură de date dinamică** care se defineşte pornind de la noțiunea de **vector.**
- Pentru **TDA listă** se definesc două categorii de **seturi de operatori**: unul **restrâns** destinat implementării cu ajutorul tablourilor şi unul **extins** destinat implementării cu ajutorul pointerilor.
- **Listele se pot implementa** cu ajutorul **structurii tablou** şi cu ajutorul **pointerilor**. În cazul pointerior se utilizează **tehnici specifice de inserție, suprimare** şi **traversare** a listelor
- Listele au foarte multe **aplicații**. Dintre cele mai cunoscute se remarcă **problema concordanței** şi creearea **listelor înlănțuite ordonate**.
- Structura listă stă la baza multor alte structuri de date considerate drept **liste speciale**. Dintre acestea se amintesc **listele circulare, listele dublu înlănțuite, stivele, cozile** şi **cozile bazate pe prioritate**. Fiecare dintre acestea se bucură de **implementări specifice** funcție de contextul aplicației şi de performanța impusă.
- O categorie aparte de liste o reprezintă **listele generalizate**.

6.8. Exerciții

1. Ce este o *listă*? Descrieți *modelul matematic* al TDA listă.
2. Care sunt principalii operatori ai *setului restrâns*? Care sunt operatorii care diferențiază *setul extins* de operatori ai TDA listă?
3. Realizați o implementare în limbajul C a *setului restâns* de operatori utilizând structura tablou. Precizați performanța fiecărui operator implementat în termenii funcției O.
4. Realizați o implementare în limbajul C a operatorilor *InserInceput, InserDupa, InserInFata*, *SuprimaUrm* şi *SuprimaCurent* aparținând setul extins utilizând pointerii.
5. Enunțați *problema concordanței*. Se cere să se realizeze o implementare în limbajul C a *problemei concordanței*. Textul analizat va fi format dintr-o succesiune de numere întregi. Se consideră drept terminator de text cifra 0.
6. Se cere să se realizeze un program care citeşte de la tastaură un şir de numere întregi şi construieşte o *listă înlănțuită ordonată*. Se va defini o funcție care implementează căutarea şi inserția utilizând *tehnica celor doi pointeri*.
7. Ce este o *listă circulară*? Dar o *listă dublu înlănțuită*? Descrieți structurile de date specifice definirii celor două tipuri de liste cu ajutorul pointerilor.

8. Se cere să se impelementeze *TDA stivă* utilizînd structura tablou. Se va preciza performanța fiecărui operator implementat în termenii funcției O.
9. Se cere să se impelementeze *TDA coadă* (setul 1) utilizând pointerii. Precizați performanța operatorilor implementați în termenii funcției O.
10. Se cere să se impelementeze *TDA coadă bazată pe prioritate* utilizând tablouri. Se va preciza performanța fiecărui operator implementat în termenii funcției O.
11. Se cere să se impelementeze *TDA coadă bazată pe prioritate* utilizând structura *ansamblu*. Precizați performanța fiecărui operator implementat în termenii funcției O.
12. Ce este o *listă generalizată*? Dați un exemplu de listă generalizată şi precizați structura de date aferentă.