

PROIECTAREA ALGORITMILOR

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

Adrian Runceanu

Curs 5

Liste dublu înlănțuite

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

4) Ștergerea unui element din listă

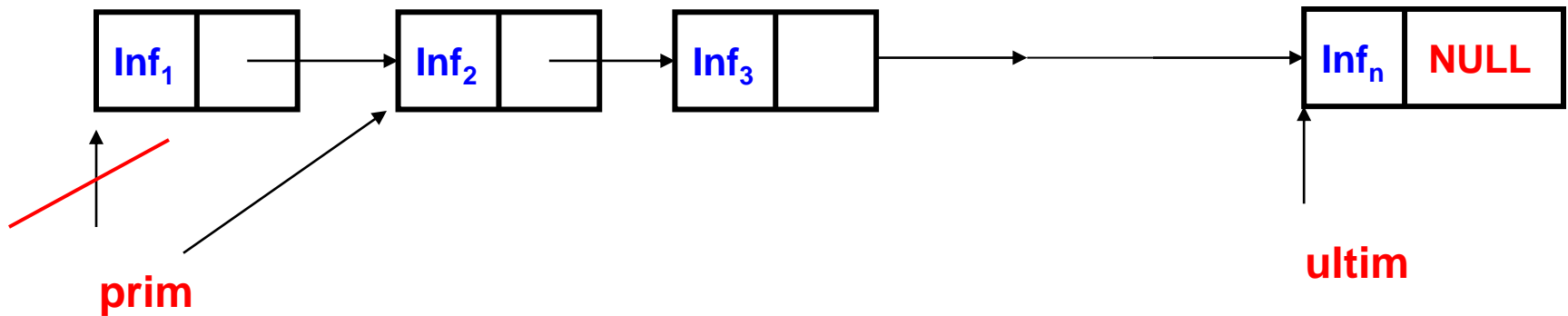
Operația de ștergere este destul de complexă și presupune studierea mai multor cazuri, fiecare cu particularitățile lui.

Astfel distingem următoarele cazuri de ștergere a unui element:

- a) Ștergerea primului element
- b) Ștergerea ultimului element
- c) Ștergerea unui element din interiorul listei

- a) **Ștergerea primului element** din listă determină schimbarea valorii variabilei **prim** deoarece nodul pe care îl indică inițial trebuie să fie eliminat.
- Astfel **prim** devine adresa celui de-al doilea element din listă care acum devine primul element al listei.
 - Pentru aceasta este necesar un pointer **p** pe care-l folosim pentru a reține adresa primului element (valoarea inițială a lui **prim**) pentru a putea elibera zona de memorie alocată acestuia.
 - Pornind de la o reprezentare grafică asemănătoare celei ce urmează, se poate deduce **algoritmul de ștergere** pentru această situație:

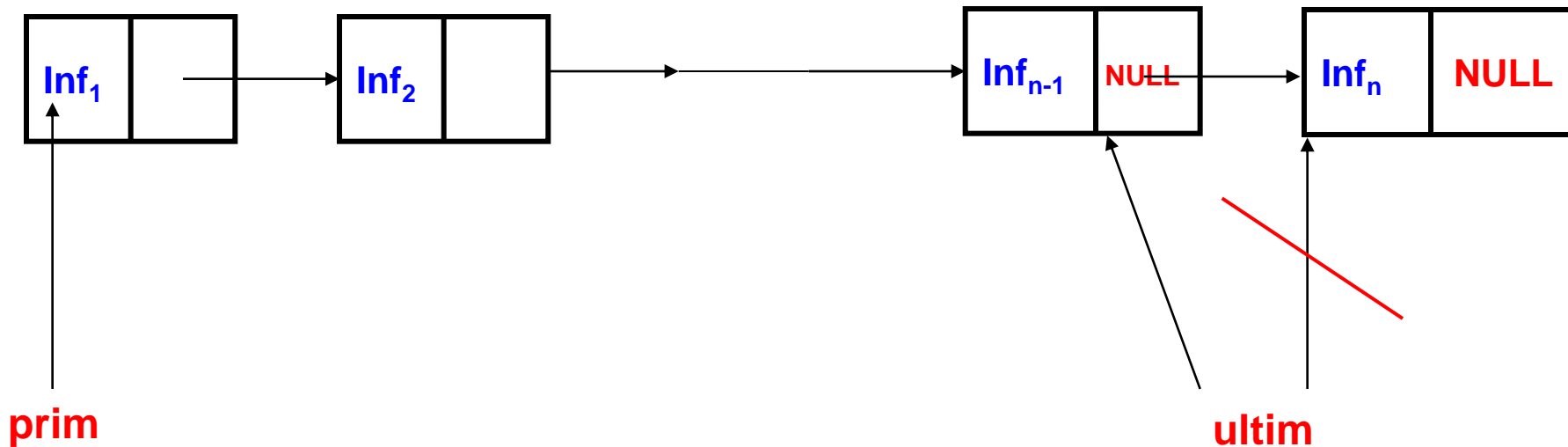
4.3. Lista simplu înlănțuită



4.3. Lista simplu înlănțuită

- b) **Ștergerea ultimului element** al listei are ca efect schimbarea pointerului ultim precum și a informației de legătură din penultimul nod care va deveni **NULL**, acest nod devenind astfel ultimul.
- Reprezentarea grafică a acestei situații este următoarea:

4.3. Lista simplu înlănțuită



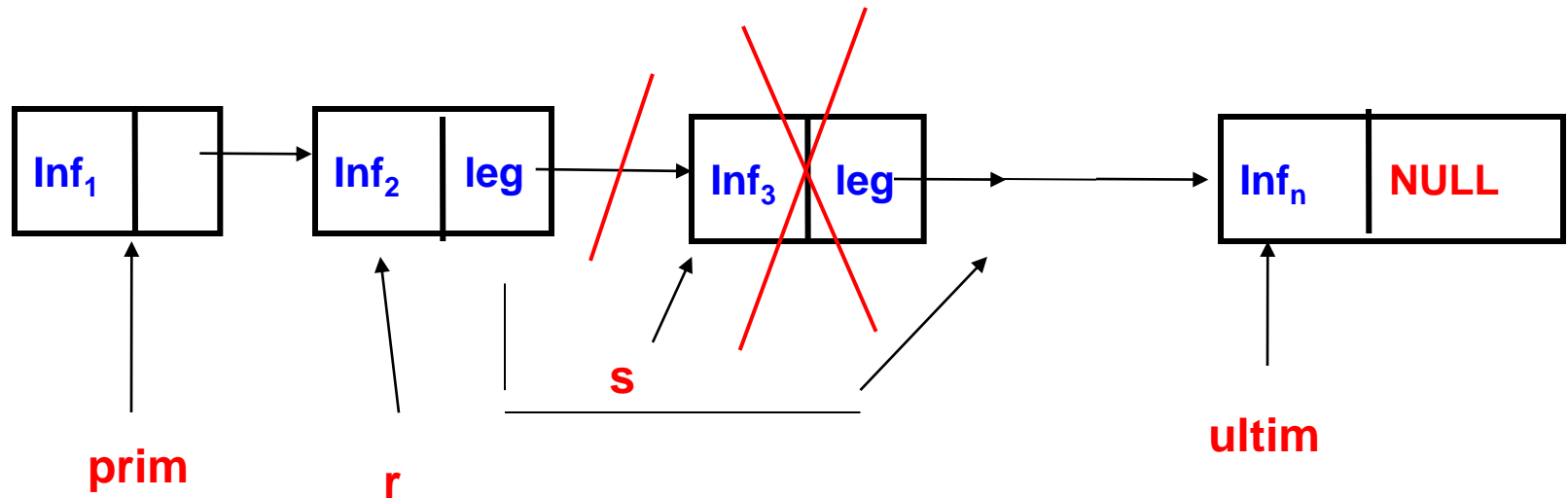
4.3. Lista simplu înlănțuită

c) Ștergerea unui nod din interiorul listei

presupune ștergerea unui nod referit de un pointer r , obținut prin localizarea anterioară a acestuia.

- Se realizează *o copie a nodului care urmează celui pe care-l vom șterge (indicat de $r \rightarrow \text{leg}$)*, iar apoi *îl vom șterge pe acesta (pe cel referit de $r \rightarrow \text{leg}$)*.

4.3. Lista simplu înlănțuită



4.3. Lista simplu înlănțuită

- Pe baza observațiilor făcute anterior, următoarea funcție de eliminare a unui element din listă va lua în considerare toate cele trei cazuri.
- Îi vom transmite un singur parametru și anume adresa nodului ce urmează a fi șters (**r**).

```
void stergere(LISTA *r)
{
    LISTA *s,*q;
    if (r==prim)
        {
            // Se va șterge primul element al listei
            s=prim;
            prim=prim->leg;
            delete s;
        }
}
```

```
else
  if (r==ultim)
  {
    // Se va sterge ultimul nod al listei
    q=prim; //va retine adresa penultimului nod
    while (q->leg != ultim)
      q=q->leg;
    s=ultim;
    ultim=q;
    ultim->leg=NULL;
    delete s;
  }
```

```
else
{
    // stergerea unui nod din interiorul listei
    s = r->leg; // în s se retine adresa nodului care
va fi sters efectiv, adica cel caruia îi vom face dublura
    r->inf = r->leg->inf; // am creat dublura nodului
urmator
    r->leg = r->leg->leg;
    delete s;
}
}
```

5) Căutarea unui element în listă

- De multe ori apar situații când este necesar să căutăm anumite informații în listă.
- Pentru aceasta, se va parcurge lista până când este găsită informația respectivă sau, în caz contrar, este epuizată lista și nu am găsit-o.
- Este indicat să folosim o variabilă care să indice dacă am găsit informația în listă sau nu, notată cu **gasit**:
 - 1** - dacă s-a găsit elementul
 - 0** - în caz contrar

- Pentru a face cât mai utilă operația de căutare, și având în vedere că elementul pe care îl vom găsi va fi folosit ulterior, în general avem nevoie de pointerul ce îl indică.
- Astfel putem folosi o funcție ce are ca rezultat pointerul ce indică elementul căutat.
- Dacă nu este găsit, acest pointer are valoarea **NULL**.


```
LISTA* caut(int x) // x-informatia pe care o cautam în lista
{
    LISTA *p;           // folosit pentru parcurgerea listei
    int gasit;
    gasit=0;
    p=prim;
    while (p!=NULL && !gasit )
        if (p->inf==x)
            gasit=1;
        else
            p=p->leg;
    if (gasit) return p;
    else return NULL;
}
```

}

Folosind inserarile la începutul și la sfârșitul listei, se pot realiza *două moduri de creare a listei*:

a) **Crearea prin inserare la sfârșitul listei** presupune *introducerea nodurilor unul după altul în ordinea citirii datelor de la tastatură*.

- Se crează mai întâi primul nod al listei, iar apoi se atașează după el celelalte elemente.
- Evident, elementele unei liste pot fi introduse prin citire, atribuire de valori sau diverse operații aritmetice.
- Alegem varianta citirii informației propriu-zise din fiecare nod.
- O astfel de creare a listei are la baza operația de adăugare a unei informații la sfârșitul listei.

```
void creare_la_sfarsit(LISTA *prim, LISTA *ultim)
{
    int x;
    char c;
    ultim=prim=new LISTA; /*este necesara folosirea pointerului
ultim deoarece adaugarile de elemente se fac la sfarsitul listei */
    cout<< "Introduceti prima informatie a listei: ";
    cin>>prim->inf;
    prim->leg=NULL;
    cout<<"Continuati sa introduceti elemente?(D/N)";
    cin>>c;
    while (toupper(c)=='D')
    {
        cin>>x;
        adaug_la_sfarsit(x,ultim); // functie definita anterior
        cout<<"Continuati sa introduceti elemente?(D/N)";
        cin>>c;
    }
}
```

b) O altă modalitate de **creare a unei liste este adăugarea elementelor la începutul listei**, inițial vidă.

Bineînțeles se va folosi funcția de adăugare la începutul listei.

```
void creare_la_inceput(LISTA *prim)
```

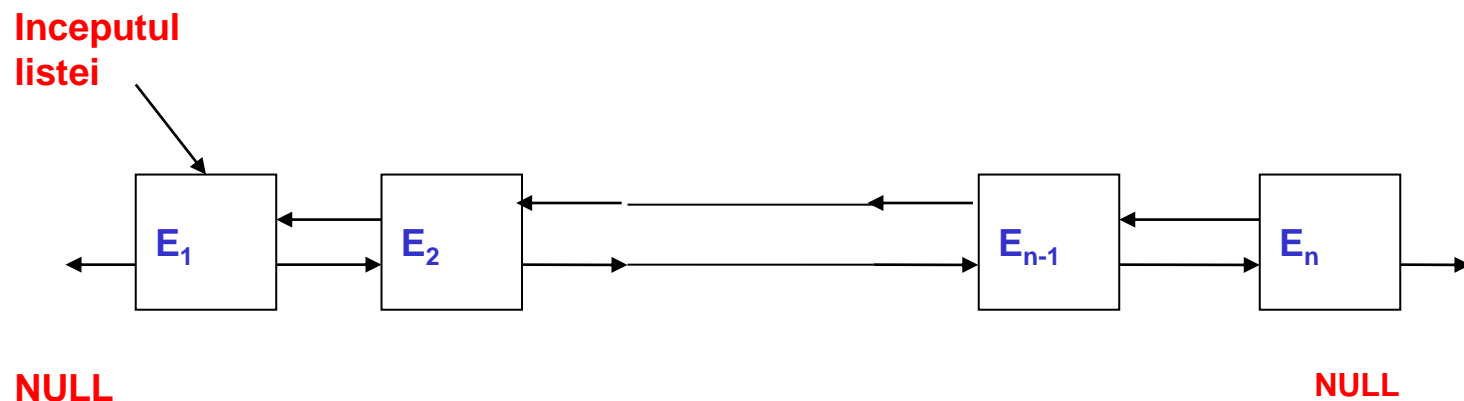
```
{  
    int x;  
    char c;  
    init(prim);  
    cout<<"Continuati sa introduceti elemente?(D/N)";  
    cin>>c;  
    while (toupper(c)=='D')  
    {  
        cin>>x;  
        adaug_la_inceput(x,prim); // funcție definita anterior  
        cout<<"Continuati sa introduceti elemente?(D/N)";  
        cin>>c;  
    }  
}
```

Conținutul cursului

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

4.4. Lista dublu înlănțuită

Listele dublu înlănțuite se deosebesc de cele simplu înlănțuite prin faptul că fiecare nod are două referințe, spre succesorul, respectiv predecesorul lui, ca în figura următoare:



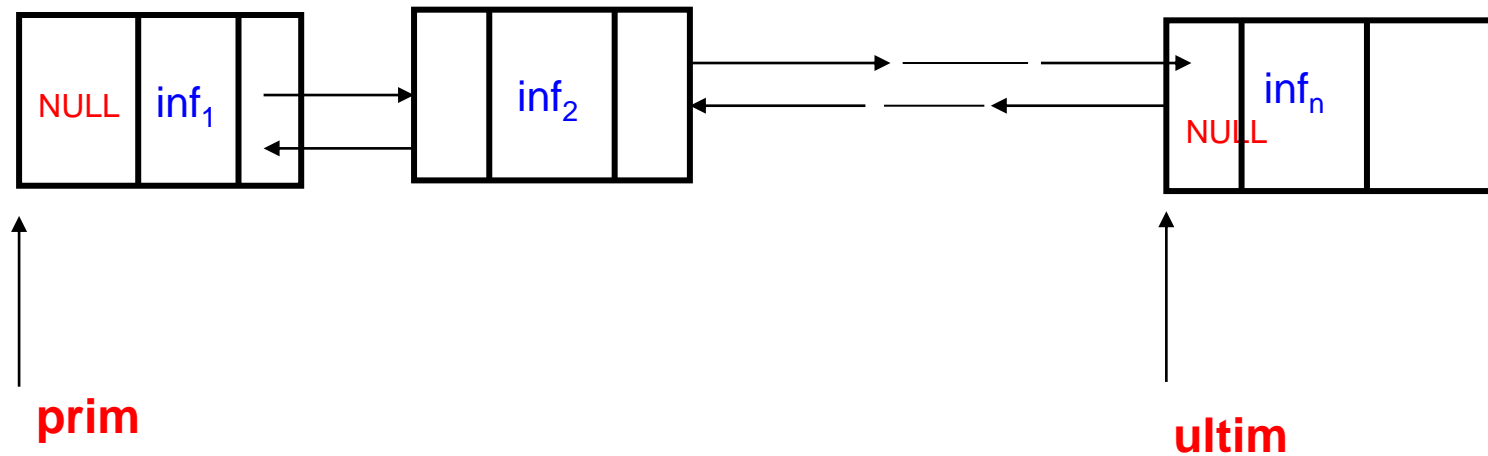
- Fiecare element al unei liste dublu înlănțuite are două câmpuri de legătură:
 - **ant** – adresa nodului anterior din listă
 - **urm** – adresa nodului următor din listă.
- Se observă că datorită câmpului **ant** structura de listă dublu înlănțuită este mai flexibilă decât cea de lista simplu înlănțuită.
- În primul rând este posibilă o parcurgere în ambele sensuri a listelor dublu înlănțuite, iar pe de altă parte dubla informație de legatură ne permite să accesăm mai ușor datele care ne interesează din listă.

4.4. Lista dublu înlănțuită

Declarațiile necesare pentru implementarea operațiilor cu listele dublu înlănțuite sunt:

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


4.4. Lista dublu înlănțuită



4.4. Lista dublu înlănțuită

- Pointerii **prim** și **ultim** indică primul respectiv ultimul nod al listei.
- Evident, primul element al listei are pentru câmpul **ant** valoarea **NULL** (nu are predecesor), iar ultimul element are pentru câmpul **urm** tot valoarea **NULL** (nu are succesor).
- Bineînțeles, pentru parcurgerea listelor dublu înlănțuite se poate porni de la primul element spre ultimul sau de la ultimul element spre primul, datorită existenței dublei informații de legătură.
- Aceasta va permite o anumită lejeritate în rezolvarea problemelor ce implică liste dublu înlănțuite.
- Chiar operațiile ce se efectuează asupra listelor dublu înlănțuite vor demonstra acest lucru.

4.4. Lista dublu înlănțuită

Ca și la listele simplu înlănțuite, avem următoarele operații posibile cu liste dublu înlănțuite:

- 1. inițializarea** listei
- 2. adăugarea** unui element:
 - la începutul listei
 - la sfârșitul listei
 - în interiorul listei
- 3. căutarea** unei valori
- 4. parcurgerea** listei
- 5. ștergerea** unui element

4.4. Lista dublu înlănțuită

1) Inițializarea listei (crearea unei liste vide)

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

2) Adăugarea unui element în listă

a) Adăugarea unui element la începutul listei

Inserarea la începutul listei presupune modificarea variabilei de tip referință ce indică primul element al listei și este efectuată respectând următorii pași:

- alocarea zonei de memorie necesare noului element (Se folosește un pointer de lucru **p**); **(a)**
- completarea informației utile pentru noul element; **(b)**
- completarea câmpului **urm** 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 lista, deci cel care era primul înainte de a face inserarea); **(c)**
- completarea câmpului **ant** cu valoarea NULL, deoarece nodul adăugat va fi primul nod al listei și nu va avea predecesor; **(d)**
- Actualizarea variabilei referință **prim** cu adresa elementului creat, care în acest moment devine primul element al listei; **(e)**

4.4. Lista dublu înlănțuită

```
void adaug_la_inceput( tip x, LISTA *prim)
```

```
    // x reprezinta informatia ce se adauga la inceputul listei
```

```
{
```

```
    LISTA *p;                // pointerul de lucru
```

```
    p=new LISTA;            // (a)
```

```
    p->inf=x;                // (b)
```

```
    p->urm=prim;            // (c)
```

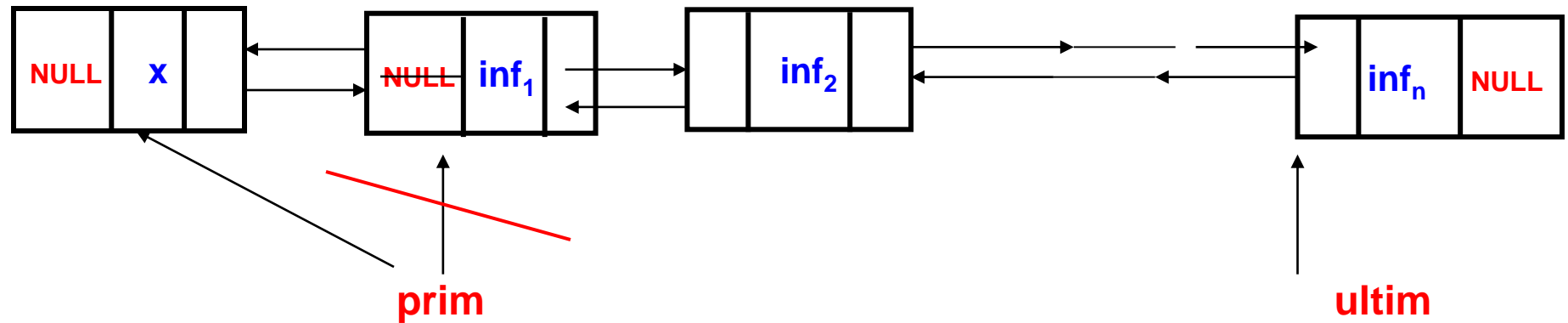
```
    p->ant=NULL;           // (d)
```

```
    prim->ant=p;           // (e)
```

```
    prim=p;                // (f)
```

```
}
```

4.4. Lista dublu înlănțuită



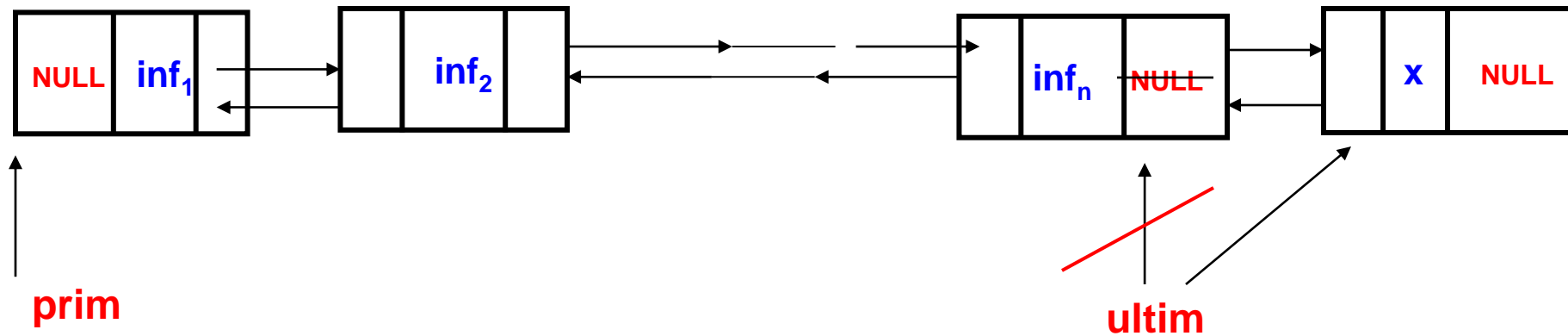
b) Adăugarea unui element la sfârșitul listei

- **Inserarea la sfârșitul listei** are ca efect atât modificarea pointerului **ultim** cât și a câmpului **urm** al ultimului nod.
Pașii algoritmului de adăugare a unui element la sfârșitul listei:
 - Alocarea zonei de memorie necesară pentru noul element; **(a)**
 - Completarea câmpului **urm** al elementului creat cu **NULL**, deoarece el va deveni ultimul element al listei; **(b)**
 - Completarea câmpului **ant** al elementului creat cu adresa ultimului nod al listei (care va deveni penultimul); **(c)**
 - Completarea informației utile; **(d)**
 - Câmpul **urm** al celui ce a fost înainte ultimul element al listei își schimbă valoarea cu adresa noului element (îl va indica pe acesta); **(e)**
 - Se actualizează pointerul **ultim** cu adresa nodului adăugat listei; **(f)**

4.4. Lista dublu înlănțuită

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

4.4. Lista dublu înlănțuită



4.4. Lista dublu înlănțuită

c) Adăugarea unui element în interiorul listei

- Operația de inserare în interiorul listei se face mai ușor având în vedere dublul acces la adresele elementelor 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:
 - **q** – indică nodul după care este făcută inserarea
 - **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**.

4.4. Lista dublu înlănțuită

Sucesiunea logică a etapelor necesare inserării după un nod (indicat de **q**) sunt următoarele:

- alocarea zonei de memorie necesare noului element (folosirea pointerului **p**); **(a)**
- inițializarea informației utile; **(b)**
- inițializarea câmpului **urm** al noului nod cu adresa nodului ce urmează în listă după nodul indicat de **q**; **(c)**
- inițializarea câmpului **ant** al noului nod cu valoarea **q** (nodul ce va precede noul nod); **(d)**
- actualizarea câmpului **urm** din nodul după care s-a inserat noul element cu adresa zonei de memorie alocată pentru acesta **(p)**; **(e)**
- actualizarea câmpului **ant** al lui **q->urm** cu adresa nodului creat; **(f)**

4.4. Lista dublu înlănțuită

```
void adaug_dupa( int x, LISTA *q)
{
    // adauga valoarea lui x intr-un nod ce urmeaza nodului
    // indicat de q in lista
    LISTA *p;           // pointer de lucru
    p=new LISTA;       // (a)
    p->inf=x;           // (b)
    p->urm=q->urm;     // (c)
    p->ant=q;           // (d)
    q->urm=p;           // (e)
    q->urm->ant=p;     // (f)
}
```

4.4. Lista dublu înlănțuită

- De asemenea, *se poate realiza adăugarea unui nod înainte* celui indicat de **q**.
- O astfel de operație devine mult mai simplă, deoarece adresa nodului precedent lui **q** este ușor de găsit, datorită dublei legături:
 - alocarea unei zone de memorie pentru noul element; **(a)**
 - completarea informației utile a noului nod; **(b)**
 - inițializarea câmpurilor de legătură ale nodului nou; **(c și d)**
 - actualizarea câmpului **urm** al nodului precedent lui **q** cu adresa noului nod ; **(e)**
 - actualizarea câmpului **ant** al lui **q** cu adresa nodului creat; **(f)**

4.4. Lista dublu înlănțuită

```
void adaug_inainte(tip x, LISTA *q)
{
    LISTA *p;           // pointer de lucru
    p=new LISTA;       // (a)
    p->inf=x;           // (b)
    p->urm=q;           // (c)
    p->ant=q->ant;      // (d)
    q->ant->urm=p;      // (e)
    q->ant=p;           // (f)
}
```

3). Traversarea (parcurgerea) listei

- Parcurgerea pentru diverse prelucrari a unei liste dublu înlănțuite se poate face în două sensuri, atât de la primul nod la ultimul, cât și invers, de la ultimul nod la primul, datorită dublei legături.

a) parcurgerea de la primul catre ultimul nod

```
void parcurgere1 ( LISTA *prim)
```

```
{
    LISTA *p;
    p=prim;
    while (p!=NULL)
    {
        prelucrare(p->inf);    // o operație oarecare asupra
informației din listă
        p=p->urm;
    }
}
```


b) parcurgerea de la ultimul nod către primul:

```
void parcurgere2 ( LISTA *ultim)  
{  
    LISTA *p;  
    p=ultim;  
    while (p!=NULL)  
    {  
        prelucrare(p->inf);  
        p=p->ant;  
    }  
}
```

4) Ștergerea unui element din listă

Operația de ștergere a unui element al listei dublu înălțuite este în principiu aceeași cu cea de la listele simplu înălțuite, însă ținând cont de legăturile duble:

```
void stergere(LISTA *r)
{
    // r - adresa nodului care va fi șters
    LISTA *s,*q;
    if (r==prim)
    {
        // se va șterge primul element al listei
        s=prim;
        prim=prim->urm;
        prim->ant=NULL;
        delete s;
    }
}
```

```
else
  if (r==ultim)
  {
    // se va șterge ultimul nod al listei
    s=ultim;
    ultim=ultim->ant;
    ultim->urm=NULL;
    delete s;
  }
else
  {
    // ștergerea unui nod din interiorul listei
    s=r;
    // în s se reține adresa nodului care va fi șters efectiv
    r->urm->ant=r->ant;
    r->ant->urm=r->urm;
    delete s;
  }
}
```

Spre deosebire de listele simplu înlănțuite, *crearea unei liste dublu înlănțuite se face prin adăugarea de elemente la sfârșitul listei*, deoarece este necesară adresa ultimului nod din lista.

```
void creares(LISTA *prim, LISTA *ultim)
{
    int x;
    char c;
    ultim=prim=new LISTA; /* este necesara folosirea pointerului
ultim deoarece adaugarile de elemente se fac la sfarșitul listei */
    cout<< "Introduceti prima informatie a listei: ";
    cin>>prim->inf;
    prim->urm=prim->ant=NULL;
    cout<<"Continuati sa introduceti elemente?(D/N)";
    cin>>c;
```

4.4. Lista dublu înlănțuită

```
while (toupper(c)=='D')
{
    cin>>x;
    adaug_la_sfarsit(x,ultim); // funcție definita anterior
    cout<<"Continuati sa introduceti
    elemente?(D/N)";
    cin>>c;
}
}
```



Grile

Grile cu alegere multiplă.

Identificați litera care corespunde răspunsului corect.





Grila nr. 1

Ce executa urmatorul program C++?

```
#include <iostream>
using namespace std;
struct nod {
    int info;
    nod* urm;
    nod* ant;
};
int main()
{
    nod* q=NULL; nod* p;
    nod *prim; nod* ultim;
    int i;
    prim=ultim=NULL;
    for ( i = 1 ; i <= 10; i++) {
        if(prim == NULL) {
            p = new nod;
            p->info = i;
            p->urm = NULL;
            p->ant = NULL;
            prim = p;
            ultim = p;
        }
```

```
        else
        {
            p = new nod;
            p->info = i;
            p->urm = NULL;
            p->ant = ultim;
            ultim->urm = p;
            ultim = p;
        }
    }
    q = prim;
    while (q)
    {
        cout<<q->info<<" ";
        q = q->urm;
    }
    return 0;
}
```

- a) 0 1 2 3 4 5 6 7 8 9 10
- b) 1 2 3 4 5 6 7 8 9 10
- c) 0 1 2 3 4 5 6 7 8 9
- d) 1 2 3 4 5 6 7 8 9



Grila nr. 2

Ce executa urmatorul program C++?

- a) 10 9 8 7 6 5 4 3 2 1
- b) 9 8 7 6 5 4 3 2 1 0
- c) 10 9 8 7 6 5 4 3 2 0
- d) 10 9 8 7 6 5 4 3 2 1 0

```
#include <iostream>
using namespace std;
struct nod {
    int info;
    nod* urm;
    nod* ant;
};
int main()
{
    nod* q=NULL; nod* p;
    nod *prim; nod* ultim;
    int i;
    prim=ultim=NULL;
    for ( i = 1 ; i <= 10; i++) {
        if(prim == NULL) {
            p = new nod;
            p->info = i;
            p->urm = NULL;
            p->ant = NULL;
            prim = p;
            ultim = p;
        }
    }
```

```
else
    {
        p = new nod;
        p->info = i;
        p->urm = NULL;
        p->ant = ultim;
        ultim->urm = p;
        ultim = p;
    }
}
q = ultim;
while (q)
    {
        cout<<q->info<<" ";
        q = q->ant;
    }
return 0;
}
```




Grila nr. 3

Ce executa programul C++, daca se introduc urmatoarele 8 valori: 1 2 3 4 5 6 7 8?

```
#include <iostream>
using namespace std;
struct nod {
    int info;
    nod* urm;
    nod* ant;
};
int main()
{
    nod* q=NULL; nod* p;
    nod *prim; nod* ultim;
    int i, n, x;
    prim=ultim=NULL;
    cin>>n;
    for ( i = 1 ; i <= n; i++) {
        cin>>x;
        if(prim == NULL) {
            p = new nod;
            p->info = x;
            p->urm = NULL;
            p->ant = NULL;
            prim = p;
            ultim = p;
        }
    }
```

```
else
    {
        p = new nod;
        p->info = x;
        p->urm = NULL;
        p->ant = ultim;
        ultim->urm = p;
        ultim = p;
    }
}
q = prim;
while (q)
{
    if(q->info%2==0) cout<<q->info<<" ";
    q = q->urm;
}
return 0;
}
```

- a) 2 4 6 8 1 3 5 7
- b) 1 2 3 4 5 6 7 8
- c) 2 4 6 8 10
- d) 2 4 6 8



Grila nr. 4

Ce executa programul C++, daca se introduc urmatoarele 9 valori: 11 6 35 77 24 4 9 8 55?

```
#include <iostream>
using namespace std;
struct nod {
    int info;
    nod* urm;
    nod* ant;
};
int main()
{
    nod* q=NULL; nod* p;
    nod *prim; nod* ultim;
    int i, n, x;
    prim=ultim=NULL;
    cin>>n;
    for ( i = 1 ; i <= n; i++) {
        cin>>x;
        if(prim == NULL) {
            p = new nod;
            p->info = x;
            p->urm = NULL;
            p->ant = NULL;
            prim = p;
            ultim = p;
        }
        else
```

```
        {
            p = new nod;
            p->info = x;
            p->urm = NULL;
            p->ant = ultim;
            ultim->urm = p;
            ultim = p;
        }
        q = prim;
        while (q)
        {
            if(q->info%2!=0) cout<<q->info<<" ";
            q = q->urm;
        }
        return 0;
    }
}
```

- a) 11 35 77 9
- b) 11 35 77 9 55
- c) 6 24 4 8
- d) 11 6 35 77 24 4 9 8 55



Grila nr. 5

Ce executa programul C++, daca se introduc urmatoarele 8 valori: 1 2 3 4 5 6 7 8?

```
#include <iostream>
using namespace std;
struct nod {
    int info;
    nod* urm;
    nod* ant;
};
int main()
{
    nod* q=NULL; nod* p;
    nod *prim; nod* ultim;
    int i, n, x;
    prim=ultim=NULL;
    cin>>n;
    for ( i = 1 ; i <= n; i++) {
        cin>>x;
        if(prim == NULL) {
            p = new nod;
            p->info = x;
            p->urm = NULL;
            p->ant = NULL;
            prim = p;
            ultim = p;
        }
    }
```

```
else
{
    p = new nod;
    p->info = x;
    p->urm = NULL;
    p->ant = ultim;
    ultim->urm = p;
    ultim = p;
}
}
q=prim;
while (q->urm->urm)
{
    cout<<q->urm->info<<" ";
    q=q->urm->urm;
}
return 0;
}
```

- a) 2 4 6 8
- b) 1 3 5 7
- c) 2 4 6
- d) 1 3 5



Grila nr. 6

Ce executa programul C++, daca se introduc urmatoarele 8 valori: 1 2 3 4 5 6 7 8?

```
#include <iostream>
using namespace std;
struct nod {
    int info;
    nod* urm;
    nod* ant;
};
int main()
{
    nod* q=NULL; nod* p;
    nod *prim; nod* ultim;
    int i, n, x;
    prim=ultim=NULL;
    cin>>n;
    for ( i = 1 ; i <= n; i++) {
        cin>>x;
        if(prim == NULL) {
            p = new nod;
            p->info = x;
            p->urm = NULL;
            p->ant = NULL;
            prim = p;
            ultim = p;
        }
    }
```

```
else
    {
        p = new nod;
        p->info = x;
        p->urm = NULL;
        p->ant = ultim;
        ultim->urm = p;
        ultim = p;
    }
}
q=ultim;
while (q->ant->ant)
    {
        cout<<q->ant->info<<" ";
        q=q->ant->ant;
    }
return 0;
}
```

- a) 8 6 4 2
- b) 7 5 3 1
- c) 6 4 2
- d) 7 5 3



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_STRUCTUREI_DE_DATE_ALOCATE_DINAMIC_Aspecte_metodice_Implementari_in_limbajul_C/download

Întrebări?