



PROIECTAREA ALGORITMILOR

Lect. univ. dr. Adrian Runceanu

Curs 4

Liste simplu înlănțuite

Conținutul cursului

- 4. Structuri implementate dinamic:
 - 4.1. Stiva
 - 4.2. Coadă
 - 4.3. Lista simplu înlănțuită
 - 4.4. Lista dublu înlănțuită

4.2. Coada

- O structură de date care respectă regula **FIFO** se numește **coadă**.

FIFO (First In First Out)
Primul-intrat-primul-iesit

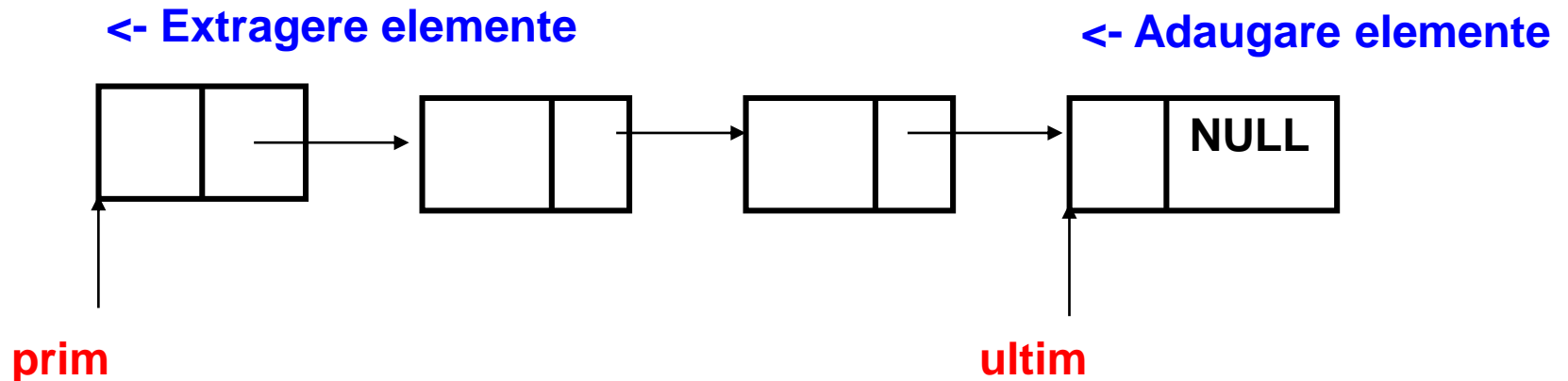
- Pentru a respecta această regulă, vom efectua adăugarea în coada listei, iar extragerea din capătul cozii.
- O coadă este un tip de date cu operații caracteristice comportării **primul venit / primul servit**.
- Elementul care poate fi servit este cel care a sosit în coadă cel mai devreme.
- Dacă acesta a fost scos din coadă, atunci următorul element care a sosit cel mai devreme devine elementul care va putea fi servit în continuare.

4.2. Coada

- În cazul implementării dinamice, un element al cozii conține două tipuri de informații:
 - **informația utilă (*inf*)**
 - și **informația de legătură (*leg*)** - care este de fapt un pointer ce conține adresa următorului element din coadă.
- Ultimul element al cozii nu mai are succesori, deci informația sa de legătură are valoarea **NULL**.
- De asemenea, având în vedere că structura de coadă are două capete (**primul** și **ultimul**) sunt necesare două variabile de tip referință care să indice primul element din coadă (**prim**) și ultimul element din coadă (**ultim**).
- În cazul în care coada este vidă, **prim** și **ultim** au valoarea **NULL**.

4.2. Coada

- Astfel, o reprezentare grafică a structurii de tip coadă poate arăta în felul următor:



4.2. Coada

Pentru implementarea dinamică a cozii, sunt necesare următoarele declarații:

```
typedef struct tnod
{
    tip inf; // informatia propriu-zisa
    struct tnod *leg; // informatia de legatura
} TNOD;
TNOD *prim,*ultim; /* adresa primului, respectiv a ultimului
element din coada */
```

4.2. Coada

Având aceste date, operațiile cu coada folosind alocarea dinamică a memoriei, în limbajul C++ se pot implementa astfel:

1) Inițializarea cozii:

```
void Creare_vida (TNOD *prim, TNOD *ultim)
{
    prim=ultim=NULL;
}
```


4.2. Coada

2) **Adăugarea unui element** x în coadă:

Având în vedere definiția structurii de coadă, se poate face observația că *adăugarea unui element se efectuează numai la sfârșitul cozii, după ultimul element.*

Dacă coada este vidă, atunci la inserarea unei valori se creează de fapt primul element al cozii.

În acest caz, **prim** și **ultim** indică același element (au aceeași valoare).

4.2. Coada

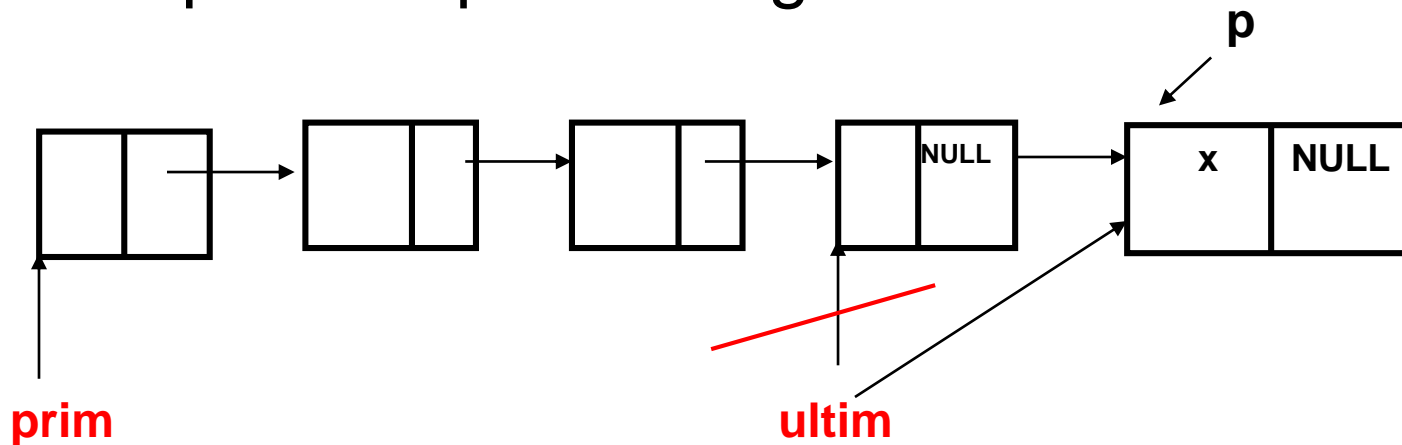
Astfel etapele acestei operații sunt:

- Alocarea unei zone de memorie pentru noul element - **(a)**
- Se introduce informația utilă pentru acest element - **(b)**
- Informația de legatură a elementului creat se completează cu **NULL** - **(c)**
- Se reinițializează informația de legatură a elementului indicat de **ultim** cu adresa noului element - **(d)**
- Se modifică valoarea pointerului **ultim** dându-i ca valoare adresa noului element introdus - **(e)**

```
void adaug(tip x, TNOD *ultim, TNOD *prim)
{
    TNOD *p;                // pointer de lucru
    if (prim==NULL) // se verifica daca primul element din coada
        este creat. Daca nu este, se creeaza separat de celelalte
    {
        prim=ultim=new TNOD;
        prim->inf=x;
        prim->leg=NULL;
    }
    else
    {
        p=new TNOD;          // (a)
        p->inf=x;             // (b)
        p->leg=NULL;         // (c)
        ultim->leg=p;        // (d)
        ultim=p;            // (e)
    }
}
```

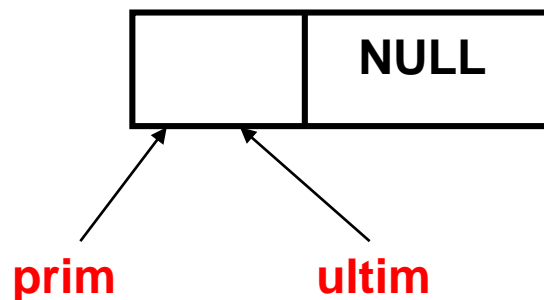
4.2. Coada

- Adăugarea unui element într-o coadă nevidă se poate reprezenta grafic astfel:



4.2. Coada

- Dacă coada era inițial vidă, după adăugarea unui element arată astfel:



4.2. Coada

3) Extragerea unui element din coadă (a primului element)

- Operația de extragere se poate face doar asupra primului element din coadă.
- Bineînțeles, putem extrage un element numai dintr-o coadă care nu este vidă.
- Se observă că primul element din coadă fiind eliminat, va fi înlocuit de cel care inițial era al doilea.

4.2. Coada

Deci, se pot deduce următoarele etape ale extragerii din coadă:

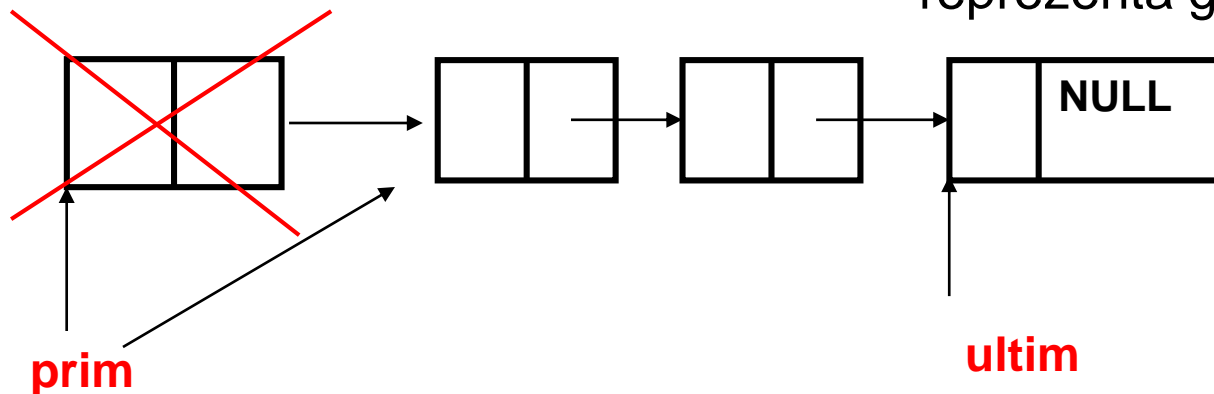
- Se reține într-o variabilă (***q**) informația utilă din primul element al cozii - **(a)**
- Se reține într-un pointer de lucru adresa acestui element (să numim pointerul de lucru **p**) - **(b)**
- Se actualizează valoarea pointerului **prim** cu adresa următorului element din coadă - **(c)**
- Se eliberează zona de memorie ce fusese alocată inițial primului element din coadă (acum indicat de pointerul **p**) - **(d)**

```

void extrag(TNOD *prim, tip *q)
{
    TNOD *p;           // pointer de lucru
    *q=prim->inf;      // (a)
    p=prim;           // (b)
    prim=prim->leg;    // (c)
    delete p;         // (d)
}

```

Operația de ștergere, se poate reprezenta grafic astfel:



4) Verificarea dacă coada este vidă

```
int cvida (TNOD *prim)
{
    return (prim==NULL);
}
```

5) Numărarea elementelor din coadă

```
int cardinal (TNOD *prim)
{
    int m=0;           // contorizează elementele cozii
    TNOD *p;          // pointer de lucru
    p=prim;
    while(p != NULL)
    {
        m++;
        p=p->leg;
    }
    return m;
}
```

Conținutul cursului

- 4. Structuri implementate dinamic:**
 - 4.1. Stiva**
 - 4.2. Coadă**
 - 4.3. Lista simplu înlănțuită**
 - 4.4. Lista dublu înlănțuită**

4.3. Lista simplu înlănțuită

- Lista simplu înlănțuită este structura de reprezentare a informațiilor cea mai cunoscută și implicit cea mai utilizată atunci când ne referim la alocarea dinamică a memoriei.
- *O listă simplu înlănțuită este formată dintr-o colecție de elemente de același tip.*

4.3. Lista simplu înlănțuită

- Fiecare element conține în afară de **elementul propriu-zis** și **o legătură** care ne spune unde putem găsi un alt element din mulțime.
- Elementele listei vor fi numite și *noduri*.
- Ideea este că fiecare nod dintr-o listă simplu înlănțuită conține informații despre cum putem localiza un alt nod dintr-o mulțime de noduri.
- *Accesul imediat la fiecare nod din mulțime nu este necesar pentru că fiecare nod conduce la un altul.*

4.3. Lista simplu înlănțuită

- Acest tip de element se numeste **NOD**.



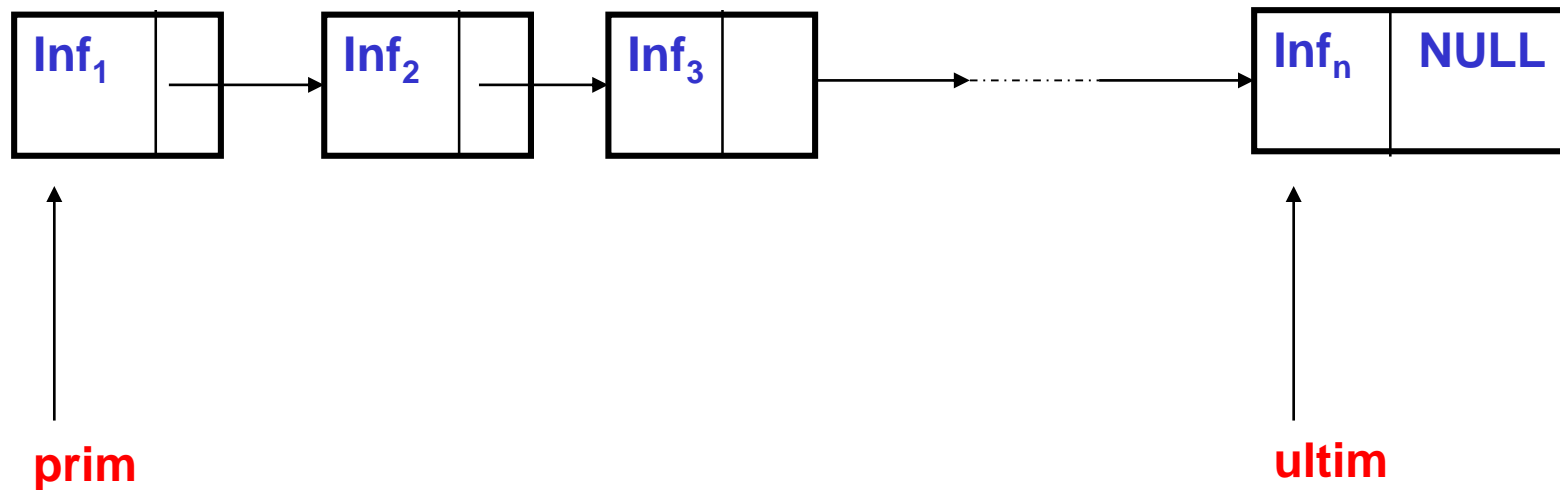
- Prin **inf** înțelegem informația atașată elementului respectiv, și care poate fi de orice tip de date cunoscut de către limbajul C++, iar prin **leg** înțelegem un câmp de tip referință care va conține adresa următorului element din mulțimea de elemente.

4.3. Lista simplu înlănțuită

- Pentru a putea construi și a folosi cât mai eficient o listă simplu înlănțuită este necesară o variabilă de tip referință care să indice primul element din listă.
- Convenim să notăm cu *prim* – adresa primului nod.
- Uneori, întâlnim aplicații care necesită și folosirea adresei ultimului nod, notată cu *ultim*.

4.3. Lista simplu înlănțuită

O formă grafică sugestivă pentru o listă simplu înlănțuită ar fi următoarea:



Avantaje

Listele simplu înlănțuite reprezintă o utilizare foarte importantă a alocării dinamice a memoriei deoarece:

1. Sunt mai flexibile decât stiva și coada (care restricționează operațiile de adăugare, acces și ștergere a elementelor conform definițiilor lor)
2. Se recomandă folosirea listelor simplu înlănțuite în rezolvarea problemelor specifice vectorilor, deoarece se utilizează eficient memoria care poate fi alocată sau eliberată în funcție de cerințele programatorului
3. Anumite genuri de probleme (cum ar fi operațiile cu matrici rare; respectiv polinoame rare) își găsesc o rezolvare mai rapidă, eficientă și utilă folosind listele

4.3. Lista simplu înlănțuită

Declarațiile necesare lucrului cu o listă simplu înlănțuită sunt:

```
typedef struct tnod  
{  
    tip inf;                // informatia propriu-zisa  
    struct tnod *leg;      // informatia de legatura  
} LISTA;  
LISTA *prim,*ultim; /* adresa primului, respectiv a  
ultimului element din lista */
```

4.3. Lista simplu înlănțuită

Cu listele simplu înlănțuite se pot face următoarele operații:

1) Inițializarea listei (crearea unei liste vide)

```
void init(TNOD *prim)
{
    prim=NULL;
}
```

4.3. Lista simplu înlănțuită

2) Adăugarea unui element în listă

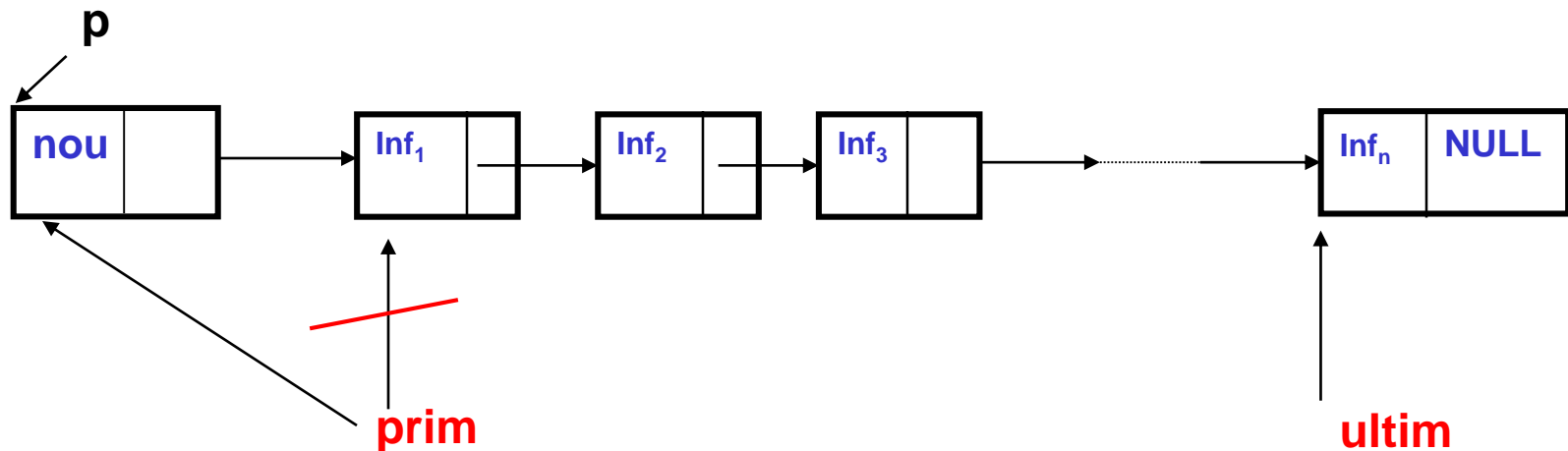
Aceasta operație se poate face în trei moduri, în funcție de locul unde se inserează elementul în listă.

Astfel putem avea:

- a) Inserarea unui element la începutul listei
- b) Inserarea unui element la sfârșitul listei
- c) Inserarea unui element în interiorul listei

4.3. Lista simplu înlănțuită

- a) **Adăugarea unui element la începutul listei** se poate reprezenta grafic astfel:



Etapele adăugării unui element la începutul listei:

- alocarea zonei de memorie necesare noului element (Se folosește un pointer de lucru **p**) - **(a)**
- completarea informației utile pentru noul element (notată cu *nou*) - **(b)**
- completarea informației de legătură cu adresa conținută în variabila **prim** (ținând cont că acesta va deveni primul element al listei și, conform definiției acesteia, trebuie să conțină adresa elementului următor din listă, deci cel care era primul înainte de a face inserarea) - **(c)**
- Actualizarea variabilei referință **prim** cu adresa elementului creat, care în acest moment devine primul element al listei - **(d)**

4.3. Lista simplu înlănțuită

```
void adaug_la_inceput( tip x, LISTA *prim )  
    // x reprezinta informatia ce se adauga la începutul listei  
{  
    LISTA *p;           // pointerul de lucru  
    p=new LISTA;       // (a)  
    p->inf=x;          // (b)  
    p->leg=prim;      // (c)  
    prim=p;           // (d)  
}
```

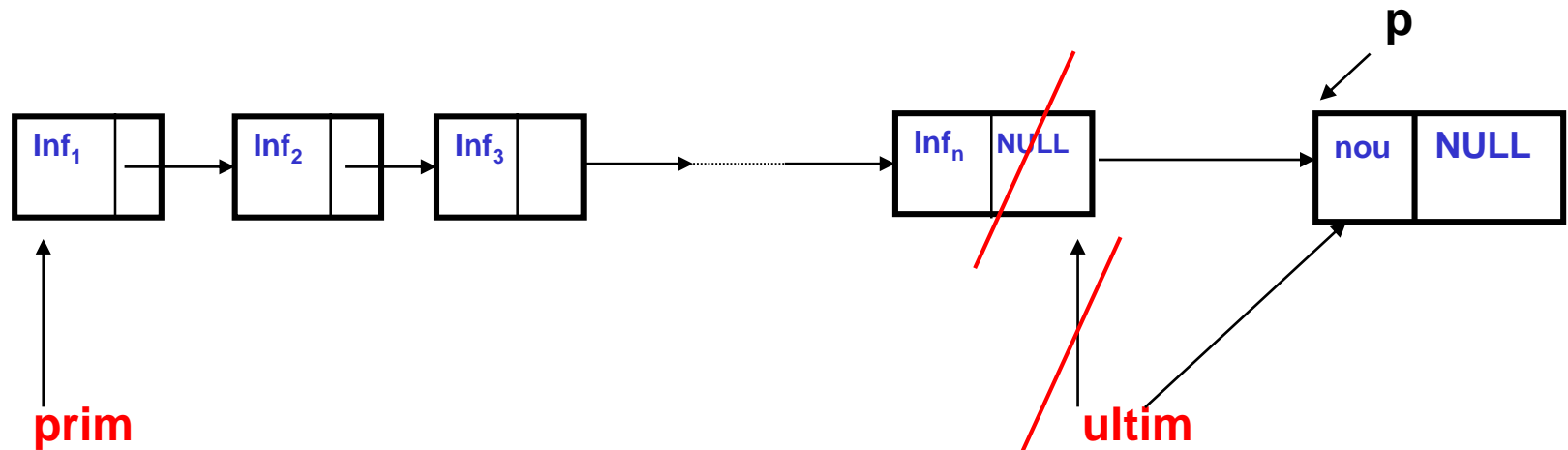
4.3. Lista simplu înlănțuită

b) Adăugarea unui element la sfârșitul listei, presupune inserarea acestuia după ultimul nod al listei.

În acest caz avem nevoie de variabila *ultim* care indică nodul după care se va insera.

4.3. Lista simplu înlănțuită

Descrierea operațiilor necesare se pot deduce și de această dată folosind o reprezentare grafică a listei:



Pașii algoritmului de adăugare a unui element la sfârșitul listei:

- Alocarea zonei de memorie necesară pentru noul element - **(a)**
- Completarea informației de legătură a elementului creat cu *NULL*, deoarece el va deveni ultim element - **(b)**
- Completarea informației utile - **(c)**
- Informația de legatură a celui ce a fost înainte ultimul element al listei își schimbă valoarea cu adresa noului element (îl va indica pe acesta) - **(d)**
- Se actualizează pointerul *ultim* cu adresa nodului adăugat listei - **(e)**

4.3. Lista simplu înlănțuită

```
void adauga_la_sfarsit( tip x, LISTA *ultim )  
    // realizeaza adaugarea valorii x la sfarșitul listei  
{  
    LISTA *p;           // pointer de lucru  
    p=new LISTA;       // (a)  
    p->leg=NULL;       // (b)  
    p->inf=x;          // (c)  
    ultim->leg=p;     // (d)  
    ultim=p;          // (e)  
}
```

c) Adăugarea unui element în interiorul listei

- Această operație se poate face **înaintea** sau **după un element al listei**.
- Cel mai comod se face ***inserarea după un element specificat al listei***.
- Deoarece se realizează o inserare după un nod, acest lucru poate determina o adăugare la sfârșitul listei în cazul în care nodul respectiv este ultimul nod din această listă, operație descrisă anterior în funcția ***adaug_la_sfarsit***.

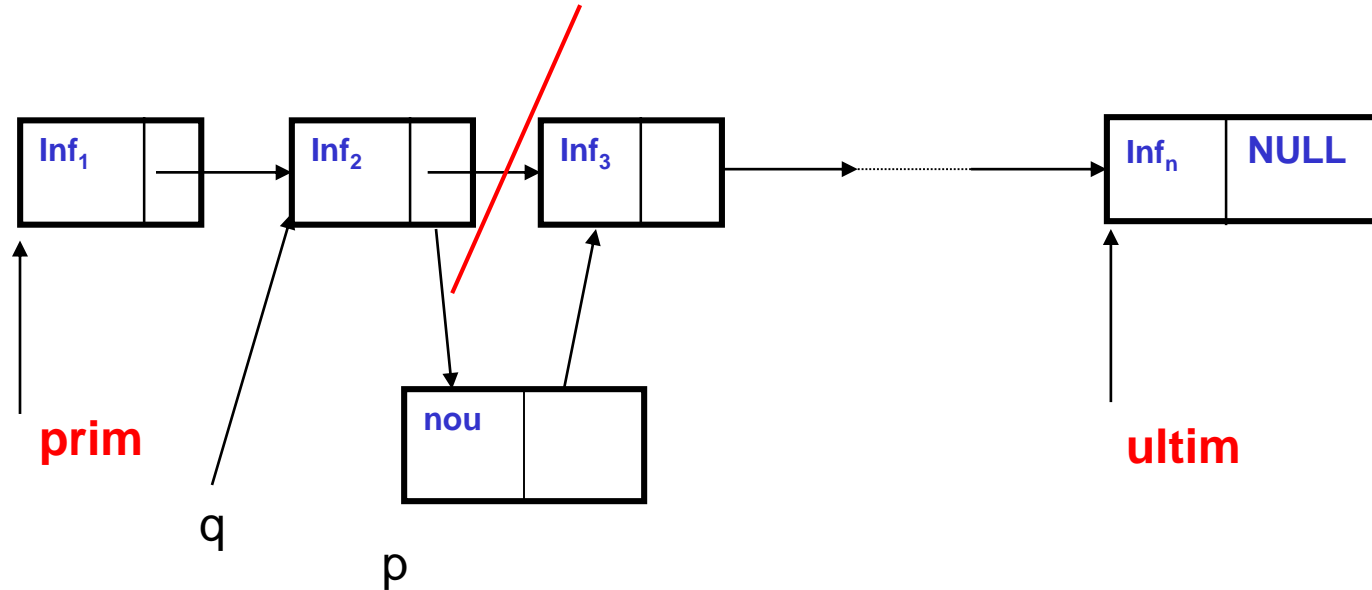
- Sunt necesari doi pointeri de lucru:
 1. **q** – indică nodul după care este făcută inserarea
 2. **p** – pointer de lucru necesar pentru crearea unui nou element
- Presupunem că avem o listă cu cel puțin două elemente, unde după nodul indicat de **q** vom adăuga un nou element cu valoarea informației propriu-zise **x**.

4.3. Lista simplu înlănțuită

Succesiunea logică a etapelor necesare *inserării după un nod* (indicat de **q**) este următoarea:

- alocarea zonei de memorie necesară noului element (folosirea pointerului **p**) - **(a)**
- inițializarea informației utile (cu valoarea notată **nou**) - **(b)**
- inițializarea informației de legătură cu adresa următorului nod (cu adresa reținută în acest moment în variabila **q->leg**) - **(c)**
- actualizarea informației de legatură din nodul după care s-a inserat noul element cu adresa zonei de memorie alocată pentru acesta (**p**) - **(d)**

4.3. Lista simplu înlănțuită



4.3. Lista simplu înlănțuită

```
void adaug_dupa_o_valoare( int x, LISTA *q)
{ // adauga valoarea lui x într-un nod ce urmeaza nodului
  indicat de q în lista
    LISTA *p;           // pointer de lucru
    p=new LISTA;       // (a)
    p->inf=x;          // (b)
    p->leg=q->leg;     // (c)
    q->leg=p;          // (d)
}
```

4.3. Lista simplu înlănțuită

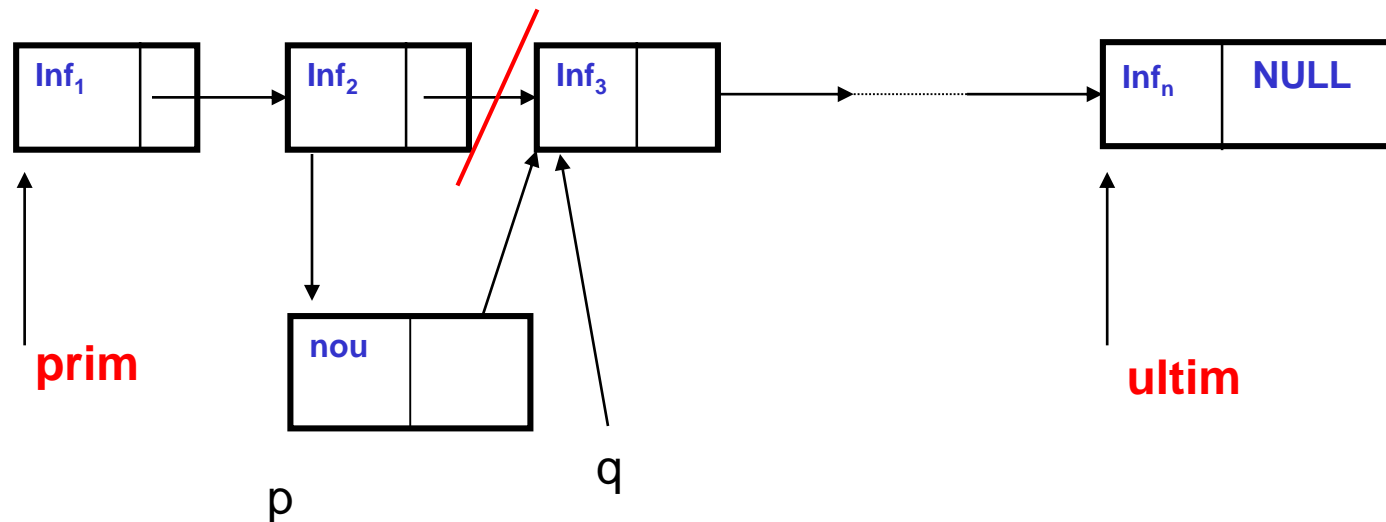
Cealaltă situație de inserare a unui nod în interiorul listei este de a realiza aceasta *înaintea unui nod* indicat de **q**.

Pentru această situație putem folosi următoarea soluție, care este cea mai utilizată, fiind mai rapidă și eficientă și va fi descrisă în următorii pași:

- alocarea unei zone de memorie pentru noul element, dar cu scopul de face inserare după nodul referit de **q** (folosind pe **p**) - **(a)**
- se completează acest nod cu informația ***q** (se realizează o dublură a informației propriu-zise din nodul indicat de **q**) - **(b)**
- se actualizează câmpurile *inf* și *leg* din nodul **q** - **(c,d)**

4.3. Lista simplu înlănțuită

Reprezentarea grafică corespunzătoare este următoarea:



4.3. Lista simplu înlănțuită

```
void adaug_inainte_de_o_valoare( tip x, LISTA *q )  
{  
    LISTA *p;           // pointer de lucru  
    p=new LISTA;       // (a)  
    p->inf=q->inf;      // (b)  
    p->leg=q->leg;      // (b)  
    q->inf=x;           // (c)  
    q->leg=p;           // (d)  
}
```

3. Traversarea (parcurgerea) listei

- *Reprezintă o operație necesară pentru afișarea sau prelucrarea elementelor listei, ce constă în trecerea prin fiecare element al listei o singură dată.*
- Se folosește un pointer de lucru pe care-l voi nota cu **p** și care *va fi inițializat cu valoarea **prim**, iar apoi își va schimba valoarea cu adresa nodului următor (folosind atribuirea $p=p \rightarrow leg$), ceea ce se va repeta până când **p** va avea valoarea **NULL** (ceea ce înseamnă că a fost prelucrat și ultimul nod).*
- La fiecare schimbare a valorii pointerului **p**, se va afișa informația utilă din nodul indicat de **p**.
- Bineînțeles, lista nu trebuie să fie vidă.
- *Parcurgerea unei liste simplu înlanțuite se face secvențial, într-un singur sens, de la primul către ultimul nod.*

4.3. Lista simplu înlănțuită

Afișarea elementelor listei pentru exemplificarea procedurii de parcurgere a listei:

```
void parcurgere ( LISTA *prim )  
{  
    LISTA *p;  
    p=prim;  
    while (p!=NULL)  
        {  
            cout<<p->inf;  
            p=p->leg;  
        }  
}
```

Problemă rezolvată

- a) Sa se creeze o lista liniara simplu inlantuita care sa memoreze urmatoarele informatii despre studentii unei grupe formata din n studenti:
- numele (sir de maxim 20 de caractere)
 - prenumele (sir de maxim 20 de caractere)
 - 5 note intr-un vector cu 5 componente reale
- b) Sa se afiseze numele, prenumele si media fiecarui student.
- c) Sa se scrie o functie care calculeaza si returneaza media grupei.

Problemă rezolvată

```
#include<fstream.h>
fstream f("studenti.in",ios::in);
struct student{
    char nume[20];
    char prenume[20];
    float note[5];
    float media;
};
struct nod{
    student s;
    nod *leg;
};
nod *prim;
```

Sau, se poate introduce o singura structura:

```
struct nod{
    char nume[20];
    char prenume[20];
    float note[5];
    float media;
    nod *leg;
};
```

Problemă rezolvată

```
void adaugf(nod *prim, student x)
{
// adaugarea unui nod intr-o lista simplu inlantuita
  nod *nou=new nod;
  nou->s=x;
  nou->leg=prim;
  prim=nou;
}
```

Problemă rezolvată

```
void citire()
{
    int n; student x;
    f >> n;
    for(int i=1; i<=n; i++)
    {
        f >> x.nume >> x.prenume >> x.note[0] >> x.note[1];
        f >> x.note[2] >> x.note[3] >> x.note[4];
        x.media = (x.note[0] + x.note[1] + x.note[2] + x.note[3] + x.note[4]) / 5;
        adaugf(prim, x);
    }
}
```


Problemă rezolvată

```
void afis(nod *prim)
{
    nod *p=prim;
    while(p)
    {
        cout<<p->s.nume<<" " <<p->s.prenume<<" " <<p->
s.media<<endl;
        p=p->leg;
    }
}
```

Problemă rezolvată

```
float mediagen(nod *prim)
{
    float s=0;
    int n=0;
    nod *p=prim;
    while(p)
    {
        s+=p->s.media;
        n++;
        p=p->leg;
    }
    return s/n;
}
```

Problemă rezolvată

```
int main()
{
    citire();
    afis(prim);
    cout<<mediagen(prim);
}
```

Întrebări?