SDA (PC2)
Curs 7
Liste speciale
(continuare)

Iulian Năstac

Recapitulare

Operaţii ce ţin de o listă înlănţuită:

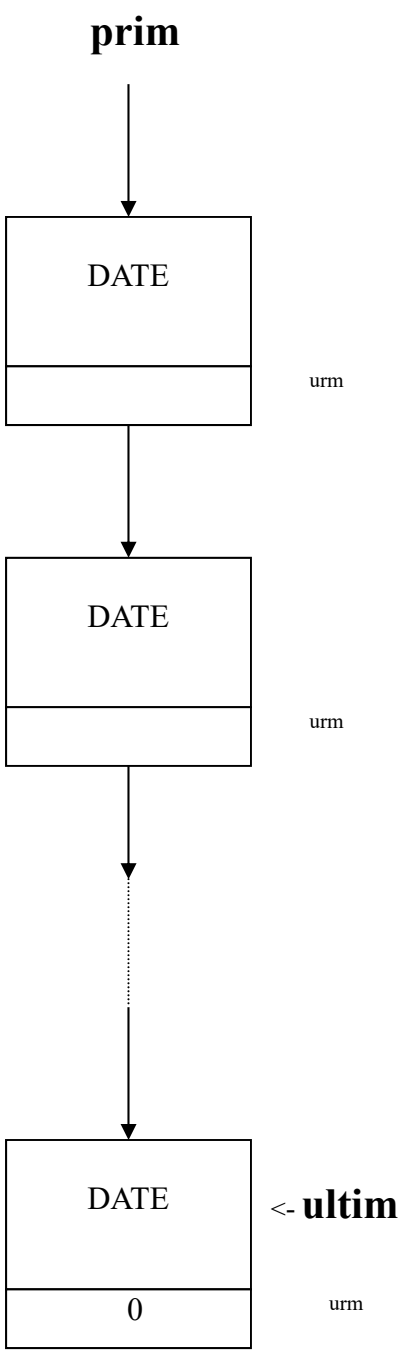
a) crearea listei înlănţuite;

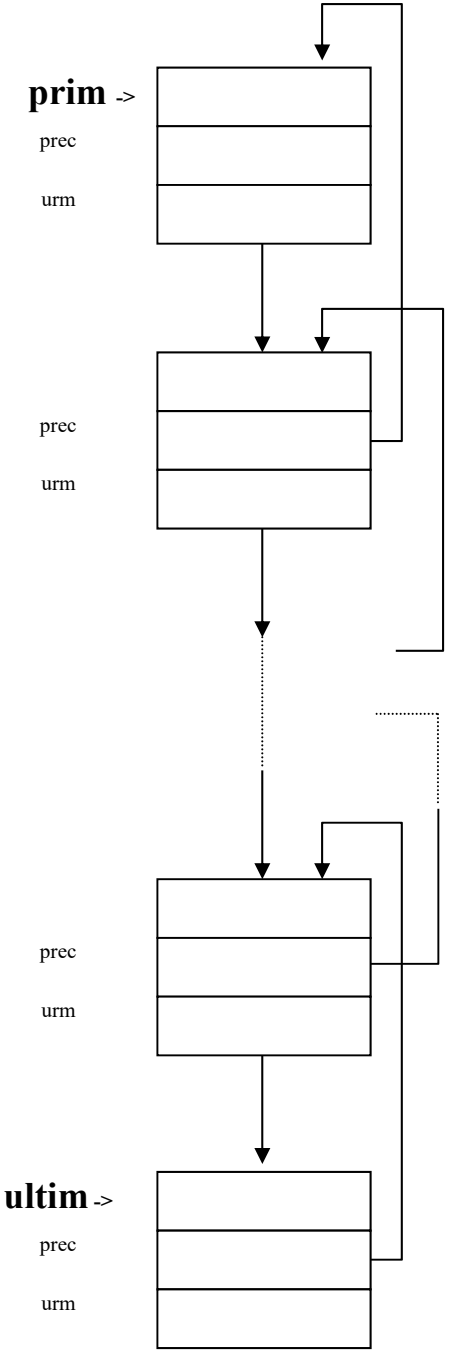
b) accesul la un nod oarecare al listei;

c) inserarea unui nod într-o listă înlănţuită;

d) ştergerea unui nod dintr-o listă înlănţuită;

e) ştergerea unei liste înlănţuite.


prim
DATE
urm
DATE
urm
DATE
<- ultim
0
urm


prim ->
prec
urm
prec
urm
prec
urm
ultim ->
prec
urm

Stiva (Recapitulare)

- **O *stivă* este o listă simplu înlănţuită gestionată conform principiului LIFO (Last In First Out).**

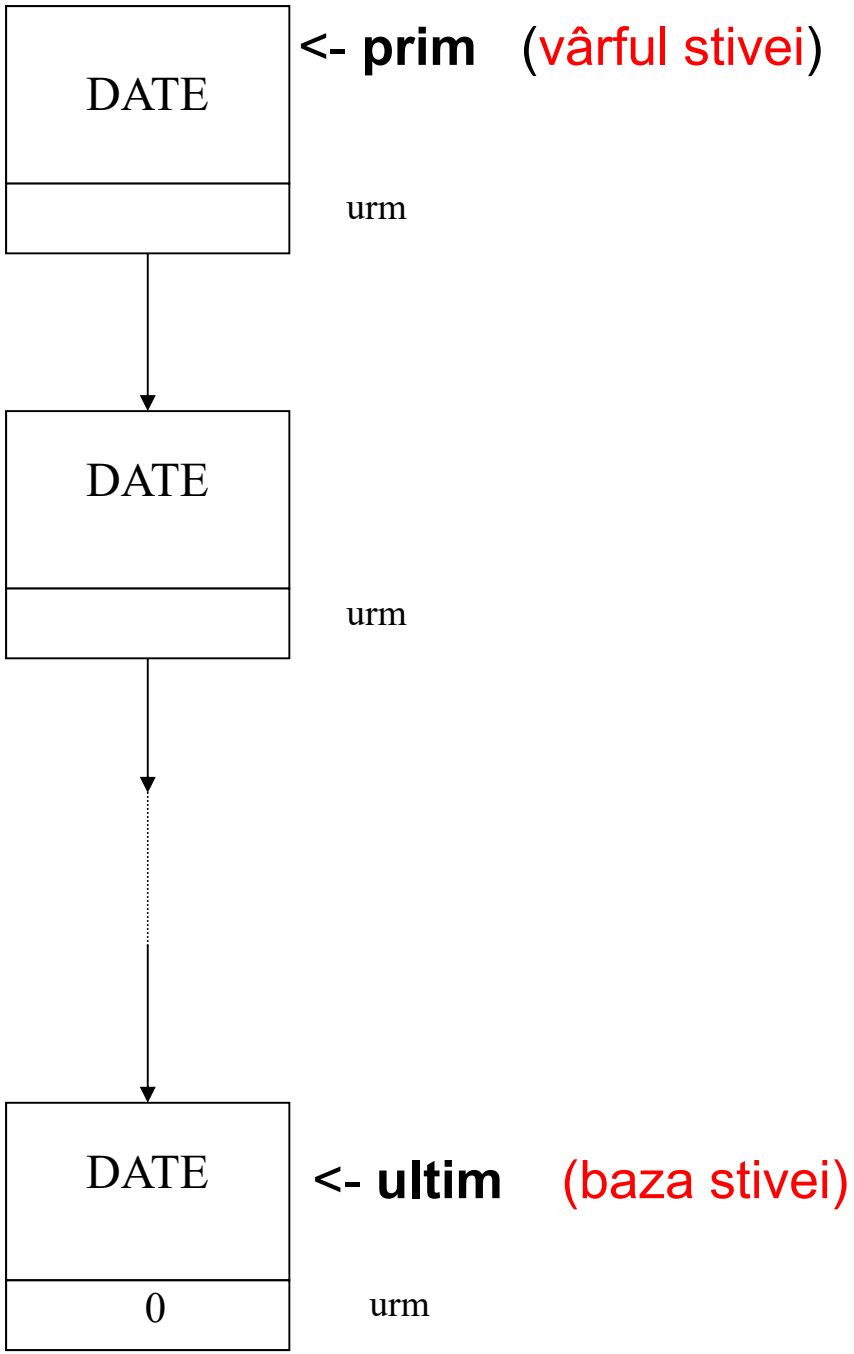
- Conform acestui principiu, ultimul nod pus în **stivă** este primul nod care este scos din stivă. **Stiva**, ca şi lista obişnuită, are două capete:
  - **baza stivei**
  - **vârful stivei**

Asupra unei stive se definesc câteva operaţii, dintre care cele mai importante sunt:

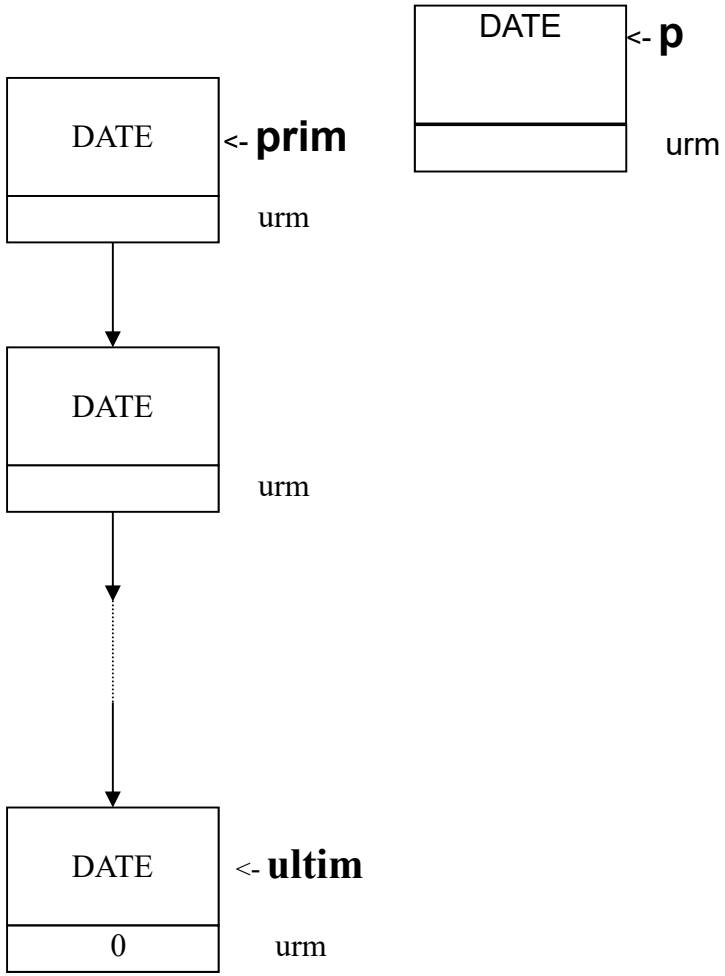
1. pune un element pe stivă (**PUSH**);

2. scoate un element din stivă (**POP**);

3. şterge (videază) stiva (**CLEAR**).

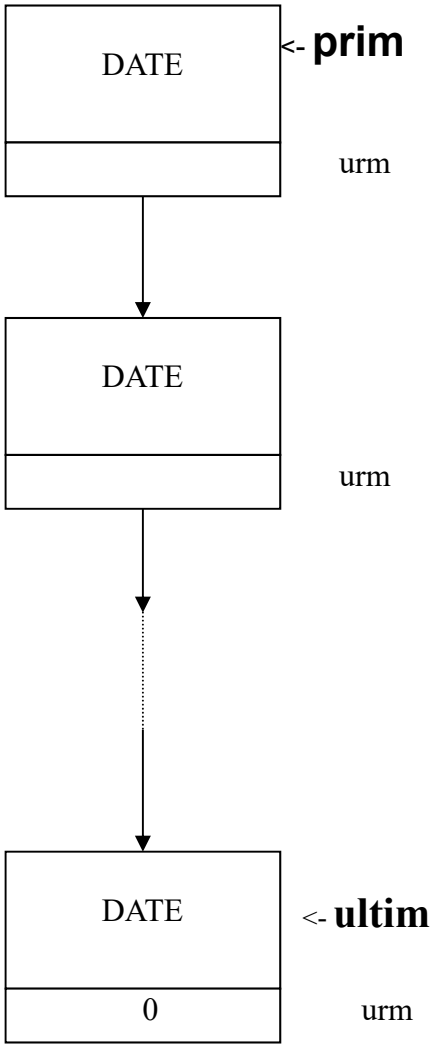

DATE
<- prim (vârful stivei)
urm
DATE
urm
DATE
<- ultim (baza stivei)
0
urm

PUSH (inserarea înaintea primului nod)




```
…
p->urm = prim;
prim=p;
….
```

POP (stergerea primului nod)




```
...

p = prim;
prim = p -> urm;

elibnod(p);
...
```

Coada (Recapitulare)

- Un alt principiu de gestionare a listelor simplu înlănţuite este principiul **FIFO** (**First In First Out**). Conform acestui principiu, primul element introdus în listă este şi primul care este scos din listă.

- Despre o listă gestionată în acest fel se spune că formează o **coadă**. Cele doua capete ale listei simplu înlănţuite care implementează o coadă sunt şi capetele cozii.

Asupra cozilor (ca şi asupra stivelor) se definesc trei operaţii:

a. pune un element în coadă;

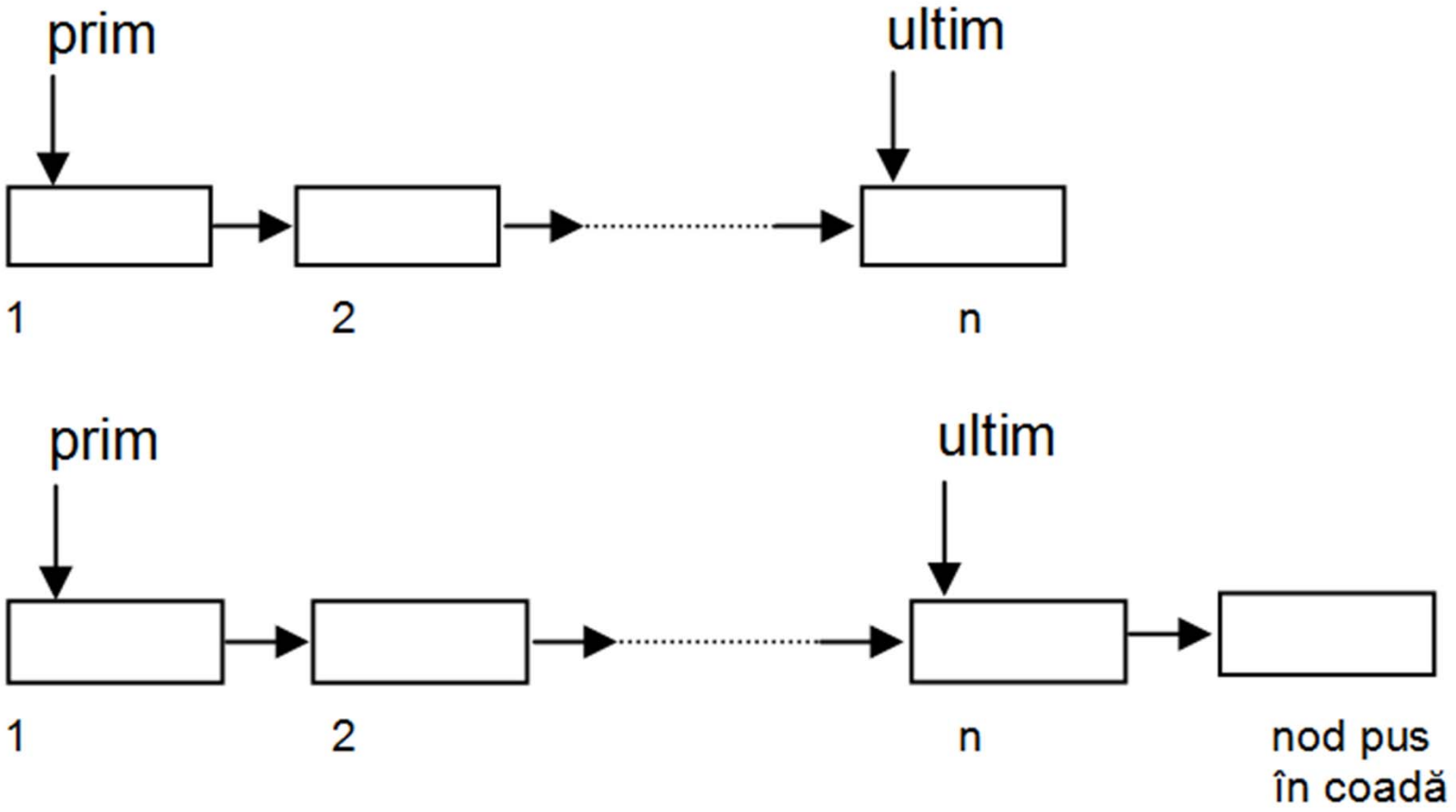
b. scoate un element din coadă;

c. ştergerea (vidarea) unei cozi.

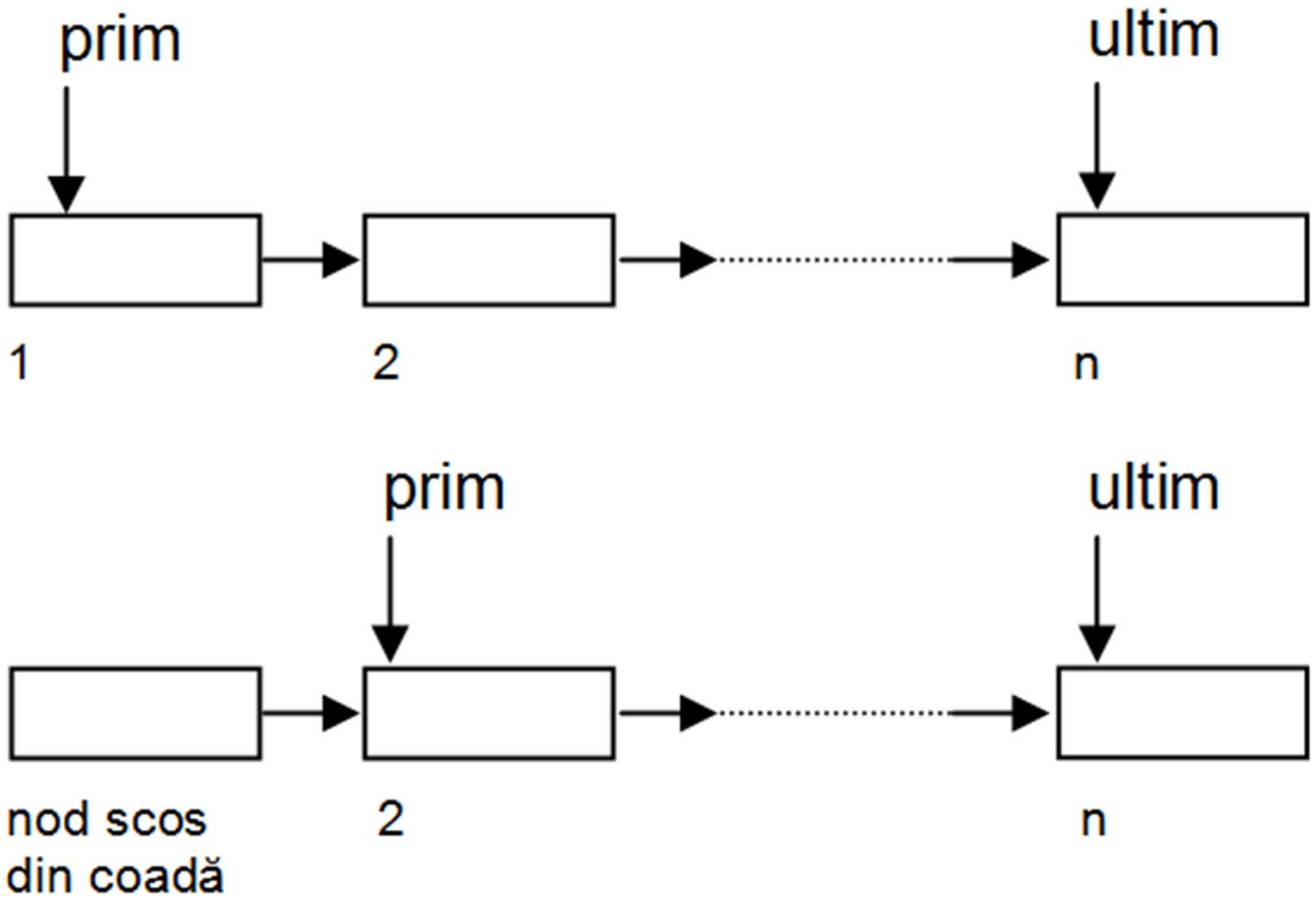
Pentru a respecta principiul FIFO, vom pune un element în coadă folosind funcţia **adauga** şi vom scoate un element din coadă folosind funcţia **spn**.

Deci, la un capăt al cozii se pun elementele în coadă, iar la celalalt capăt se scot elementele din coadă.

Operaţia de punere în coadă:



Operaţia de scoatere din coadă



Lista circulară simplu înlănţuită

Lista simplu înlănţuită conţine:
- un nod care nu are succesor (pointează ultim);
- un nod care nu are predecesor (pointează prim).

Se ştie că într-o listă obişnuită: **ultim -> urm = 0**;

Dacă facem **ultim -> urm = prim**, atunci rezultă o **listă circulară simplu înlănţuită**.

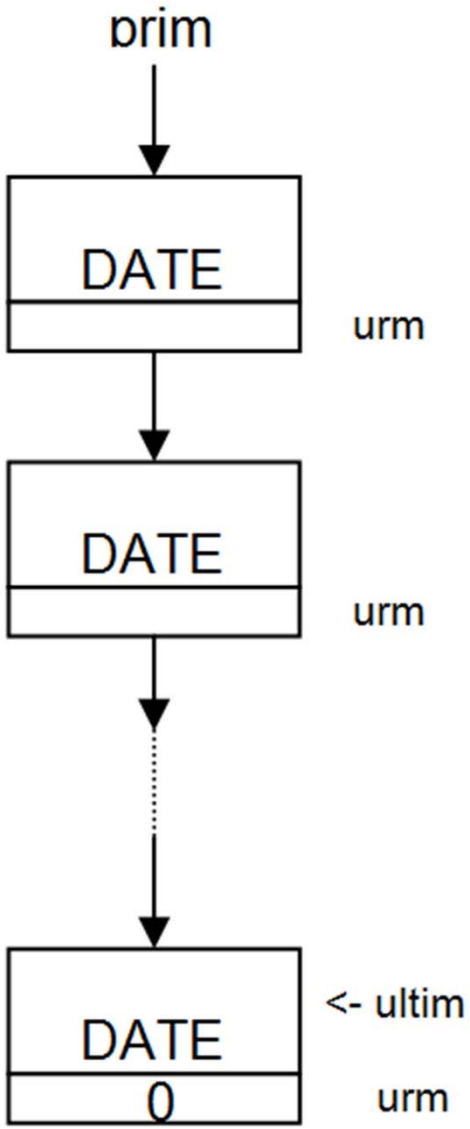
Observaţii:

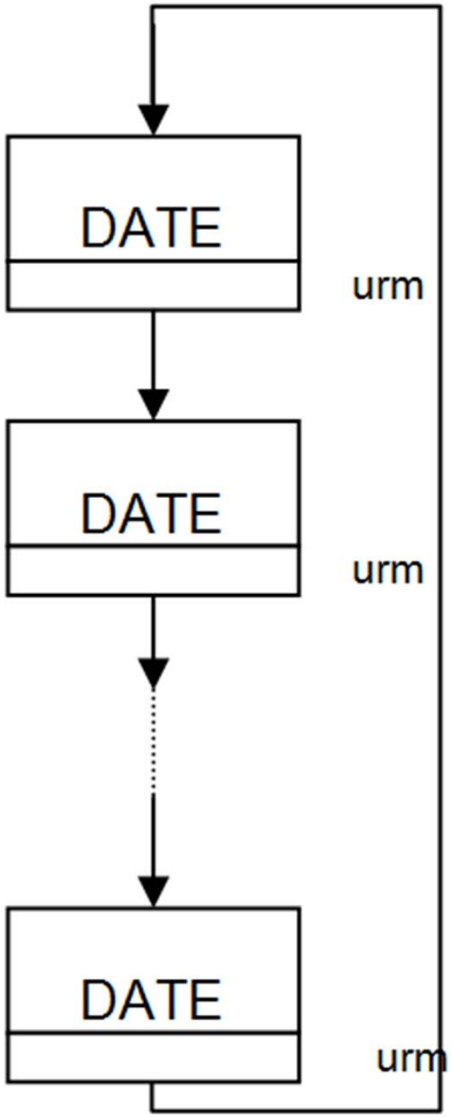
- Într-o listă circulară toate nodurile sunt echivalente.
- Fiecare nod are un succesor şi un predecesor.
- Într-o astfel de listă nu avem capete, deci nu sunt necesare variabilele prim şi ultim.

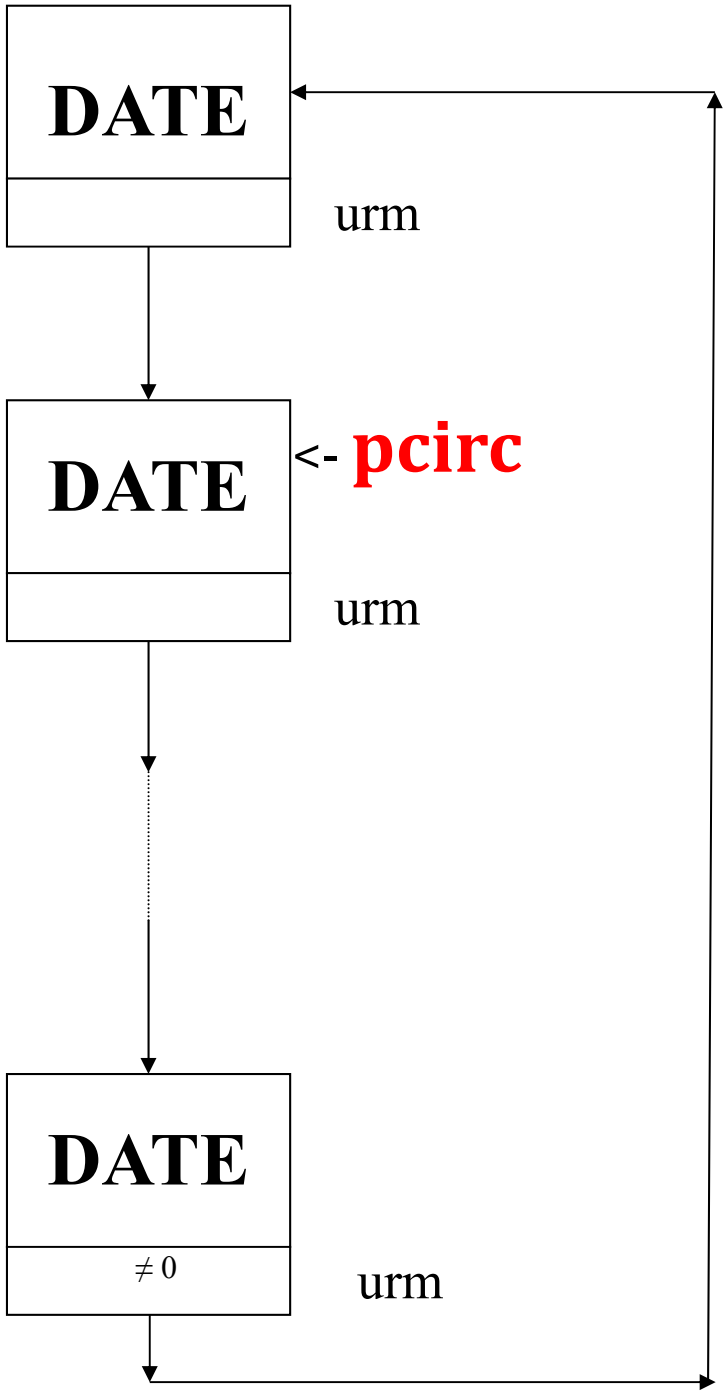
Pentru gestiunea nodurilor, se foloseşte o variabilă globală care pointează spre un nod oarecare al listei *pcirc*:

NOD *pcirc;

unde NOD este tipul comun al nodurilor listei.


prim
DATE
urm
DATE
urm
DATE
0
<- ultim
urm


DATE
urm
DATE
urm
DATE
urm


DATE
urm
DATE
<- pcirc
urm
DATE
≠ 0
urm

Listele circulare au o serie de aplicaţii:

- operaţii cu numere întregi cu un număr mare de cifre;
- operaţii asupra polinoamelor de una sau mai multe variabile;
- alocarea dinamică a memoriei.

Observaţii:

- Funcţiile **malloc** şi **free** utilizează o zonă de memorie numită heap organizată ca o listă circulară.

- Când apelăm **malloc** se parcurg nodurile listei până când se găseşte o zonă de memorie de dimensiune cel puţin egală cu cea cerută de apel. Dacă zona liberă este mare, se divide şi partea neutilizată se înlănţuie cu celelalte blocuri ale listei.

- Dacă nu se găseşte o zonă de memorie corespunzătoare, se încearcă obţinerea ei printr-un apel la sistemul de operare.

Observaţii:

- Funcţia **free** eliberează o zonă de memorie care se înlănţuie la lista circulară.

- Dacă două noduri aparţin de două zone de memorie care se învecinează, atunci ele se vor concatena formând o singură zonă liberă de memorie.

- Se utilizează **free** pentru a elibera o zonă de memorie imediat ce nu mai este nevoie de datele din ea.

Operaţiile uzuale asupra unei liste circulare sunt:

1. Crearea unei liste circulare.
2. Accesul la un nod al unei liste circulare.
3. Inserarea unui nod într-o listă circulară.
4. Ştergerea unui nod dintr-o listă circulară.
5. Ştergerea unei liste circulare.

Crearea unei liste circulare

Se utilizează funcţiile de tipul **incnod** şi **elibnod**

Paşii sunt următorii:

1) La început lista circulară este vidă (**pcirc** = 0).

2) Se rezervă zonă de memorie (cu **malloc**) în memoria heap pentru nodul curent.

3) Există date de încărcat?
- NU → se revine din funcţie (se face şi **elibnod**);
- DA → se încarcă nodul curent cu datele curente (**incnod(p)**) şi se trece la punctul 4;

4) Dacă lista:

• era vidă:

```
pcirc = p;
pcirc -> urm = p;
```

• nu era vidă:

```
p -> urm = pcirc -> urm;
pcirc -> urm = p;
```

Nodul curent se inserează după cel spre care pointează pcirc.

5) Pointerul pcirc pointează spre noul nod inserat

```
pcirc = p;
```

6) Se reia procesul de la punctul 2.

Accesul la un nod al unei liste circulare

Definim o funcţie **acces** care caută un nod după o cheie numerică şi returnează una din valorile:

- pointerul spre nodul căutat;
- **0** dacă nu există un astfel de nod.

Considerăm declarat tipul structură:

```
typedef struct nod
        {
                declaratii;
                int cheie;      /*practic orice tip*/
                declaratii;
                struct nod *urm;
        } NOD;
```

O posibilă soluţie:

```
NOD *acces(int c)
/* cauta nodul de cheie c   (de tip integer)
- returneaza pointerul  spre nodul respectiv sau 0 daca nu
exista un astfel de nod */
{ extern NOD *pcirc;
  NOD *p;
  p=pcirc;
  if (p==0) return 0; /* lista vida */
  do
   {
     if (p->cheie == c) return p;
     p = p->urm;
   } while (p!=pcirc);
 return 0;
}
```

+ Accesul la un nod prin numărare!

...

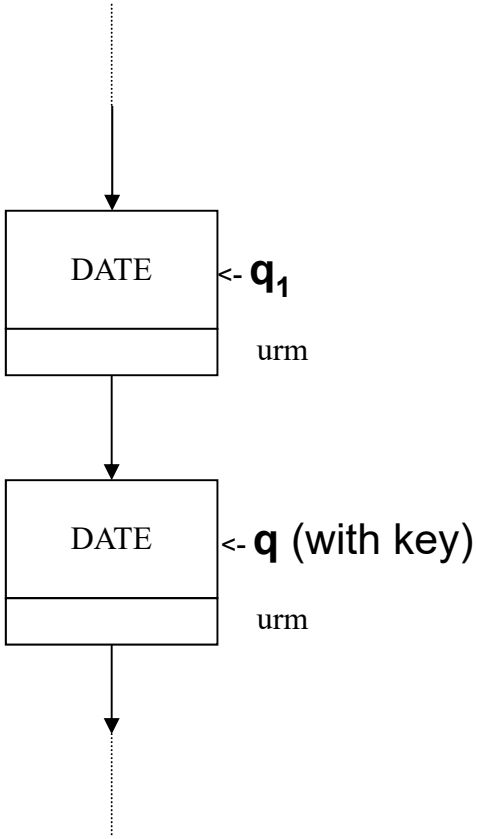
Inserarea unui nod într-o listă circulară

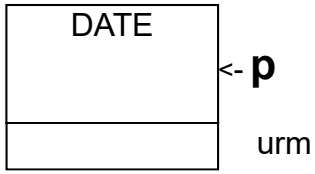
Cazuri (pentru care se va prezenta schema logică):

1. Inserarea înaintea unui nod precizat printr-o cheie

2. Inserarea după un nod precizat printr-o cheie

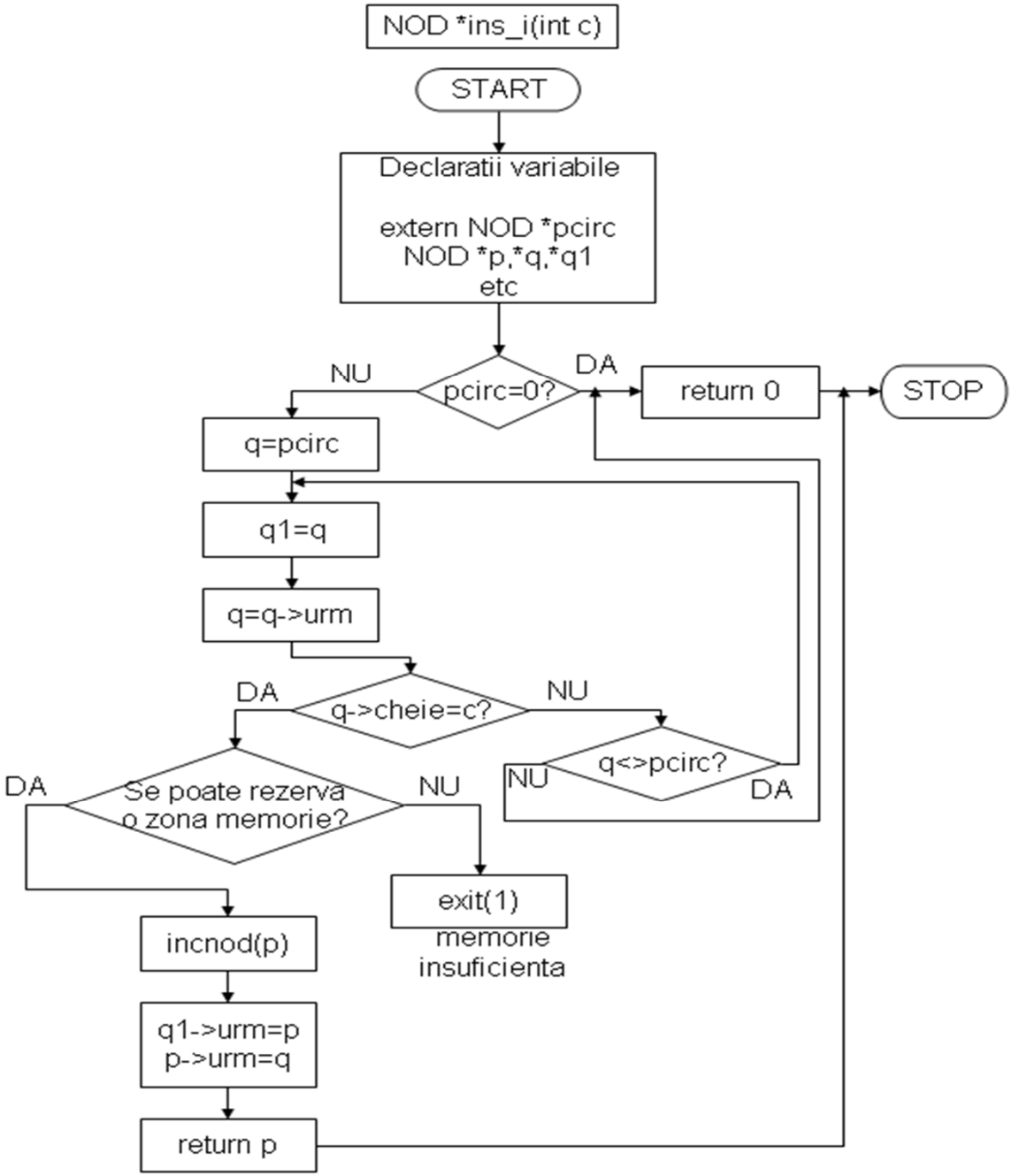
Inserarea înaintea unui nod precizat printr-o cheie



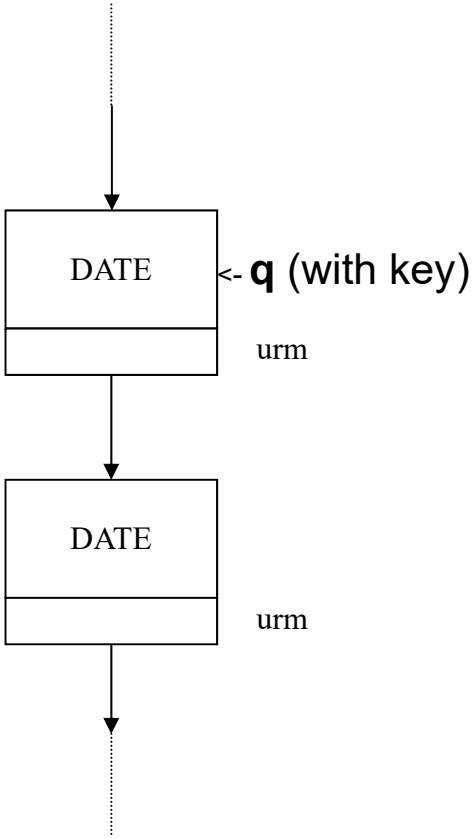


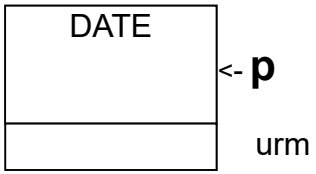

```
…
p->urm=q;
q1->urm=p;
…
```

Inserarea înaintea unui nod precizat printr-o cheie



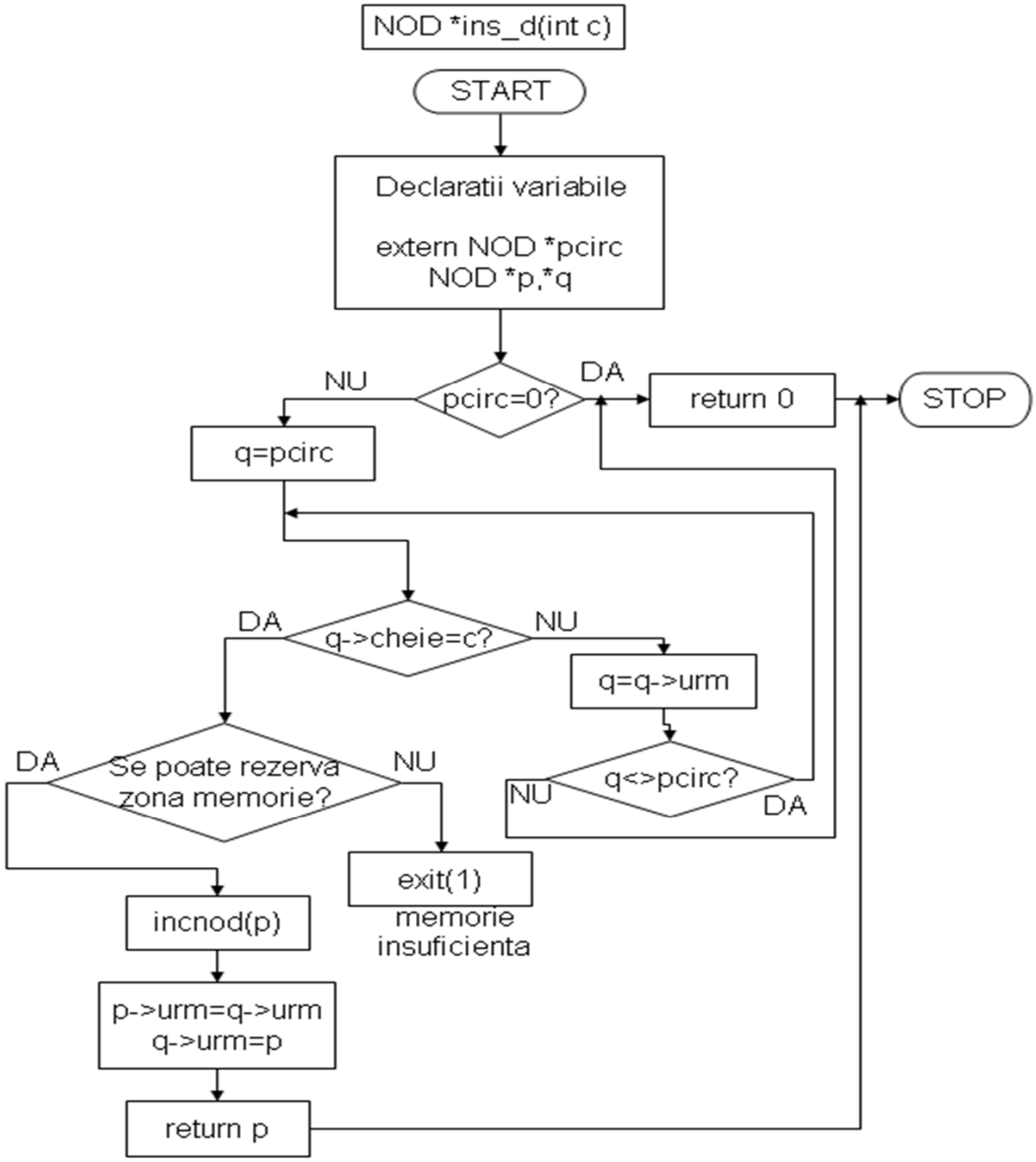
Inserarea după un nod precizat printr-o cheie






```
…
p->urm=q->urm;
q->urm=p;
…
```

Inserarea după un nod precizat printr-o cheie



+ Inserarea unui nod prin numărare!

...

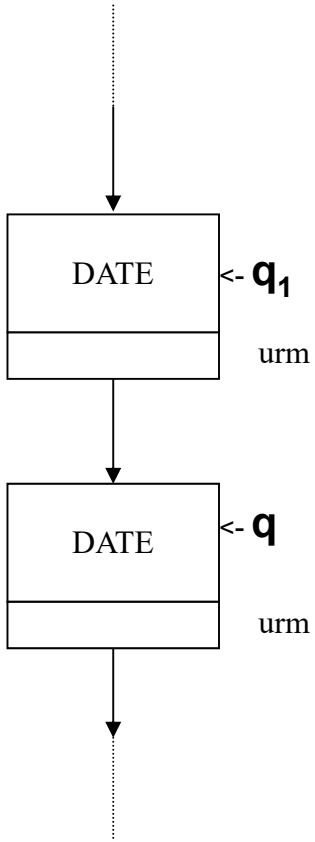
Ştergerea unui nod dintr-o listă circulară

Pentru o funcţie care şterge dintr-o listă circulară un nod precizat printr-o cheie, se pot conveni următoarele:

- dacă **pcirc** pointează spre nodul care se şterge, după ştergere **pcirc** va pointa spre nodul precedent.

- dacă lista devine vidă, atunci **pcirc=0**.

Ștergerea unui nod precizat printr-o cheie



(nodul unde a fost găsită cheia)

```
...

q1 ->urm = q -> urm;

elibnod(q);
...
```

Observaţii:

- Funcţia de ştergere se aseamănă cu

  NOD *ins_i(int c)

- Ştergerea unui nod se mai poate face şi prin precizarea poziţiei acestuia faţă de un reper (pe care îl vom nota cu **rep**).

```
typedef struct individ
{
    char *nume_cavaler;
    struct individ *urm;
}SOLDAT;
```

Exemplu:

- Funcția **eliminare** șterge din lista circulară indicată de pointerul **rep** cel de-al n-lea nod pornind de la **rep** și apoi întoarce noul pointer către lista circulară, care este precedentul celui șters.

- Pointerul **rep** (reper) joaca rolul lui **pcirc**.

```
SOLDAT *eliminare(SOLDAT *rep, int n)
{
      SOLDAT *q,*p;
      int k=1;

      if(rep==0)
         return(0); /*cazul in care lista era deja vida*/

…
```

```
…
    if(rep!=rep->urm) /*lista are cel putin doua noduri*/
    {
       p=rep;
       while(k < n)
                {
                 q=p;
                 p=p->urm;
                 k++;
                }
       q->urm=p->urm;
       elibnod(p);
       rep=q;
    }
…
```

```
...
   else  /*a ramas cazul in care lista era dintr-un singur nod*/
         {
           elibnod(rep);
           rep=0;
         }

   return(rep);
}
```

Ştergerea unei liste circulare

```
void sterge_l_c()
{
   extern NOD *pcirc;
   NOD *p,*p1;
   if ((p=pcirc)==0) return; /*lista vida*/
   do
      {
         p1=p;
         p=p->urm;
         elibnod(p1);
      } while (p!=pcirc);
   pcirc=0;
}
```

Problema lui Josephus

(posibilă problemă de examen)

O cetate asediată este apărată de un număr de cavaleri (soldaţi). Se pune problema de a alege pe unul dintre ei pentru a pleca după ajutor. Soldaţii sunt dispuşi în cerc (ca într-o listă circulară). Se alege un număr ***n***, după care pornind de la unul din soldaţi, se elimină succesiv tot al ***n***-lea soldat până când mai rămâne unul singur care pleacă în misiune.

Paşii algoritmului sunt:

1) Se porneşte cu nodul imediat următor celui care conţine pointerul spre şirul dat şi se elimină din listă al **n**-lea nod care urmează.

2) Se execută pasul 1 continuând cu nodul imediat următor celui şters până când lista se reduce la un singur nod.

Programul rezolvă următoarele cerinţe:

- crează lista circulară spre care pointează un pointer **pcirc**.
- citeşte un număr **n** indicat în formularea problemei.
- elimină nodurile listei conform paşilor 1 şi 2.
- afişează cuvântul din nodul rămas în listă.