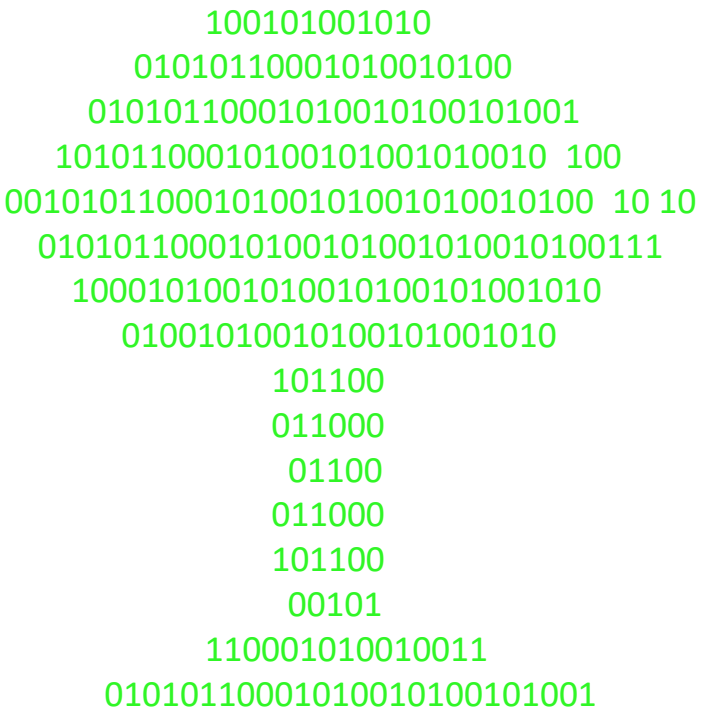
Structuri de date și algoritmi

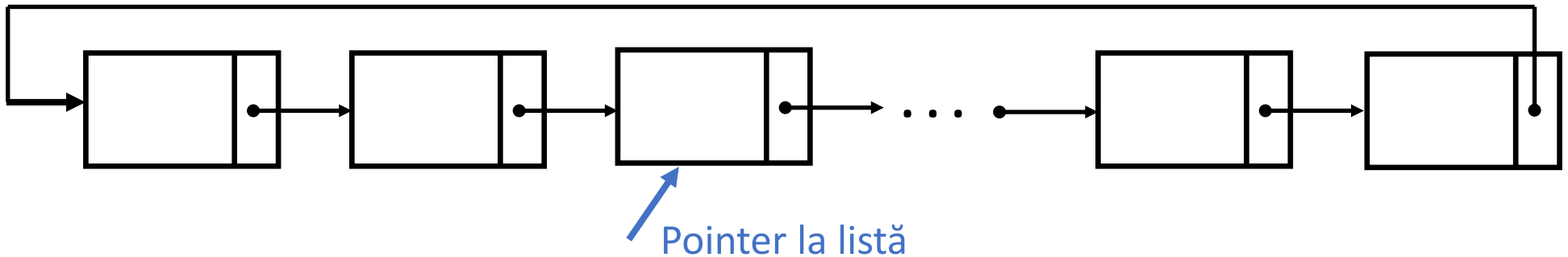


Curs, IS – An II

Liste circulare. Liste dublu înlănțuite. Aplicații.

Listă circulară simplu înlănțuită – Implementare dinamică

- O listă circulară este o variantă a listei liniare simplu înlănțuite de care se deosebește prin faptul că ultimul element are drept informație de înlănțuire pointer la ”capul” listei;
- Accesul la listă se realizează printr-un pointer la oricare element (pentru comoditate îl putem numi ”cap”) și nu mai are importanță care este primul sau ultimul element;
- Parcurgerea se realizează într-un singur sens;



Lista circulară simplu înlănțuită – Implementare dinamică

Declarații

```
typedef  int Atom;//tip predefinit

struct Element
{
   Atom data;
   Element *succ;
};
```

... main ....

```
Element *L;//pointer la lista
Element *LC=0;//inițializarea listei
```

Deosebiri față de lista liniară

- ✓ Inserarea primului nod presupune să specificăm și legătura spre el însuși;
- ✓ Condiția de terminare a listei: se compară adresa curentă a legăturii cu adresa ”capului” în locul comparației cu ”NULL” (0);
- ✓ Toate operțiile aferente unei liste liniare se particularizează pentru o listă circulară.

Operații cu liste circulare simplu înlănțuite

Parcurgerea listei (V1)

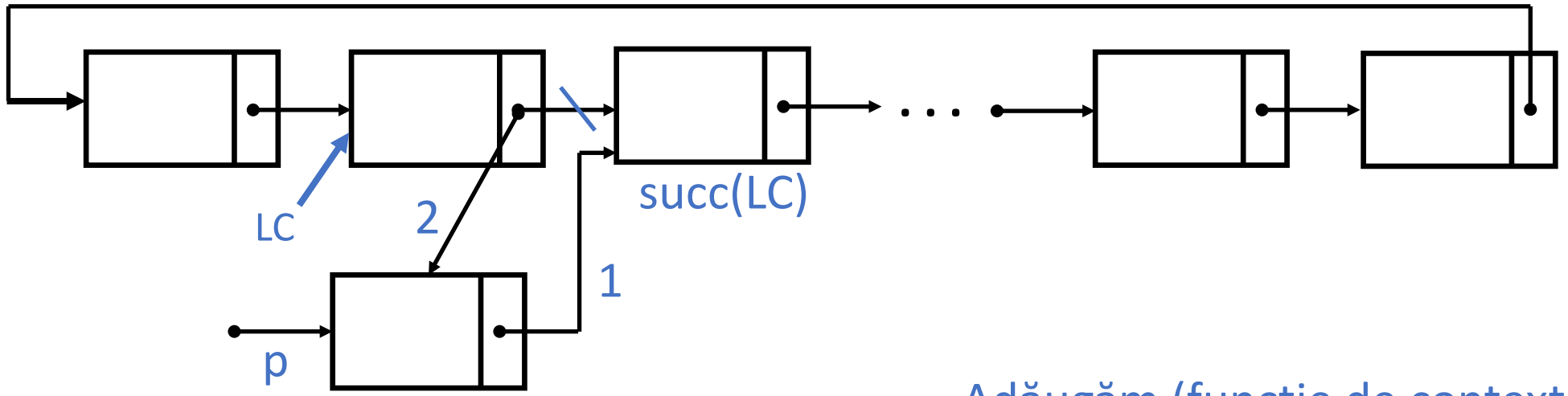
```
p← LC; //pointer la primul element
prelucrare (data(p));
While succ(p)≠LC Do
    p ← succ(p); // următorul element
    prelucrare (data(p));
EndWhile
```

Parcurgerea listei (V2)

```
p← LC; //pointer la primul element
While succ(p)≠LC Do
    prelucrare (data(p));
    p ← succ(p); // următorul element
EndWhile
prelucrare (data(p));
```

Operații cu liste circulare simplu înlănțuite

Inserarea după primul element



p ← get_sp();
data(p) ← info;
succ(p) ← succ(LC); (1)
succ(LC) ← p; (2)// p va fi succesorul lui LC

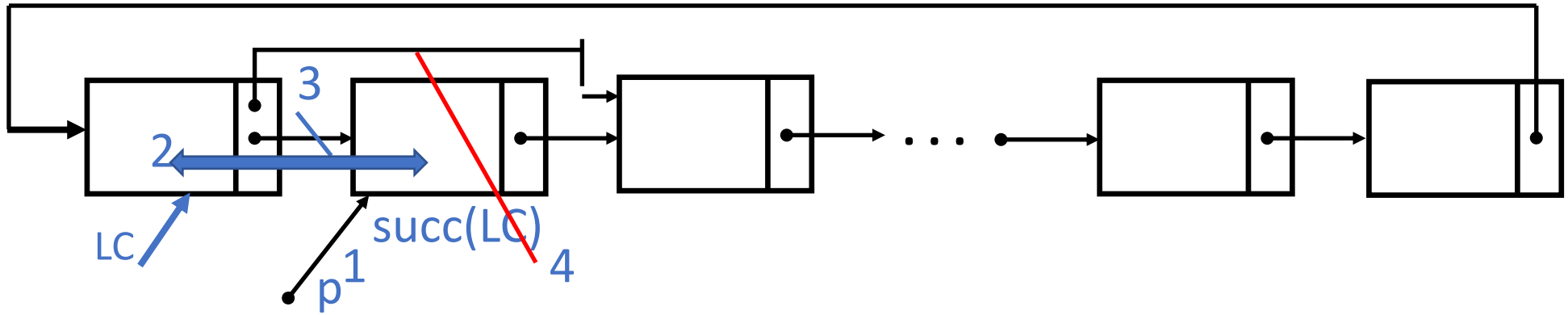
Adăugăm (funcție de context)
LC← p; (3a)//noul element devine ”cap”

sau
swap(data(LC),data(p)); (3b)

Complexitate O(1)

Operații cu liste circulare simplu înlănțuite

Ștergerea primului element



V1, lista are cel puțin 2 elemente

p ← succ(LC); (1)

swap(data(LC),data(p)); (2)

succ(LC)← succ(p); (3)

free_sp(p); (4)

Complexitate O(1)

V2

p← LC;

Se parcurge lista până când succ(p) ≡LC;

Se șterge LC ca la o listă liniară

Complexitate O(1)

Cum procedăm dacă lista are un singur element?

Operații cu liste circulare simplu înlănțuite

Inserarea sau ștergerea unui element în/din interiorul liste

Se realizează similar cu operațiile discutate la listele liniare.

Deoarece aceste operații presupun parcurgerea listei până la elementul care precede elemntul ce urmează a fi șters sau înaintea căruia se inserează, complexitatea este O(n)!

Exemplu: crearea unei liste circulare

```
struct Nod{
    int data;
    Nod *succ;
};
```

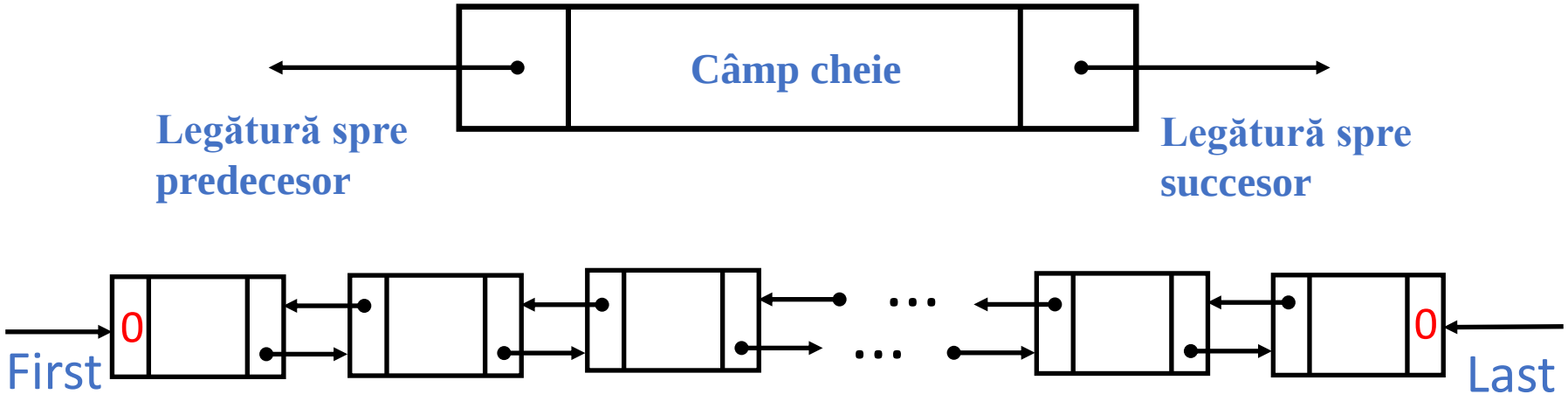
Apel in main()

```
Nod *p=0;//initializarea!!
CreateLC(p);
```

```
void CreateLC(Nod*& cap)
{   Nod* p; int n;
    cout << " valori - cu 0 se iese"<<endl;
    cin >> n;
    while (n) {
        p = new Nod; p->data = n;
        if (!cap){//pun primul element
            p->succ = p;
            cap = p;
        }
        else{
            p->succ = cap->succ;
            cap->succ = p;
            cap = p;
        }
        cin >> n;
    }
    if (cap) cap = cap->succ;//ultimul nod devine cap
}
```

Liste liniare dublu înlănțuite

Un element al unei liste dublu înlănțuite se poate identifica prin succesorul sau predecesorul său și are forma generală:



Lista poate fi parcursă în ambele sensuri ⟶ doi pointeri la capetele listei (First, Last)

First nu are predecesor (pointer NULL), Last nu are succesor (pointer NULL).

```
struct Nod{
    int data;
    Nod *succ, *pred;
};
```

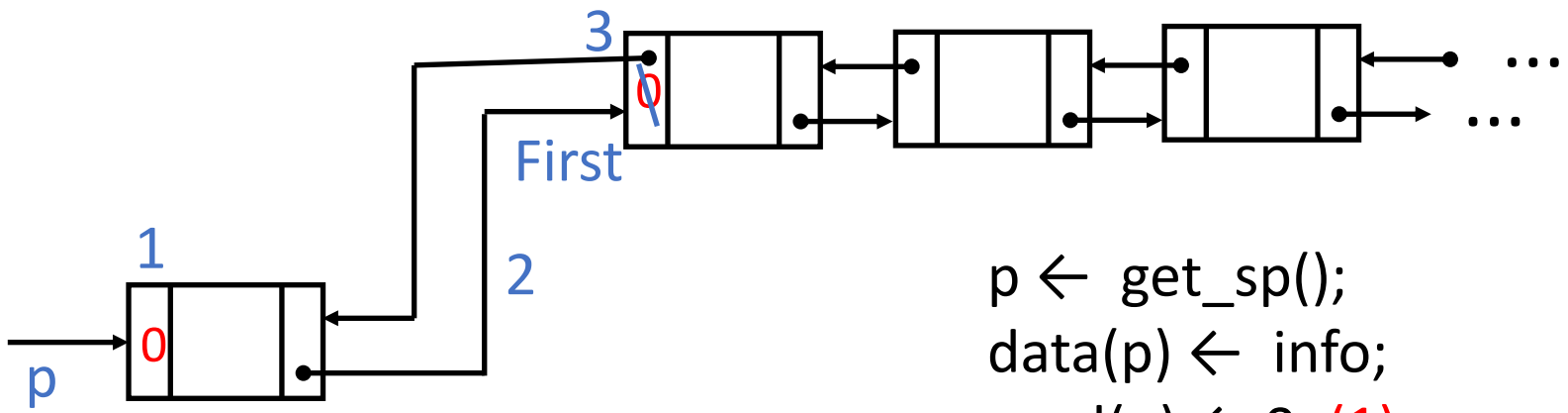
```
struct ListDoubleLinked{
    Nod *first, *last;
};
```

```
...main ...
ListDoubleLinked  LDL;
LDL.first=0;}
LDL.last=0; }//initializare
```

Operații cu liste dublu înlănțuite

Toate operațiile de la listele simplu înlănțuite se particularizează corespunzător.

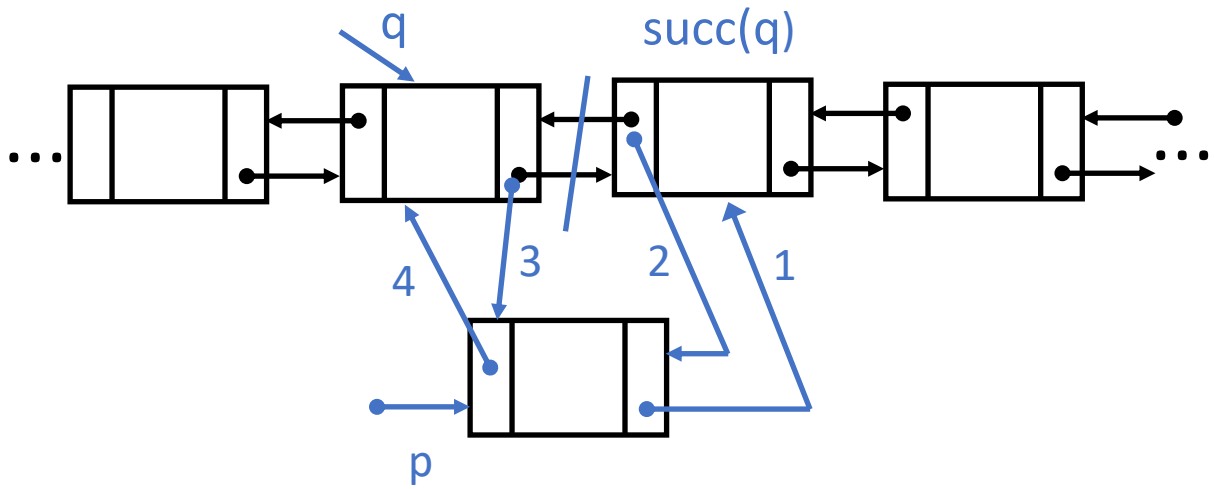
Inserarea în fața nodului first



p ← get_sp();
data(p) ← info;
pred(p) ← 0; (1)
succ(p) ← First; (2)
pred(First) ← p; (3)
First ← p (4)

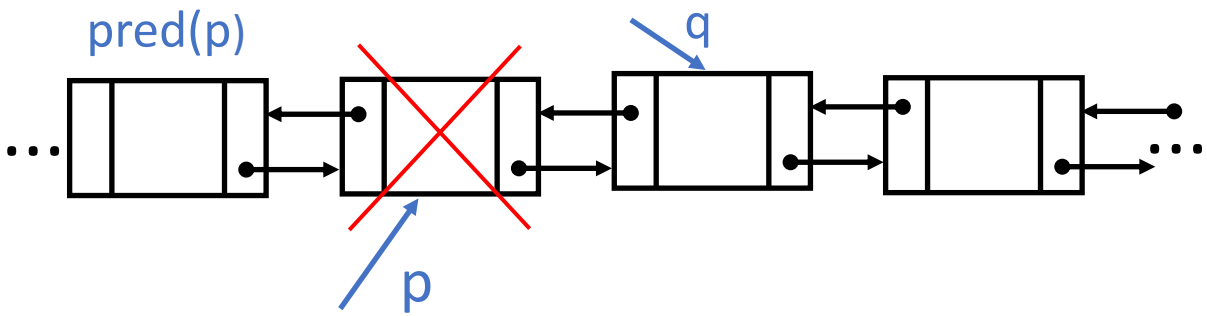
Operații cu liste dublu înlănțuite

Inserarea în interiorul listei



p ← get_sp();
data(p) ← info;
succ(p) ← succ(q); (1)
pred(succ(q)) ← p; (2)
pred(p) ← q; (3)
succ(q) ← p (4)

Ștergerea în interiorul listei



p ← pred(q);
succ(pred(p)) ← q; (1)
pred(q) ← pred(p); (2)
free_sp(p); (3)

LISTE – Aplicații

Reprezentarea polinoamelor

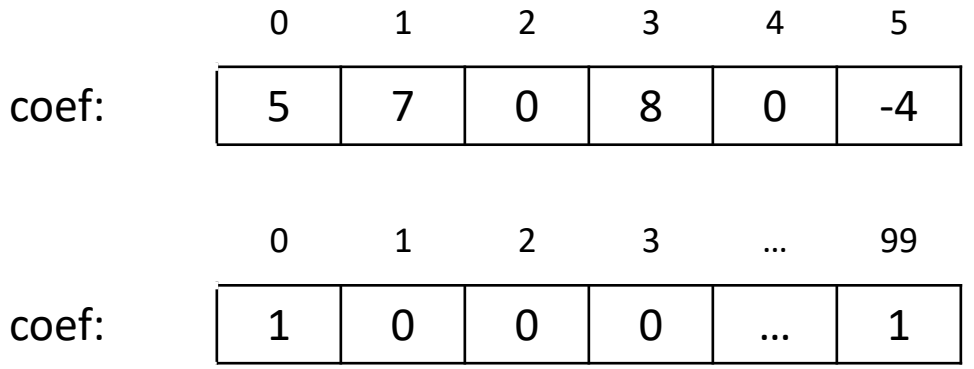
- Folosind tablouri:

```
struct polinom {
    int grad;
    int *coef;
};
```

$P_1(x) = 5 + 7x + 8x^3 - 4x^5$

$P_2(x) = 1 + x^{99}$

=> Utilizare ineficientă a spațiului!



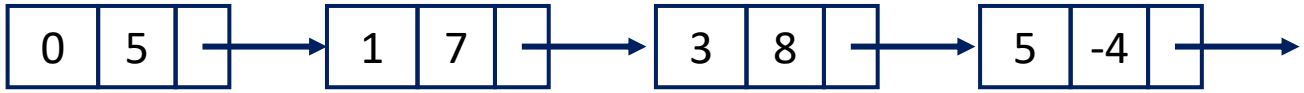
LISTE – Aplicații

Reprezentarea polinoamelor

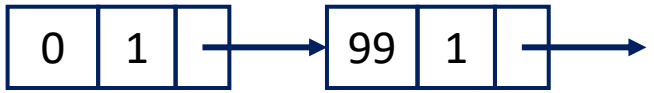
- Folosind liste:

```
struct Termen {
    int exponent;
    int coeficient;
    Termen *leg;
};
```

$P_1(x) = 5 + 7x + 8x^3 - 4x^5$



$P_2(x) = 1 + x^{99}$



LISTE – Aplicații

Reprezentarea polinoamelor

- Folosind liste:

Adunarea a 2 polinoame – problemă de interclasare a două liste

- se parcurg cele 2 polinoame
- se compară exponenții
  - Daca sunt egali, se adaugă un nou nod, cu suma coeficienților, în polinomul rezultat
  - Daca sunt diferiți,
    - se adaugă la rezultat nodul cu coef. cel mai mic
    - se parcurge lista polinomului în care s-a întâlnit nodul cu exponentul mai mic și se adaugă în listă toate nodurile, până se întâlnește un nod cu exponentul mai mare decât nodul curent din celălalt polinom

LISTE – Aplicații

Aritmetica numerelor foarte mari

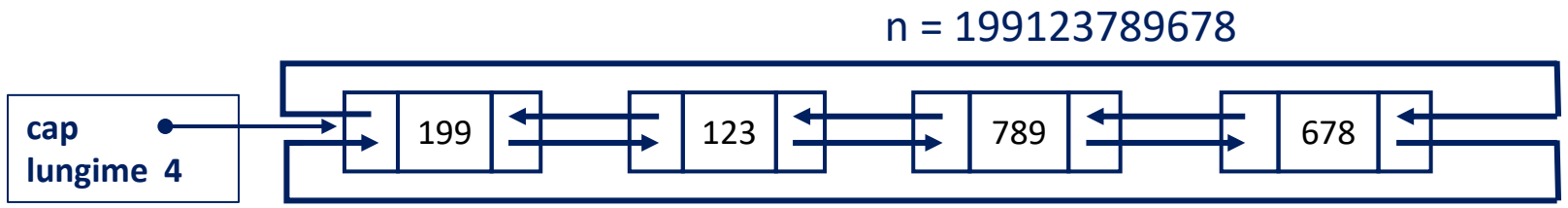
- Utilizate în matematica, cosmologie, astronomie, criptografie…

V1: unsigned int (32 biți): max ~4 mld (4 294 967 296)

V2: lista dublu înlănțuită circulară

```
struct nrFMare {
        int lungime;
        Element *cap;
};

struct Element {
        int trei_cifre; //sau 9 cifre
        Element *urm, *pred;
};
```



LISTE – Aplicații

Aritmetica numerelor foarte mari

- Compararea a două numere foarte mari:
  - Pe baza câmpurilor lungime ale celor două numere
  - Când sunt egale
    - se parcurg listele de la primul spre ultimul element și se compară

- Adunarea a două numere foarte mari:
  - Parcurgerea celor două liste se face de la ultimul spre primul adunând valorile corespunzătoare elementelor + transportul din suma anterioară

Dubla înlănțuire -> permite implementarea eficientă a ambelor parcurgeri

Circulară -> asigură accesul în timp constant la primul/ultimul element

LISTE - Aplicatii

Reprezentarea matricelor rare (sparse)

9	**0**	**0**	**0**	**0**
0	0	0	0	0
0	7	0	0	1
0	0	3	0	0
0	0	0	0	5

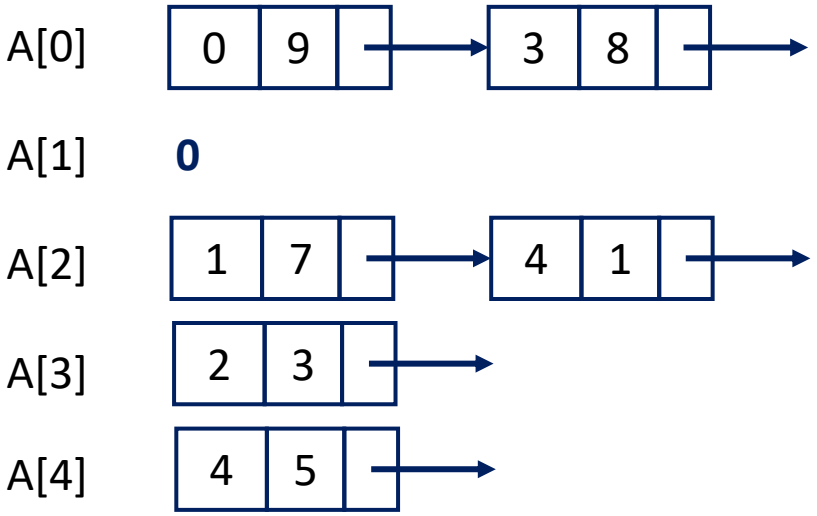
- Cu tablouri – utilizare ineficientă a spațiului

LISTE - Aplicatii

Reprezentarea matricelor rare (sparse)

9	**0**	**0**	**8**	**0**
0	0	0	0	0
0	7	0	0	1
0	0	3	0	0
0	0	0	0	5

- Tablou de liste:



LISTE – Aplicații

Managementul memoriei heap (alocarea dinamică de memorie)

- Pentru gestiunea memoriei heap, unele implementări ale funcțiilor malloc/new folosesc o listă dublu înlănțuită pentru a reprezenta memoria liberă.
  - În urma alocărilor/dealocărilor succesive, memoria heap se poate fragmenta
- Un nod în listă conține adresa de început și dimensiunea zonei libere contigue
- La alocare:
  - Se parcurg nodurile listei până se găsește o zonă de memorie cu dimensiunea cel putin egală cu cea cerută
  - Dacă zona găsită e mai mare, atunci se divide, iar partea neutilizată se înlănțuie cu celelalte noduri ale listei
  - Dacă nu se găsește o zonă adecvată, programul se termină cu eroare
- La dealocare:
  - Pentru zona eliberată se creează un nod care se înlănțuie în lista de zone libere
  - Se încearcă unirea nodurilor adiacente care împreună formează zone contigue de memorie (pt evitarea fragmentării excesive)