4

PROIECTAREA ALGORITMILOR

FI-AIA-1-Proiectarea Algoritmilor-2022/2023

Lect. univ. dr. Adrian Runceanu

Curs 4

Liste simplu înlănţuite

Conţinutul cursului

4. Structuri implementate dinamic:

- **4.1. Stiva**
- **4.2. Coada**
- **4.3. Lista simplu înlănţuită**
- **4.4. Lista dublu înlănţuită**

4.2. Coada

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

FIFO (***F****irst* ***I****n* ***F****irst* ***O****ut*)
Primul-intrat-primul-iesit

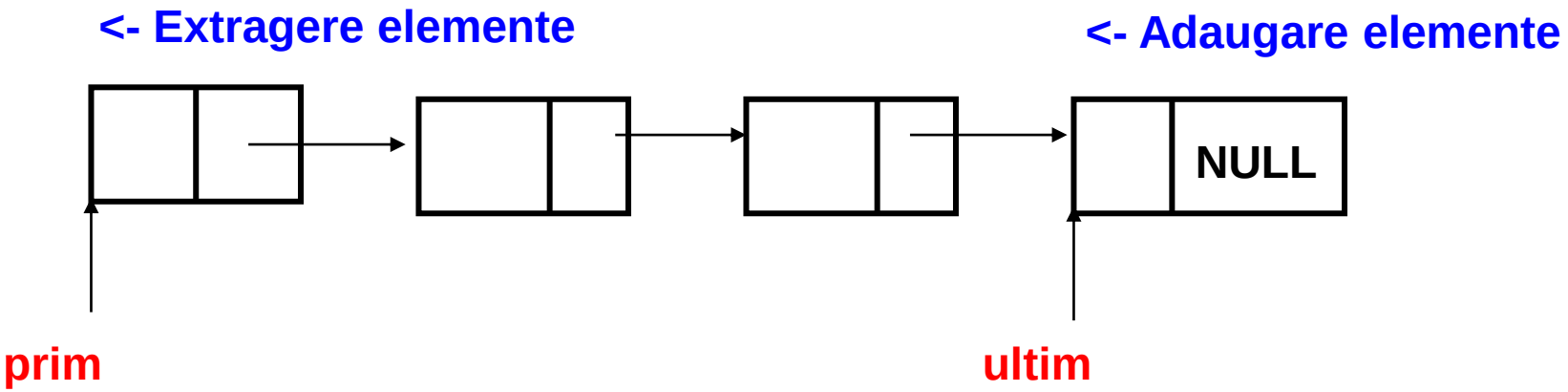
- Pentru a respecta această regulă, vom efectua adăugarea în coada listei, iar extragerea din capătul cozii.
- O coadă este un tip de date cu operaţii caracteristice comportării ***primul venit / primul servit***.
- Elementul care poate fi servit este cel care a sosit în coadă cel mai devreme.
- Dacă 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 conţine două tipuri de informaţii:
  - **informaţia utilă (*inf*)**
  - şi **informaţia de legătură (*leg*) -** care este de fapt un pointer ce conţine adresa următorului element din coadă.
- Ultimul element al cozii nu mai are succesor, deci informaţia sa de legătură are valoarea ***NULL***.
- De asemenea, având în vedere că structura de coadă are două capete (**primul** şi **ultimul**) sunt necesare două variabile de tip referinţă care să indice primul element din coadă (**prim**) şi ultimul element din coadă (**ultim**).
- În cazul în care coada este vidă, **prim** şi **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 declaraţii:

```
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, operaţiile cu coada folosind alocarea dinamică a memoriei, în limbajul C++ se pot implementa astfel:

1) **Iniţializarea cozii**:

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

4.2. Coada

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

Având în vedere definiţia structurii de coadă, se poate face observaţia că *adăugarea unui element se efectuează numai la sfârşitul 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** şi **ultim** indică acelaşi element (au aceeaşi valoare).

4.2. Coada

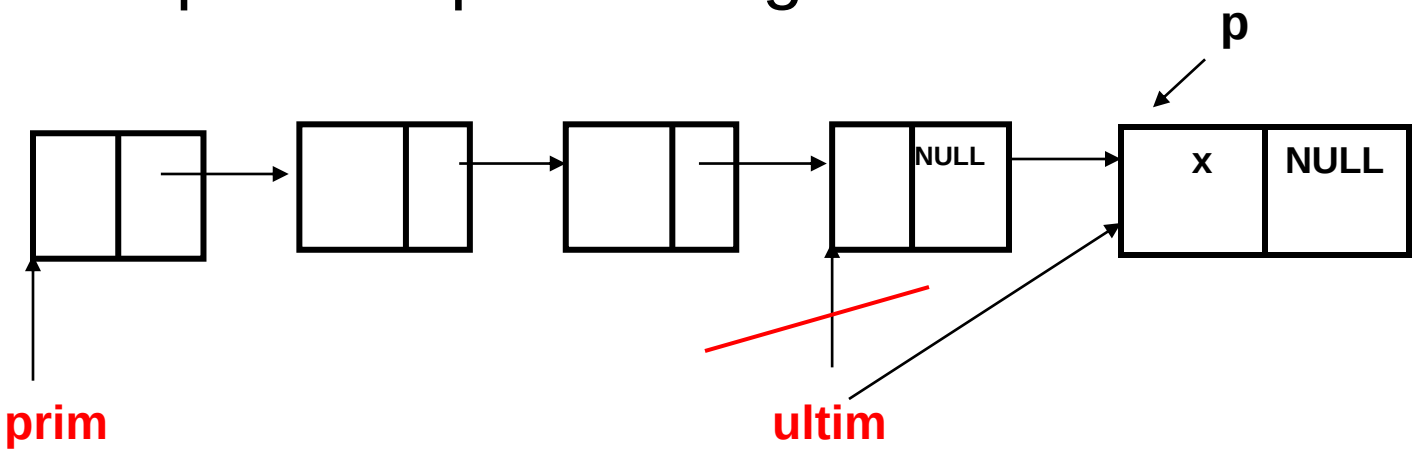
Astfel etapele acestei operaţii sunt:

- Alocarea unei zone de memorie pentru noul element -(**a**)
- Se introduce informaţia utilă pentru acest element - (**b**)
- Informaţia de legatură a elementului creat se completează cu ***NULL*** - (**c**)
- Se reiniţializează informaţia 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)
    }
}
```

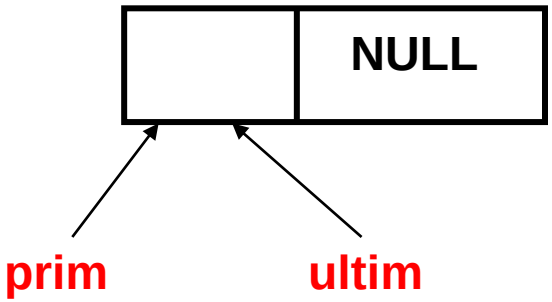
4.2. Coada

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



4.2. Coada

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



4.2. Coada

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

- Operaţia de extragere se poate face doar asupra primului element din coadă.
- Bineînţeles, 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 iniţial era al doilea.

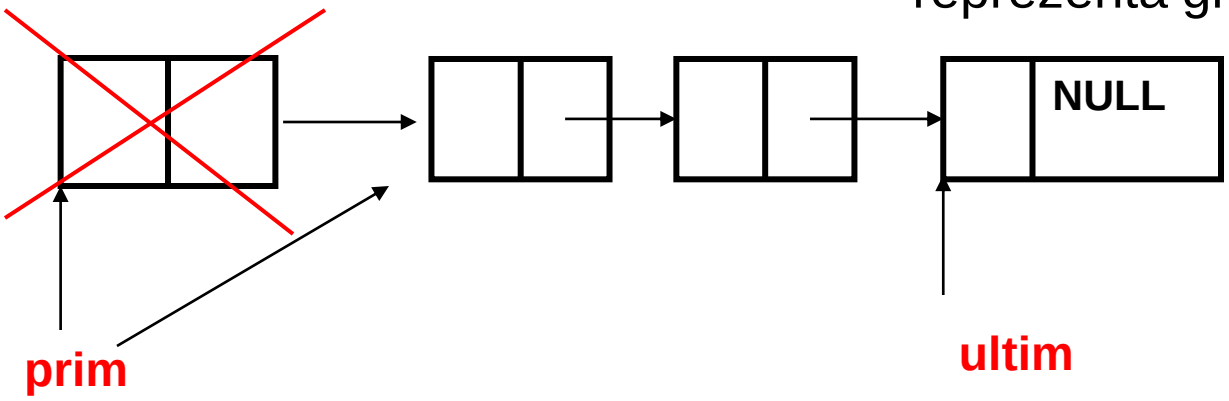
4.2. Coada

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

- Se reţine într-o variabilă (***q**) informaţia utilă din primul element al cozii - (**a**)
- Se reţine 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 iniţial 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)
}
```

Operaţia de ştergere, 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;
}
```

Conţinutul cursului

4. Structuri implementate dinamic:

- **4.1. Stiva**
- **4.2. Coada**
- **4.3. Lista simplu înlănţuită**
- **4.4. Lista dublu înlănţuită**

4.3. Lista simplu înlănţuită

- Lista simplu înlănţuită este structura de reprezentare a informaţiilor cea mai cunoscută şi implicit cea mai utilizată atunci când ne referim la alocarea dinamică a memoriei.

- *O **listă simplu înlănţuită** este formată dintr-o colecţie de elemente de acelaşi tip.*

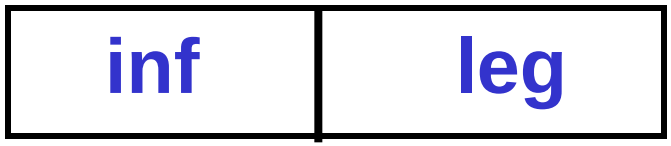
4.3. Lista simplu înlănţuită

- Fiecare element conţine în afară de **elementul propriu-zis** şi **o legatură** care ne spune unde putem găsi un alt element din mulţime.
- Elementele listei vor fi numite şi *noduri*.

- Ideea este că fiecare nod dintr-o listă simplu înlănţuită conţine informaţii despre cum putem localiza un alt nod dintr-o mulţime de noduri.
- *Accesul imediat la fiecare nod din mulţime nu este necesar pentru că fiecare nod conduce la un altul*.

4.3. Lista simplu înlănţuită

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



- Prin **inf** înţelegem informaţia ataşată elementului respectiv, şi care poate fi de orice tip de date cunoscut de către limbajul C++, iar prin **leg** înţelegem un câmp de tip referinţă care va conţine adresa următorului element din mulţimea de elemente.

4.3. Lista simplu înlănţuită

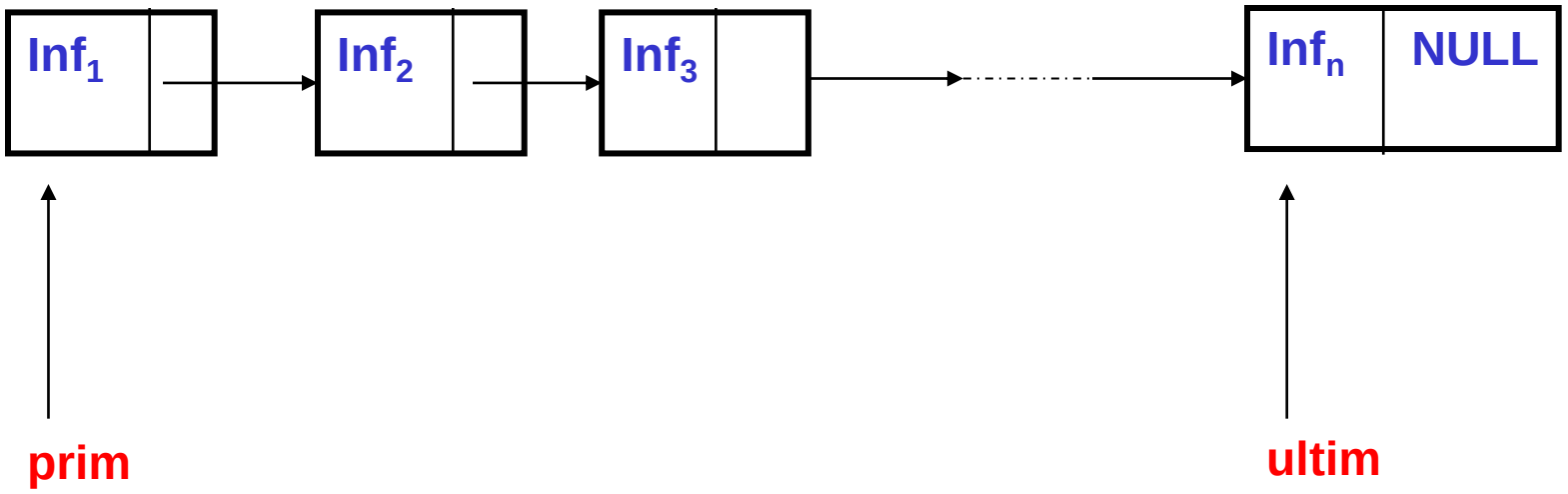
- Pentru a putea construi şi a folosi cât mai eficient o listă simplu înlănţuită este necesară o variabilă de tip referinţă care să indice primul element din listă.

- Convenim să notăm cu ***prim*** – adresa primului nod.

- Uneori, întâlnim aplicaţii care necesită şi folosirea adresei ultimului nod, notată cu ***ultim***.

4.3. Lista simplu înlănţuită

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



Avantaje

Listele simplu înlănţuite reprezintă o utilizare foarte importanta a alocarii dinamice a memoriei deoarece:

1. Sunt mai flexibile decât stiva şi coada (care restricţionează operaţiile de adăugare, acces şi ştergere a elementelor conform definiţiilor lor)
2. Se recomandă folosirea listelor simplu înlănţuite în rezolvarea problemelor specifice vectorilor, deoarece se utilizează eficient memoria care poate fi alocată sau eliberată în funcţie de cerinţele programatorului
3. Anumite genuri de probleme (cum ar fi operaţiile cu matrici rare; respectiv polinoame rare) îşi găsesc o rezolvare mai rapidă, eficientă şi utilă folosind listele

4.3. Lista simplu înlănţuită

Declaraţiile necesare lucrului cu o listă simplu înlănţuită 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 înlănţuită

Cu listele simplu înlănţuite se pot face următoarele operaţii:

1) Iniţializarea listei (crearea unei liste vide)

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

4.3. Lista simplu înlănţuită

2) Adăugarea unui element în listă

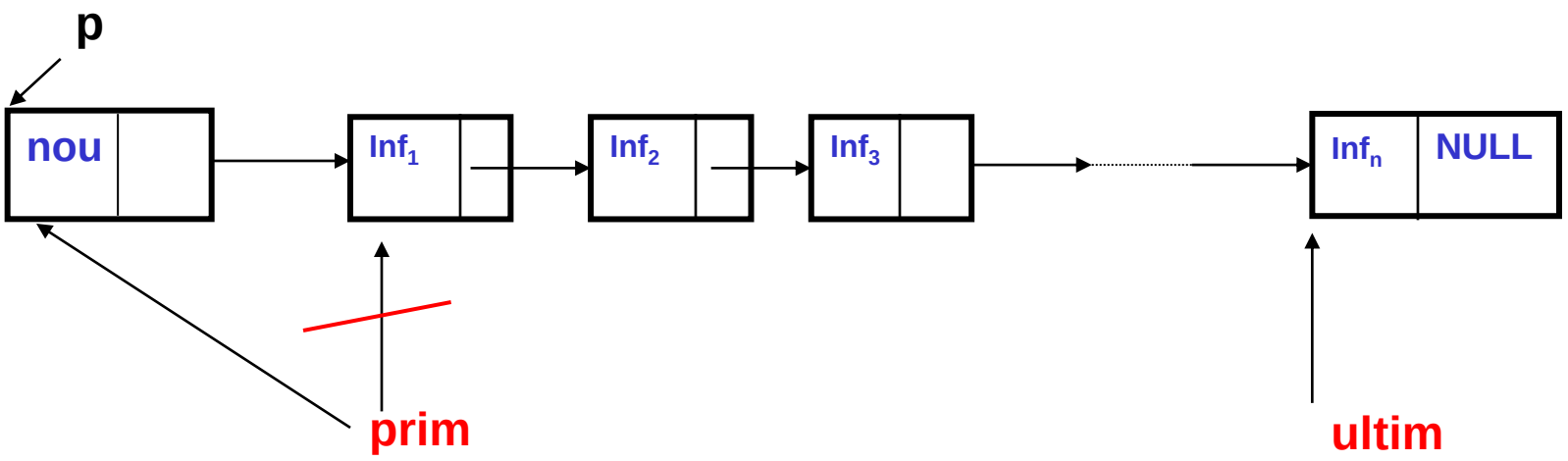
Aceasta operaţie se poate face în trei moduri, în funcţie de locul unde se inserează elementul în listă.

Astfel putem avea:

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

4.3. Lista simplu înlănţuită

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



Etapele adăugarii unui element la începutul listei:

- alocarea zonei de memorie necesare noului element (Se foloseşte un pointer de lucru **p**) - (**a**)
- completarea informaţiei utile pentru noul element (notată cu *nou*) - (**b**)
- completarea informaţiei de legătură cu adresa conţinută în variabila ***prim*** (ţinând cont că acesta va deveni primul element al listei şi, conform definirii acesteia, trebuie să conţină adresa elementului următor din listă, deci cel care era primul înainte de a face inserarea) - (**c**)
- Actualizarea variabilei referinţă ***prim*** cu adresa elementului creat, care în acest moment devine primul element al listei - (**d**)

4.3. Lista simplu înlănţuită

```
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)
}
```

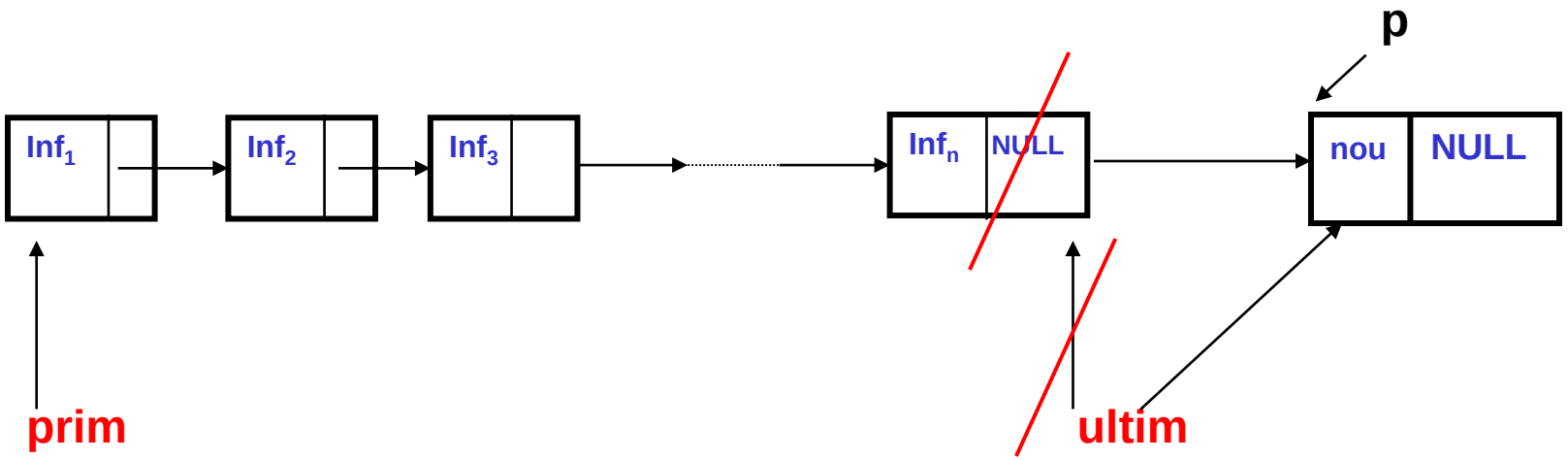
4.3. Lista simplu înlănţuită

b) Adăugarea unui element la sfârşitul 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 înlănţuită

Descrierea operaţiilor necesare se pot deduce şi de această dată folosind o reprezentare grafică a listei:



Paşii algoritmului de adăugare a unui element la sfârşitul listei:

- Alocarea zonei de memorie necesară pentru noul element - (**a**)
- Completarea informaţiei de legătură a elementului creat cu *NULL*, deoarece el va deveni ultim element - (**b**)
- Completarea informaţiei utile - (**c**)
- Informaţia de legatură a celui ce a fost înainte ultimul element al listei îşi 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 înlănţuită

```
void adauga_la_sfarsit( tip x, LISTA *ultim )
  // realizeaza adaugarea valorii x la sfarşitul 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 operaţie 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 sfârşitul listei în cazul în care nodul respectiv este ultimul nod din această listă, operaţie descrisă anterior în funcţia ***adaug_la_sfarsit***.

- Sunt necesari doi pointeri de lucru:
  1. **q –** indică nodul după care este facută inserarea
  2. **p** – pointer de lucru necesar pentru crearea unui nou element

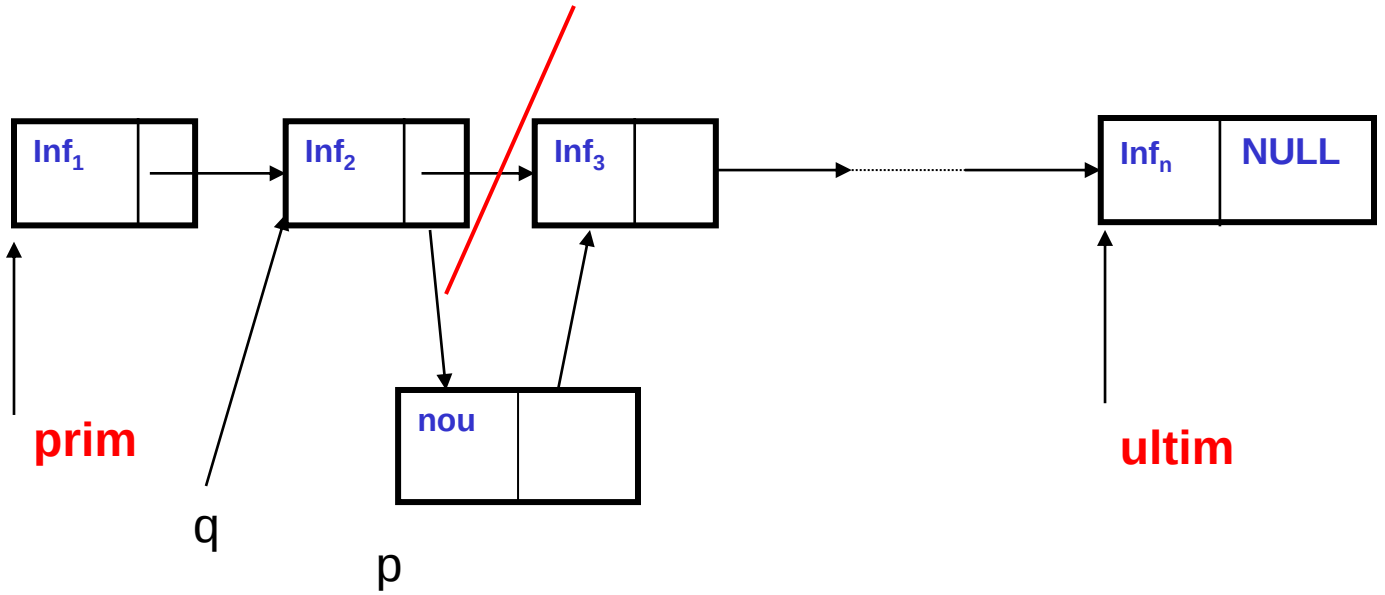
- Presupunem că avem o listă cu cel puţin două elemente, unde după nodul indicat de **q** vom adăuga un nou element cu valoarea informaţiei propriu-zise **x.**

4.3. Lista simplu înlănţuită

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**)
- iniţializarea informaţiei utile (cu valoarea notată **nou**) - (**b**)
- iniţializarea informaţiei de legătură cu adresa următorului nod (cu adresa reţinută în acest moment în variabila **q->leg**) - (**c**)
- actualizarea informaţiei 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 înlănţuită



4.3. Lista simplu înlănţuită

```
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 înlănţuită

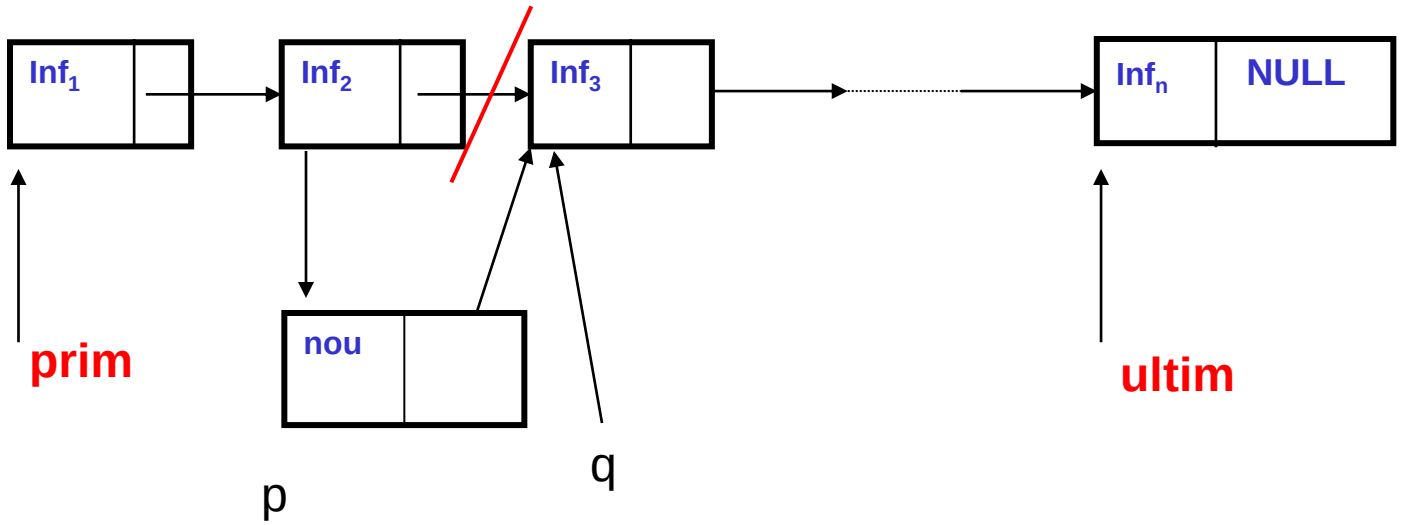
Cealaltă situaţie de inserare a unui nod în interiorul listei este de a realiza aceasta *înaintea unui nod* indicat de **q**.

Pentru această situaţie putem folosi următoarea solutie, care este cea mai utilizată, fiind mai rapidă şi eficientă şi va fi descrisă în următorii paşi:

- 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 informaţia ***q** (se realizează o dublură a informaţiei propriu-zise din nodul indicat de **q**) - (**b**)
- se actualizează câmpurile ***inf*** şi ***leg*** din nodul **q - (c,d)**

4.3. Lista simplu înlănţuită

Reprezentarea grafică corespunzătoare este următoarea:



4.3. Lista simplu înlănţuită

```
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 operaţie necesară pentru afişarea sau prelucrarea elementelor listei, ce constă în trecerea prin fiecare element al listei o singură dată.*
- Se foloseşte un pointer de lucru pe care-l voi nota cu **p** şi care *va fi iniţializat cu valoarea **prim***, iar *apoi îşi va schimba valoarea cu adresa nodului următor (folosind atribuirea p=p->leg)*, ceea ce se va repeta până când **p** va avea valoarea **NULL** (ceea ce înseamnă că a fost prelucrat şi ultimul nod).
- La fiecare schimbare a valorii pointerului **p**, se va afişa informaţia utilă din nodul indicat de **p**.
- Bineînţeles, lista nu trebuie să fie vidă.
- *Parcurgerea unei liste simplu înlănţuite se face secvenţial, într-un singur sens, de la primul către ultimul nod.*

4.3. Lista simplu înlănţuită

Afişarea 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<iostream>
using namespace std;
struct student{
    char nume[20];
    char prenume[20];
    float note[5];
    float media;
};
struct nod{
    student s;
    nod *leg;
};
nod *prim,*ultim;
int n;
```

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 creare_lista(nod *&prim, nod *&ultim)
{
    int i;
    cin>>n;
    nod *p;
    student x;
    prim=new nod;
    cin>>x.nume>>x.prenume;
    cin>>x.note[0]>>x.note[1]>>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;
```

Problemă rezolvată

```
prim->s = x;
prim->leg = NULL;
ultim = prim;
for(i=2; i<=n; i++)
     {
        cin>>x.nume>>x.prenume;
        cin>>x.note[0]>>x.note[1]>>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;
        p=new nod;
        p->s=x;
        p->leg=NULL;
        ultim->leg=p;
        ultim=p;
     }
}
```

Problemă rezolvată

```
void afis(nod *prim)
{
    nod *p = prim;
    while(p != NULL)
    {
       cout<<p->s.nume<<" "<<p->s.prenume<<" media
       = "<<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()
{
    creare_lista(prim, ultim);
    afis(prim);
    cout<<"Media generala a celor "<<n<<" studenti =
       "<<mediagen(prim);
    return 0;
}
```

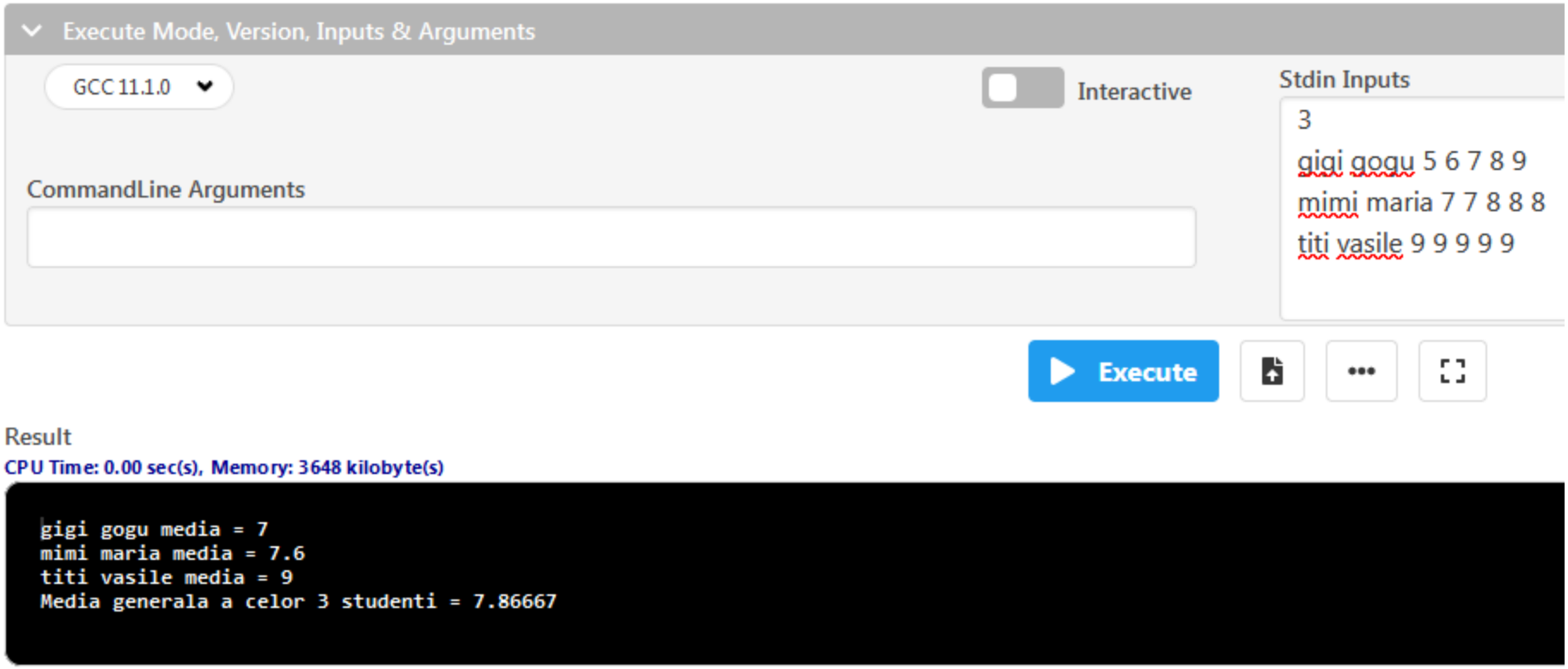
Problemă rezolvată


```
#include<iostream>
using namespace std;
struct student{
        char nume[20];
        char prenume[20];
        float note[5];
        float media;
};
struct nod{
    student s;
    nod *leg;
};
nod *prim,*ultim;
int n;

void creare_lista(nod *&prim, nod *&ultim)
{
    int i;
    cin>>n;
    nod *p;
    student x;
    prim=new nod;
    cin>>x.nume>>x.prenume;
    cin>>x.note[0]>>x.note[1]>>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;

    prim->s = x;
    prim->leg=NULL;
    ultim=prim;
    for(i=2;i<=n;i++)
    {
    cin>>x.nume>>x.prenume;
    cin>>x.note[0]>>x.note[1]>>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;

    p=new nod;
    p->s=x;
    p->leg=NULL;
    ultim->leg=p;
    ultim=p;
    }
}
```

Problemă rezolvată


```
43
44  void afis(nod *prim)
45  {
46      nod *p = prim;
47      while(p != NULL)
48      {
49          cout<<p->s.nume<<" "<<p->s.prenume<<" media = "<<p->s.media<<endl;
50          p = p -> leg;
51      }
52  }
53  float mediagen(nod *prim)
54  {
55      float s = 0;
56      int n = 0;
57      nod *p = prim;
58      while(p)
59      {
60          s += p -> s.media;
61          n++;
62          p = p -> leg;
63      }
64      return s / n;
65  }
66
67  int main()
68  {
69      creare_lista(prim,ultim);
70      afis(prim);
71      cout<<"Media generala a celor "<<n<<" studenti = "<<mediagen(prim);
72      return 0;
73  }
```

Problemă rezolvată



Grile

Grile cu alegere multiplă.

Identificați litera care corespunde răspunsului corect.

Grila nr. 1

Urmatoarea functie implementata in C++:

```
void Linked_List1(struct lista* head)
{
    if(head == NULL)  return;
    Linked_List1(head->leg);
    cout<<head->info<<' ';
}
```

a) Afiseaza toate elementele listei in ordinea in care a fost creata lista
b) Afiseaza toate elementele listei in ordine inversa celei in care a fost creata lista
c) Afiseaza elementele listei in ordine alternativa
d) Afiseaza elementele listei in ordine alternativa inversa

Grila nr. 2

Pentru lista liniara simplu inlantuita cu urmatoarele valori:
1->2->3->4->5->6, functia implementata in C++, va afisa:

```
void Linked_List2(struct lista* start)
{
   if(start == NULL)    return;
   cout<<start->info<<' ';
   if(start->leg != NULL )  Linked_List2(start->leg->leg);
   cout<<start->info<<' ';
}
```

a) 1 4 6 6 4 1
b) 1 3 5 1 3 5
c) 1 2 3 5
d) 1 3 5 5 3 1

Grila nr. 3

Pentru lista liniara simplu inlantuita cu urmatoarele valori:
1->2->3->4->5->6, functia implementata in C++, va afisa:

```
void Linked_List3(struct lista* start)
{   lista* p;
    p=start;
    while(p != NULL){
        if((p->info)%2==0)    cout<<p->info<<' ';
        p=p->leg;    }
    return;
}
```

a) 2 4 6
b) 2 2 4 4 6
c) 2 4 6 2 4 6
d) 6 4 2

Grila nr. 4

Pentru lista liniara simplu inlantuita cu urmatoarele valori: 11->202->43->99->1011->100, functia implementata in C++, va afisa:

```
void Linked_List4(struct lista* start)
{   lista* p;
    p=start;
    while(p != NULL){
          if((p->info)%2!=0 && p->info <= 99)
                cout<<p->info<<' ';
    p=p->leg;
    }
    return;
}
```

a) 11 43 99 1011
b) 11 43 99 99 43 11
c) 11 43 99
d) 11 202 42 99 1011 100

Grila nr. 5

Pentru lista liniara simplu inlantuita cu urmatoarele valori: 75->3->2->7->11->68, functia implementata in C++, va afisa:

```
void Linked_List5(struct lista* start)
{    lista* p;
     int s=0;
     for(p=start; p != NULL; p=p->leg)
          if((p->info)%2==0) s+=p->info;
    cout<<s;
    return;
}
```

a) 71
b) 70
c) 68
d) 69

Bibliografie

Mihaela Runceanu, Adrian Runceanu - **STRUCTURI DE DATE ALOCATE DINAMIC. Aspecte metodice. Implementări în limbajul C++**, 2016, Editura Academica Brancusi din Targu Jiu

https://www.researchgate.net/publication/308938197_STRUCTURI_DE_DATE_ALOCATE_DINAMIC_Aspecte_metodice_Implementari_in_limbajul_C/download

Întrebări?