Curs SDA (PC2)
Curs 6
Liste (recapitulare)

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;

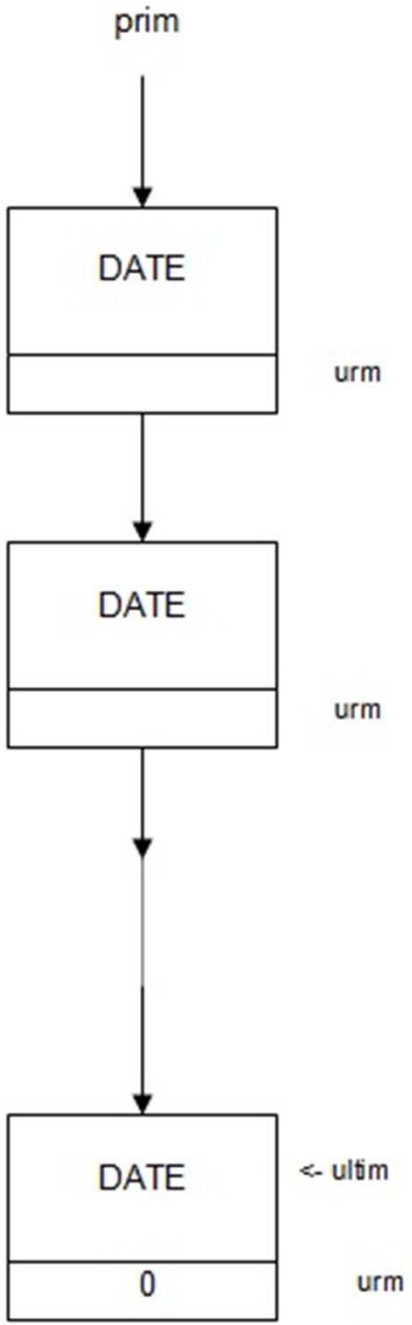
e) ştergerea unei liste înlănţuite.

Lista simplu înlănţuită
(Recapitulare)

Între nodurile listei simplu înlănţuite avem o singură relaţie de ordonare. Există un singur nod care nu mai are succesor şi un singur nod care nu mai are predecesor. Aceste noduri formează capetele listei.

Vom utiliza doi pointeri spre cele două capete pe care îi notăm cu:

- **prim** - pointerul spre nodul care nu are predecesor
- **ultim** - pointerul spre nodul care nu are succesor.



Pointerul **urm** defineşte relaţia de succesor pentru nodurile listei. Pentru fiecare nod, el are ca valoare adresa nodului următor din listă. Excepţie face nodul spre care pointează variabila **ultim** (pentru care **urm** ia valoarea zero).

Observaţii:

- Se consideră tipul utilizator:

```
typedef struct nod
        { declarații;
          struct nod *urm;
        } NOD;
```

- Uzual se încarcă datele printr-o funcție pe care o vom denumi **incnod** care încarcă datele curente într-un nod de tip NOD.

- Funcţia incnod returnează:
  - 1 la încărcarea corectă a datelor în nodul curent
  - -1 când nu mai sunt de încărcat date
  - 0 la apariția unei erori (ex: memorie insuficientă)

- Funcţia incnod are prototipul:

  int incnod(NOD *p)

- O altă funcție necesară este cea care eliberează zona de memorie rezervată unui nod:

  void elibnod(NOD *p)

2.2. Accesul la un nod într-o listă simplu înlănţuită (recapitulare)

- Putem avea acces la nodurile unei liste simplu înlănţuite începând cu nodul spre care pointează variabila globală **prim** şi trecând apoi pe rând de la un nod la altul, folosind pointerul **urm**.
- O altă metodă, mai bună, este aceea de a avea o dată, componentă a nodurilor, care să aibă valori diferite, pentru noduri diferite. În acest caz se poate defini accesul la nodul din listă pentru care data respectivă are o valoare fixată. O astfel de dată se numeşte **cheie** şi este de un tip oarecare (uzual se foloseşte char şi int). Funcţia returnează pointerul spre nodul căutat sau zero în cazul în care lista nu conţine un nod a cărui cheie să aibă valoarea indicată de parametrul ei.

Se consideră tipul utilizator:

```
typedef struct nod
        { char *cuvant;  /*aceasta este cheia */
          int frecventa;
          struct nod *urm;
        } NOD;
```

```
NOD *cncs(char *c)      /*caută nod dupa cheia c*/
{
 extern NOD *prim;
 NOD *p;
 for(p = prim; p; p = p->urm)
      if(strcmp(p->cuvant,c) == 0)
        return p; /* s-a gasit un nod cu cheia c */
 return 0; /* nu exista nici un nod cu cheia c */
}
```

2.3. Inserarea unui nod într-o listă simplu înlănţuită (recapitulare)

Într-o listă simplu înlănţuită se pot face inserări de noduri în diferite poziţii:

a) inserare înaintea primului nod;

b) inserarea înaintea unui nod precizat printr-o cheie;

c) inserarea după un nod precizat printr-o cheie;

d) inserarea după ultimul nod al listei.

Exemplu (recapitulare):

- inserarea după ultimul nod al listei

```
NOD *adauga()
/* - adauga un nod la o lista simplu înlănţuita;
   - returneaza pointerul spre nodul adaugat sau
   zero daca nu s-a realizat adaugarea. */
{
 extern NOD *prim, *ultim;
 NOD *p;
 int n;

/* se rezerva zona de memorie pentru un nod si se incarca
datele din zona respectiva */
 n = sizeof(NOD);
…
```

```
…
if(((p = (NOD *)malloc(n)) != 0) && (incnod(p) == 1))
    {
     if(prim == 0)  /* lista este vida */
           prim = ultim = p;
     else
           {
            ultim -> urm=p;  /* succesorul nodului spre care
                                pointeaza ultim devine nodul
                                spre care pointeaza p */

            ultim = p; /* acesta devine nodul spre
                          care pointeaza p */
           }
     p -> urm=0;
     return p;
    }
…
```

```
…
if(p == 0)  /* nu s-a reusit alocarea de memorie */
    {
     printf("memorie insuficienta\n");
     getch(); /* pauza pentru vizualizare */
     exit(1);
    }
 elibnod(p);
 return 0;
}
```

Supliment:

```
int incnod(NOD *p)
/*  incarca datele curente in nodul
    spre care pointeaza p  */
{
 if((p -> cuvant = citcuv()) == 0)  return -1;
 p -> frecventa = 1;
 return 1;
}
```

```
char *citcuv()
/* - citeste un cuvant si-l pastreaza in memoria heap;
   - returneaza pointerul spre cuvantul respectiv sau
   zero la sfarsit de fisier. */
{
 int c, i;
 char t[255];
 char *p;

 /*salt peste caracterele care nu sunt litere */
 while((c=getchar()) < 'A' || (c > 'Z' && c < 'a') || c > 'z')
        if(c == EOF)
                return 0; /* s-a tastat EOF */
...
```

```
....
/* se citeste cuvantul si se pastreaza in t */

i=0;

do
  {
   t[i++] = c;
  } while(((c=getchar()) >= 'A' && c <= 'Z' || c >= 'a') && c <= 'z');

if(c == EOF)
   return 0;
t[i++] = '\0';
....
```

```
 ....
 /* se pastreaza cuvantul in memoria heap */
 if((p = (char *)malloc(i)) == 0)
         {
          printf("memorie insuficienta\n");
          getch(); /* pauza pentru vizualizare */
          exit(1);
         }
 strcpy(p,t);
 return p;
}
```

Atenție! – Aici p nu este un pointer către NOD, ci către un șir de caractere

Supliment:

```
void elibnod(NOD *p)
/* elibereaza zonele din memoria heap
ocupate de nodul spre care pointeaza p */
{
 free(p -> cuvant);
 free(p);
}
```

2.4. Ştergerea unui nod dintr-o listă simplu înlănţuită (recapitulare)

Sunt avute în vedere următoarele cazuri:

a) ştergerea primului nod al listei simplu înlănţuite;

b) ştergerea unui nod precizat printr-o cheie;

c) ştergerea ultimului nod al listei simplu înlănţuite.

Ștergerea primului nod

```
void spn()
{
extern NOD *prim, *ultim;
NOD *p;
if (prim == 0) return;
p = prim;
prim = prim -> urm;
elibnod(p);
if (prim == 0)  /* lista este vida */
        ultim = 0;
}
```

Supliment:

```
void sun() /* sterge ultimul nod din lista */
{
 extern NOD *prim, *ultim;
 NOD *q, *q1;
 q1 = 0;
 q = prim;
 if(q == 0)
   return; /* lista vida */
 while(q != ultim) /* se parcurge lista pana
                      se ajunge la ultimul nod al ei */
   {
    q1 = q;
    q = q->urm;
   }
…
```

```
....

 if(q == prim) /* lista contine un singur nod care se sterge
                    - lista devine vida */
    prim = ultim = 0;

 else /* - nodul spre care pointeaza q1 are ca succesor nodul
spre care pointeaza q si acesta este ultimul nod al listei*/
    {
     q1 -> urm=0;
     ultim = q1;
    }
 elibnod(q);
}
```

2.5. Ștergerea unei liste simplu înlănțuite

```
void sterge_l()
{
        extern NOD *prim, *ultim;
        NOD *p;
        while(prim)
                {
                        p = prim;
                        prim = prim -> urm;
                        elibnod (p);
                }
        ultim = 0;
}
```

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**).

- Primele două operaţii se realizează în ***vârful stivei***. Astfel, dacă se scoate un element din stivă, atunci acesta este cel din vârful stivei şi în continuare, cel pus anterior lui pe stivă ajunge în vârful stivei.

- Dacă un element intră pe stivă, atunci acesta se pune în vârful stivei şi în continuare el desemnează vârful stivei.

Pentru a implementa o stivă printr-o listă simplu înlănţuită va trebui să identificăm baza şi vârful stivei cu capetele listei simplu înlănţuite. Există două posibilităţi:

a. **nodul spre care pointează variabila prim este baza stivei, iar nodul spre care pointează variabila ultim este vârful stivei;**

b. **nodul spre care pointează variabila prim este vârful stivei, iar nodul spre care pointează variabila ultim este baza stivei.**

Observaţii:

- În cazul **a**, funcţiile PUSH şi POP se identifică prin funcţiile **adauga** şi respectiv **sun**, definite în şedinţa anterioară de laborator. Dacă funcţia **adauga** este eficientă, în schimb funcţia **sun** nu este eficientă.

- În cazul **b**, funcţiile PUSH şi POP se identifică prin funcţiile **iniprim** şi respectiv **spn**, ce vor fi definite în această lucrare de laborator. În acest caz, ambele funcţii sunt eficiente. De aceea, se recomandă implementarea stivei printr-o listă simplu înlănţuită conform cazului **b** indicat mai sus.

Exemplu de PUSH (cazul b)

```
TNOD *iniprim()  /* - PUSH -   insereaza nodul curent inaintea primului nod al listei */
{
 extern NOD *prim, *ultim;
 NOD *p;
 int n;

 n = sizeof(NOD);
…
```

```
…
if(((p = (NOD *)malloc(n)) != 0) && (incnod(p) == 1))
   {
    if(prim == 0)
          {
           prim = ultim = p;
           p -> urm = 0;
          }
    else
          {
           p -> urm = prim;
           prim = p;
          }
    return p;
   }
…
```

```
...

if(p == 0)
   {
    printf("memorie insuficienta\n");
    getch(); /* pauza pentru vizualizare */
    exit(1);
   }
 elibnod(p);
 return 0;
} /* sfarsit iniprim */
```

Caz particular:

Problema cu trenuri de la Laborator

```
typedef struct nod
        {
         long cvag;
         long cmarfa;
         int exp;
         int dest;
         struct nod *urm;
        } NOD;
```

Caz particular: Problema cu trenuri de la Laborator

```
int incnod(NOD *p)
/* incarca un nod cu datele despre vagoane */
{
 char t[255];
 char er[ ] = "S-a tastat EOF in pozitie rea\n";
 long cod;
 int icod;

 /* citeste cod vagon */
 for( ; ; )
 {
 printf("cod vagon: ");
 if(gets(t) == 0)
            return -1; /* nu mai sunt date */
 if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
            break;
 printf("cod vagon eronat\n");
 }
 p -> cvag = cod;
 ….
```

```
....
/* citeste cod marfa */
for( ; ; )
{
printf("cod marfa: ");
if(gets(t) == 0)
   {
    printf(er);
    return 0;
   }
if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
   break;
printf("cod marfa eronat\n");
}
p -> cmarfa = cod;

/* citeste cod expeditor */
...
/* citeste cod destinatar */
...
return 1;
} /* sfarsit incnod */
```

```
void elibnod(NOD *p)
 /* elibereaza nodul spre care pointeaza p */
{
    free(p);
} /* sfarsit elibnod */
```

Exemplu de POP (cazul b)

```
void spn() /* POP - sterge primul nod din lista */
{
 extern NOD *prim, *ultim;
 NOD *p;

 if(prim == 0)
        return ;
 p = prim;
 prim = prim -> urm;
 elibnod(p);
 if(prim == 0)
        ultim = 0;
} /* sfarsit spn */
```

Observație:

În anumite cazuri o stivă se poate implementa cu ajutorul unui tablou de pointeri (pe care îl denumim **tpnod**).

Considerăm:
tpnod[0] - baza stivei
tpnod[v] - vârful stivei

unde:
int v; /* este o variabila globala care poate avea valoarea maxima MAX */

Se pot defini două funcții asupra acestei stive:
empty() - returnează 1 dacă stiva este vidă și 0 în caz contrar
full() - returnează 1 dacă stiva este plină și 0 în caz contrar.

În circumstanțele prezentate pe slide-ul anterior cele două funcții pot lua doar valori logice de adevăr.

De aceea putem utiliza tipul:

typedef enum {false, true} Boolean;

…

```
Boolean empty()
        {
          extern int v;
          return (v == 0);
        }
…

Boolean full()
        {
          extern int v;
          return (v >= MAX);
        }
```

Coada

- 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;

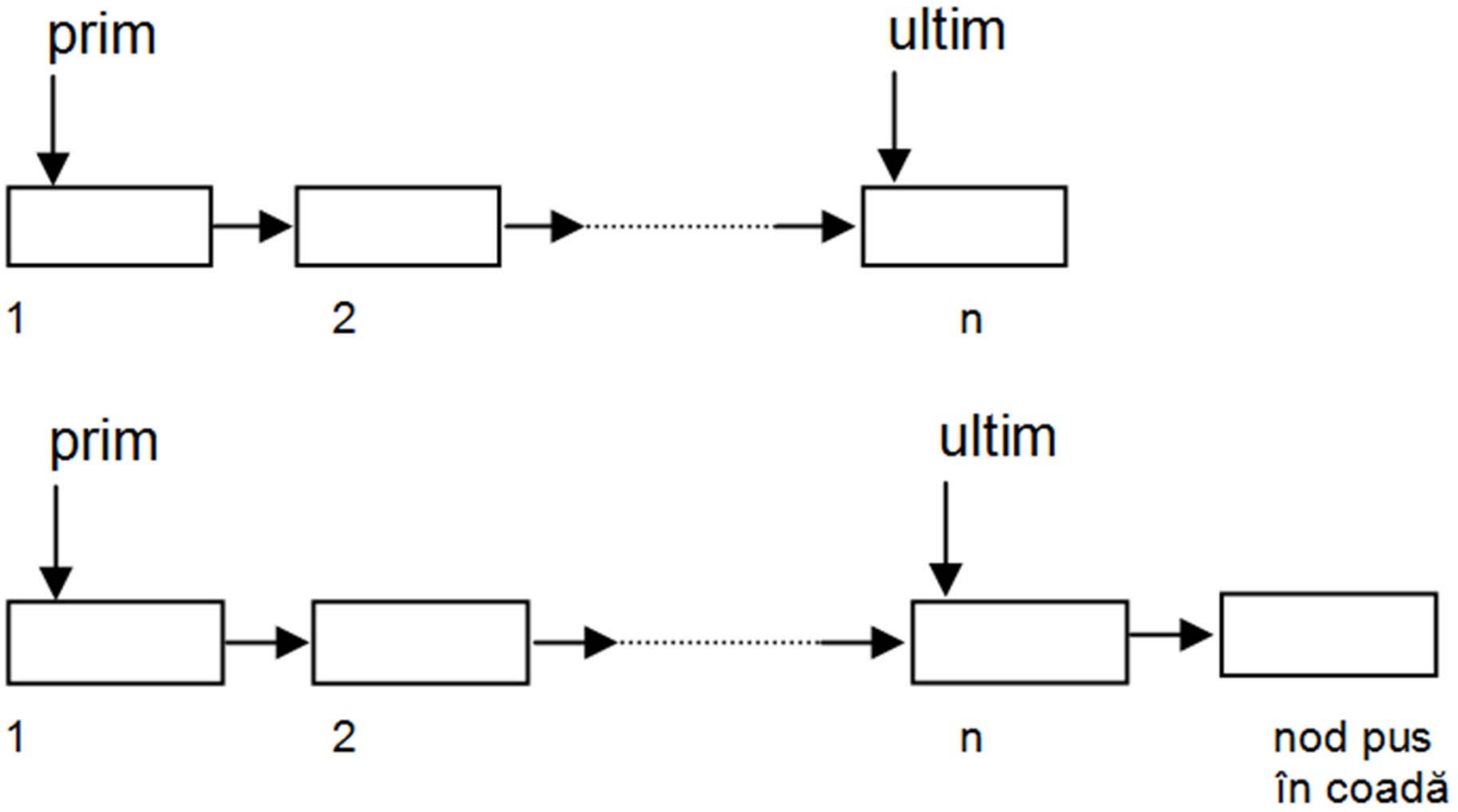
b. scoate un element din coadă;

c. ştergerea (vidarea) unei cozi.

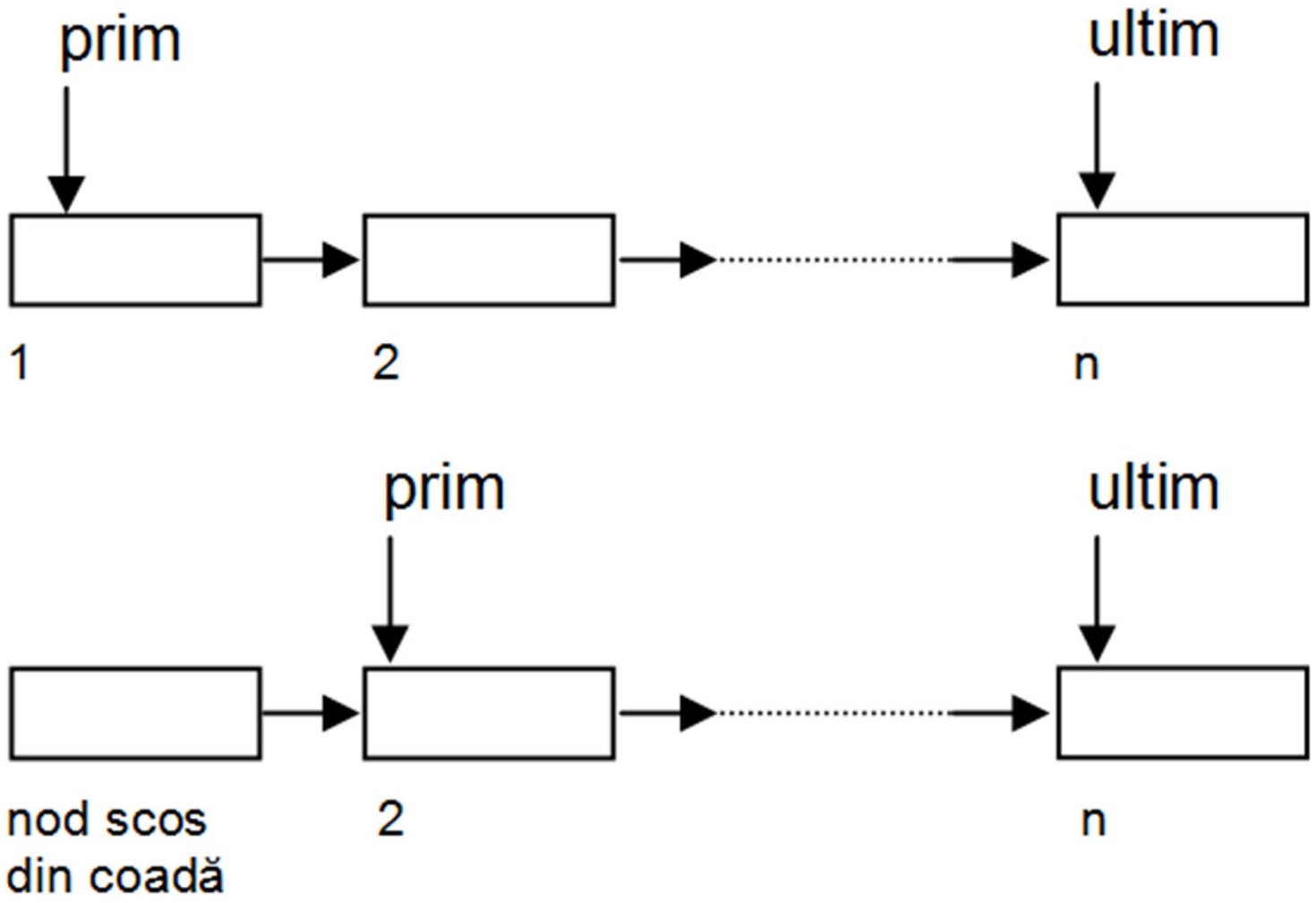
Pentru a respecta principiul FIFO, vom pune un element în coadă folosind funcţia **adauga** şi vom scoate un element din coadă folosind funcţia **spn**.

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**.