5

PROIECTAREA ALGORITMILOR

Adrian Runceanu

Curs 5

Liste dublu înlănţuite

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

4) Ştergerea unui element din listă

Operaţia de ştergere este destul de complexă şi presupune studierea mai multor cazuri, fiecare cu particularităţile lui.

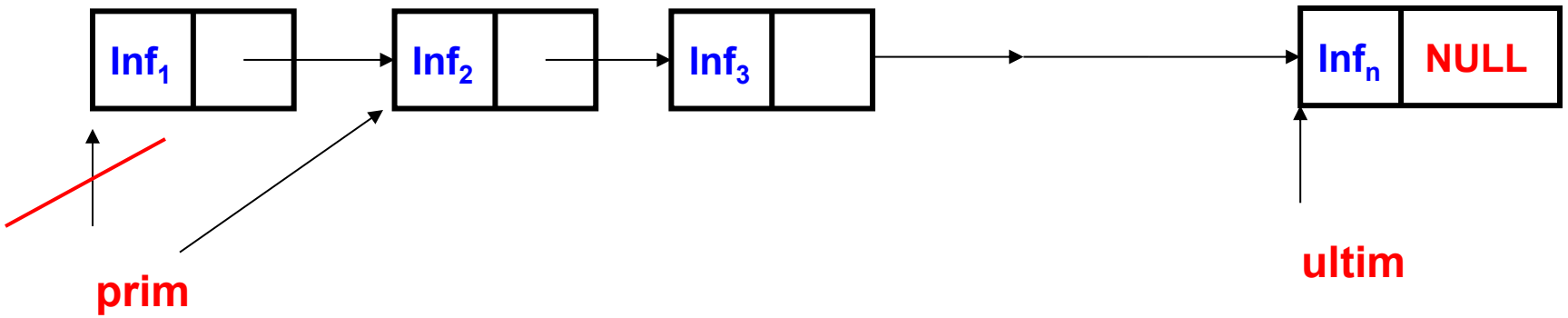
Astfel distingem următoarele cazuri de ştergere a unui element:

a) Ştergerea primului element
b) Ştergerea ultimului element
c) Ştergerea unui element din interiorul listei

a) ***Ştergerea primului element*** din listă determină schimbarea valorii variabilei **prim** deoarece nodul pe care îl indică iniţial 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 reţine adresa primului element (valoarea iniţială 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 ştergere pentru această situaţie:

4.3. Lista simplu înlănţuită

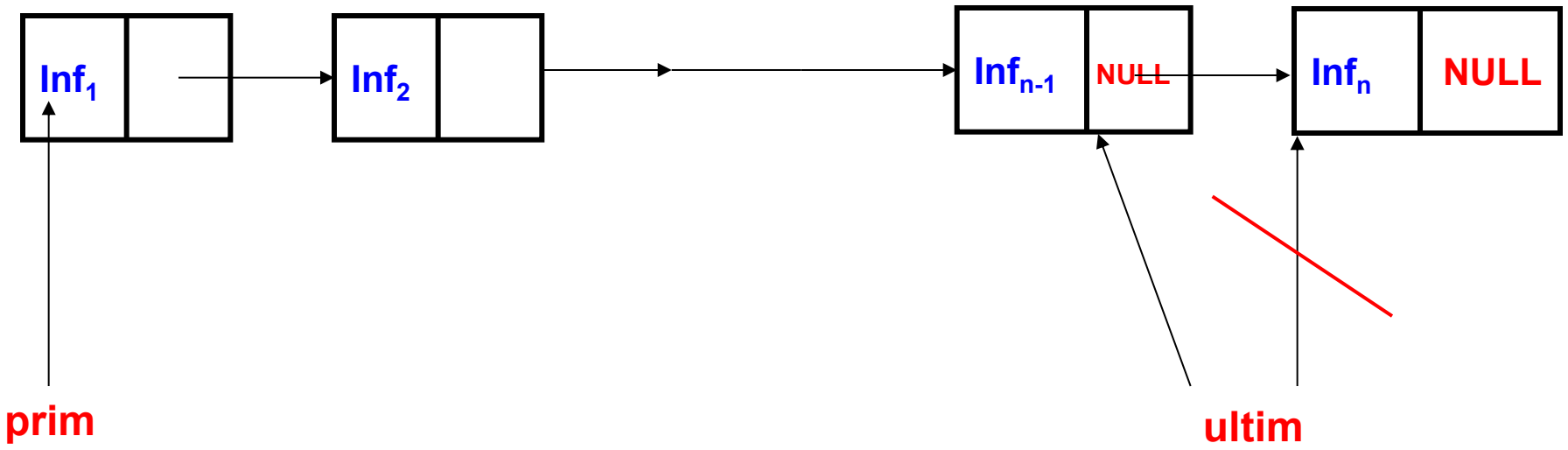


4.3. Lista simplu înlănţuită

b) ***Ştergerea ultimului element*** al listei are ca efect schimbarea pointerului ultim precum şi a informaţiei de legătură din penultimul nod care va deveni ***NULL***, acest nod devenind astfel ultimul.

- Reprezentarea grafică a acestei situaţii este următoarea:

4.3. Lista simplu înlănţuită

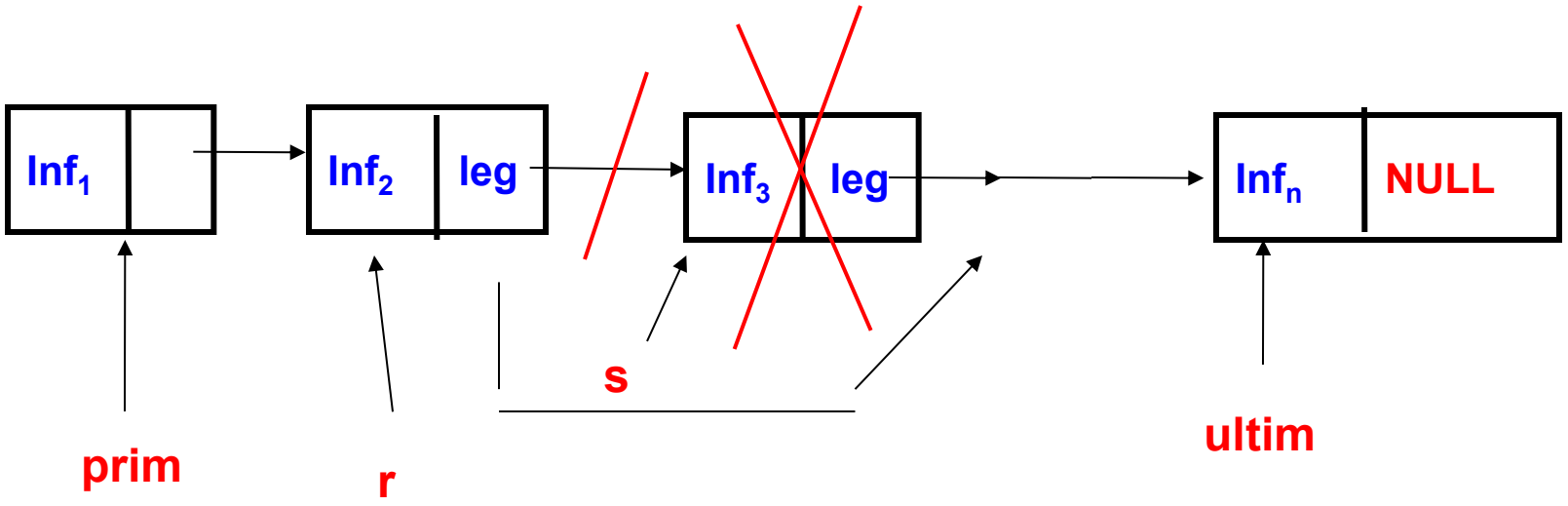


4.3. Lista simplu înlănţuită

c) ***Ştergerea unui nod din interiorul listei*** presupune ştergerea unui nod referit de un pointer **r**, obţinut prin localizarea anterioară a acestuia.

- Se realizează *o copie a nodului care urmează celui pe care-l vom şterge (indicat de r->leg),* iar apoi *îl vom şterge pe acesta (pe cel referit de r->leg).*

4.3. Lista simplu înlănţuită



4.3. Lista simplu înlănţuită

- Pe baza observaţiilor făcute anterior, următoarea funcţie de eliminare a unui element din listă va lua în considerare toate cele trei cazuri.

- Îi vom transmite un singur parametru şi anume adresa nodului ce urmează a fi şters (**r**).

```
void stergere(LISTA *r)
{
    LISTA *s,*q;
    if (r==prim)
        {
        // Se va şterge 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 situaţii când este necesar să căutăm anumite informaţii în listă.
- Pentru aceasta, se va parcurge lista până când este găsită informaţia respectivă sau, în caz contrar, este epuizată lista şi nu am găsit-o.

- Este indicat să folosim o variabilă care să indice dacă am gasit informaţia în listă sau nu, notată cu ***gasit***:
  1 - dacă s-a găsit elementul
  0 - în caz contrar

- Pentru a face cât mai utilă operaţia de căutare, şi 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 funcţie 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 şi la sfârşitul listei, se pot realiza *două moduri de creare a listei*:

a) ***Crearea prin inserare la sfârşitul 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 ataşeaza dupa el celelalte elemente.
- Evident, elementele unei liste pot fi introduse prin citire, atribuire de valori sau diverse operaţii aritmetice.
- Alegem varianta citirii informaţiei propriu-zise din fiecare nod.
- O astfel de creare a listei are la baza operaţia de adăugare a unei informaţii la sfârşitul 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 sfarşitul 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);        // funcţie 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**, iniţial vidă.

Bineînţeles se va folosi funcţia 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);   // funcţie definita anterior
        cout<<"Continuati sa introduceti elemente?(D/N)";
        cin>>c;
    }
}
```

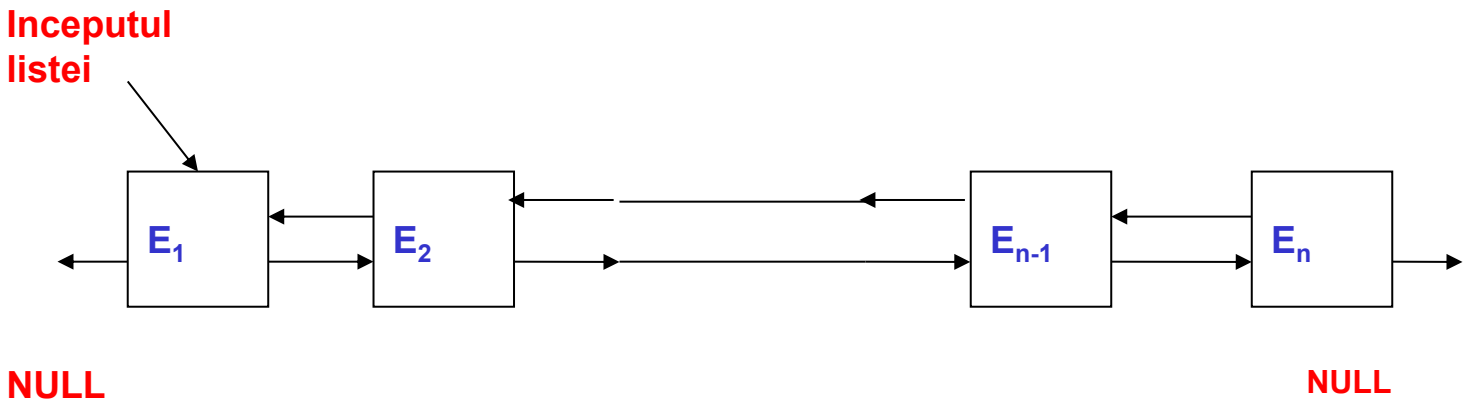
Conţinutul cursului

4. Structuri implementate dinamic:

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

4.4. Lista dublu înlănţuită

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



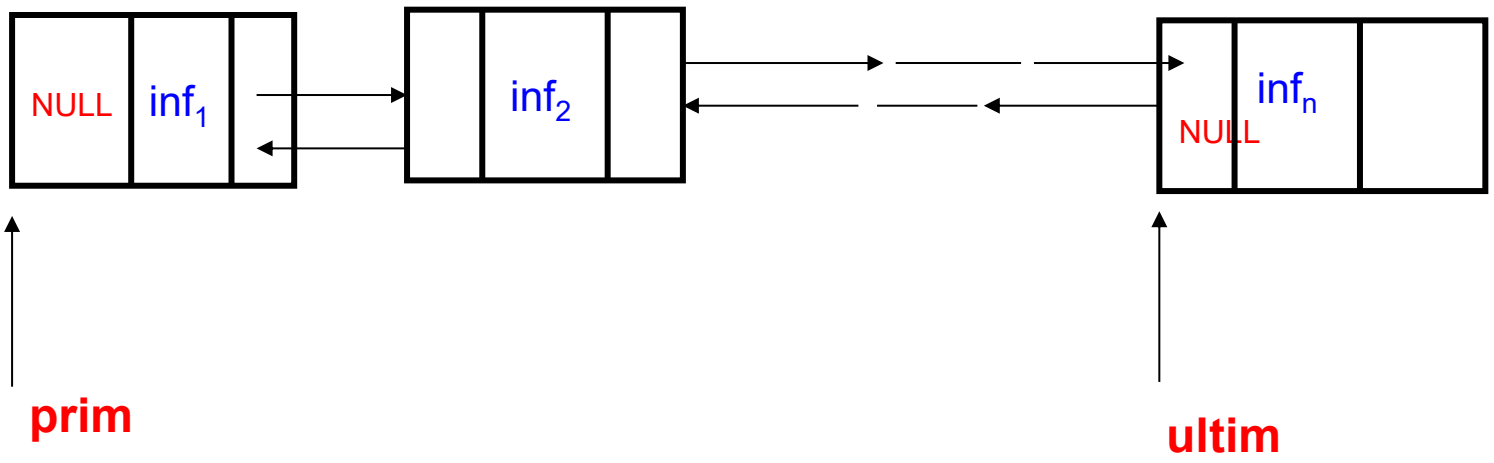
- Fiecare element al unei liste dublu înlănţuite are două câmpuri de legătură:
  - **ant – adresa** nodului anterior din listă
  - **urm – adresa** nodului următor din listă.
- Se observă că datorită câmpului **ant** structura de listă dublu înlăţuită este mai flexibilă decât cea de lista simplu înlănţuită.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

4.4. Lista dublu înlănţuită

Declaraţiile necesare pentru implementarea operaţiilor cu listele dublu înlănţuite 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 înlănţuită



4.4. Lista dublu înlănţuită

- Pointerii **prim** şi **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).
- Bineînţeles, pentru parcurgerea listelor dublu înlănţuite se poate porni de la primul element spre ultimul sau de la ultimul element spre primul, datorită existenţei dublei informaţii de legătură.
- Aceasta va permite o anumită lejeritate în rezolvarea problemelor ce implică liste dublu înlănţuite.
- Chiar operaţiile ce se efectuează asupra listelor dublu înlănţuite vor demonstra acest lucru.

4.4. Lista dublu înlănţuită

Ca şi la listele simplu înlănţuite, avem următoarele operaţii posibile cu liste dublu înlănţuite:

1. ***iniţializarea*** listei
2. ***adăugarea*** unui element:
   - la începutul listei
   - la sfârşitul listei
   - în interiorul listei
3. ***căutarea*** unei valori
4. ***parcurgerea*** listei
5. ***ştergerea*** unui element

4.4. Lista dublu înlănţuită

1) Iniţializarea 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

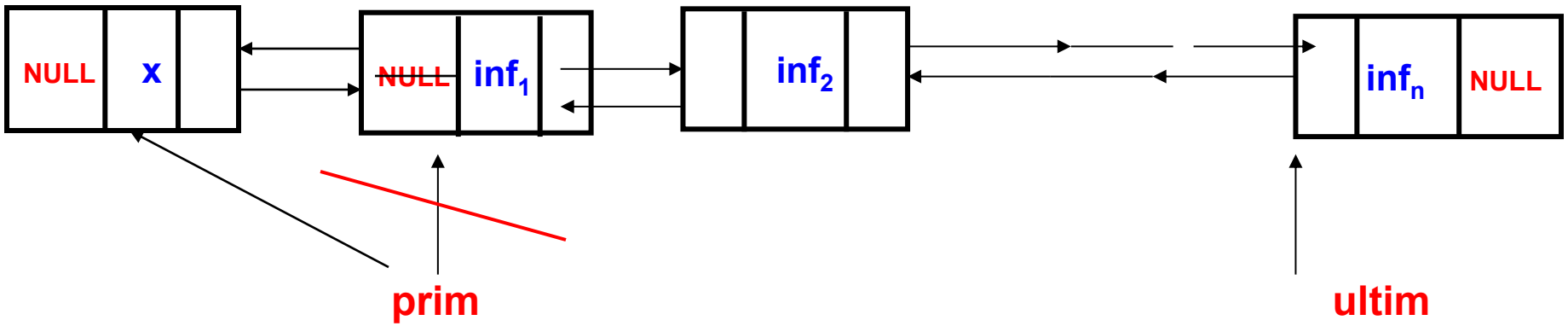
Inserarea la începutul listei presupune modificarea variabilei de tip referinţă ce indică primul element al listei şi este efectuată respectând următorii paşi:

- alocarea zonei de memorie necesare noului element (Se foloseşte un pointer de lucru **p**); (**a**)
- completarea informaţiei utile pentru noul element; (**b**)
- completarea câmpului **urm** 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 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 şi nu va avea predecesor; (**d**)
- Actualizarea variabilei referinţă ***prim*** cu adresa elementului creat, care în acest moment devine primul element al listei; (**e**)

4.4. Lista dublu înlănţuită

```
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 înlănţuită



b) Adăugarea unui element la sfârşitul listei

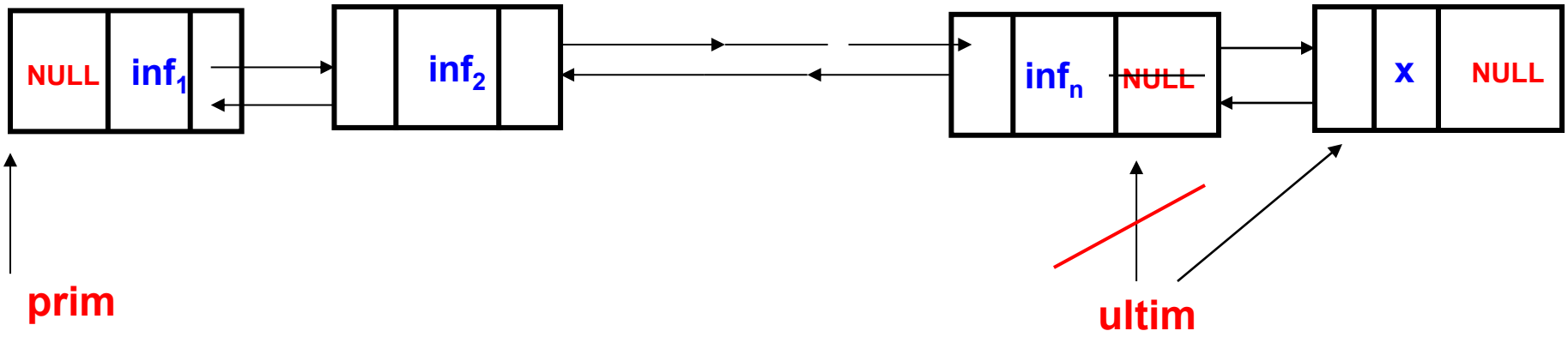
- **Inserarea la sfârşitul listei** are ca efect atât modificarea pointerului **ultim** cât şi a câmpului **urm** al ultimului nod.

  Paşii algoritmului de adăugare a unui element la sfârşitul 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 informaţiei utile; (**d**)
- Câmpul **urm** al celui ce a fost înainte ultimul element al listei îşi 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 înlănţuită

```
void adaug_la_sfarsit(tip x, LISTA *ultim)
// realizeaza adaugarea valorii x la sfarşitul 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 înlănţuită



4.4. Lista dublu înlănţuită

c) Adăugarea unui element în interiorul listei

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

4.4. Lista dublu înlănţuită

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

- alocarea zonei de memorie necesare noului element (folosirea pointerului **p**); (**a**)
- iniţializarea informaţiei utile; (**b**)
- iniţializarea câmpului **urm** al noului nod cu adresa nodului ce urmează în listă după nodul indicat de **q**; (**c**)
- iniţializarea 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 înlănţuită

```
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 înlănţuită

- De asemenea, *se poate realiza adăugarea unui nod înaintea* celui indicat de **q**.
- O astfel de operaţie devine mult mai simplă, deoarece adresa nodului precedent lui q este uşor de găsit, datorită dublei legături:
  - alocarea unei zone de memorie pentru noul element; (**a**)
  - completarea informaţiei utile a noului nod; (**b**)
  - iniţializarea câmpurilor de legatură ale nodului nou; (**c** şi **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 înlănţuită

```
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 înlănţuite se poate face în două sensuri, atât de la primul nod la ultimul, cât şi 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 operaţie oarecare asupra
    informaţiei 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) Ştergerea unui element din listă

Operaţia de ştergere a unui element al listei dublu înlănţuite este în principiu aceeaşi cu cea de la listele simplu înlănţuite, însă ţinând cont de legăturile duble:

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

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

Spre deosebire de listele simplu înlănţuite, *crearea unei liste dublu înlănţuite se face prin adăugarea de elemente la sfârşitul 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 sfarşitul 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 înlănţuită

```
while (toupper(c)==’D’)
{
   cin>>x;
   adaug_la_sfarsit(x,ultim); // funcţie 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++?

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

```
#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;
}
```

Grila nr. 2

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 = ultim;
      while (q)
         {
             cout<<q->info<<" ";
             q = q->ant;
         }
      return 0;
}
```

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

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_STRUCTURI_DE_DATE_ALOCATE_DINAMIC_Aspecte_metodice_Implementari_in_limbajul_C/download

Întrebări?