

# Liste

---

## CAPITOLUL VIII

# Cuprins

---

Introducere

Tipul de date abstract listă

Tehnici de implementare a listelor

Structuri de date derivate din TDA listă

Stive

Cozi

Multiliste

Exerciții

# Introducere

---

În capitolele anterioare am discutat despre tablouri.

Am văzut avantajele acestora, dar și limitările lor dintre care menționăm:

- nevoia de a cunoaște dimensiunea acestora
- nu putem efectua și inserție și ștergere cu complexitatea  $O(1)$ , aceste operații având o complexitate de  $O(n)$ , în cazul cel mai defavorabil

Aceste limitări pot fi depășite dacă folosim structuri înlănțuite

O structură înlănțuită este o colecție de noduri în care reținem informațiile utile și legături către alte noduri.

Astfel nodurile pot fi stocate oriunde în memorie, iar legătura către alt nod se face prin reținerea în cadrul structurii a adresei nodului spre care dorim să reținem legătura

# Tipul de date abstract listă

---

## Structura de date listă

- În cadrul structurilor de date avansate, structura listă ocupă un loc important.
- O listă, este o structură dinamică, care se definește pornind de la noțiunea de vector.
- Elementele unei liste sunt toate de același tip, ca atare o listă este o structură de date omogenă.
- Toate elementele unei liste sunt înregistrate în memoria centrală a sistemului de calcul.
- Spre deosebire de structura statică tablou la care se impune ca numărul componentelor să fie constant, în cazul listelor acest număr poate fi variabil, chiar nul.
- Listele sunt structuri flexibile particulare, care în funcție de necesități pot crește sau descrește și ale căror elemente pot fi referite, inserate sau șterse în orice poziție din cadrul listei.
- Două sau mai multe liste pot fi concatenate sau scindate în subliste

# Tipul de date abstract listă

---

## TDA Listă

- Din punct de vedere matematic, o listă este o secvență de zero sau mai multe elemente numite noduri aparținând unui anumit tip numit tip de bază.
- Formal, o listă se reprezintă de regulă ca o secvență de elemente:

$$a_1, a_2, \dots, a_n$$

- unde  $n \geq 0$  și fiecare  $a_i$  aparține tipului de bază.
- Numărul  $n$  al nodurilor se numește lungimea listei.
- Presupunând că  $n \geq 1$ , se spune că  $a_1$  este primul nod al listei iar  $a_n$  este ultimul nod.
- Dacă  $n = 0$  avem de-a face cu o listă vidă

# Tipul de date abstract listă

---

O proprietate importantă a unei liste este aceea că nodurile sale pot fi considerate ordonate liniar funcție de poziția lor în cadrul listei.

- De regulă, se spune că nodul  $a_i$  se află pe poziția  $i$ .
- Se spune că  $a_i$  precede pe  $a_{i+1}$  pentru  $i=1,2,\dots,n-1$ .
- Se spune că  $a_i$  succede (urmează) lui  $a_{i-1}$  pentru  $i=2,3,4, \dots,n$ .
- Este de asemenea convenabil să se postuleze existența poziției următoare ultimului element al listei.
- În această idee se introduce funcția  $\text{FIN}(L)$ : TipPozitie care returnează poziția următoare poziției  $n$  în lista  $L$  având  $n$  elemente.
- Se observă că  $\text{FIN}(L)$  are o distanță variabilă față de începutul listei, funcție de faptul că lista crește sau se reduce, în timp ce alte poziții au o distanță fixă față de începutul listei

# Tipul de date abstract listă

---

Ne amintim: Pentru a defini un tip de date abstract, în cazul de față TDA Listă, este necesară:

(1) Definirea din punct de vedere matematic a modelului asociat, definire precizată anterior.

(2) Definirea unui set de operatori aplicabili obiectelor de tip listă.

- Din păcate, pe de o parte este relativ greu de definit un set de operatori valabil în toate aplicațiile, iar pe de altă parte natura setului depinde esențial atât de maniera de implementare cât și de cea de utilizare a listelor.

- În continuare se prezintă două seturi reprezentative de operatori care acționează asupra listelor, unul restrâns și altul extins.

# Tipul de date abstract listă

---

TDA Listă (Set de operatori restrâns)

Modelul matematic: o secvență formată din zero sau mai multe elemente numite noduri toate încadrate într-un anumit tip numit tip de bază.

Notății:

L: TipLista;

p: TipPozitie;

x: TipNod.



# Tipul de date abstract listă

---

## Operatori:

**1.  $Fin(L:TipLista):TipPozitie$ ;** - operator care returnează poziția următoare ultimului nod al listei, adică poziția următoare sfârșitului ei. În cazul listei vide  $Fin(L)=0$ ;

**2.  $Insereaza(L:TipLista,x:TipNod,p:TipPozitie)$ ;** - inserează în lista  $L$ , nodul  $x$  în poziția  $p$ . Toate nodurile care urmează acestei poziții se mută cu un pas spre pozițiile superioare, astfel încât  $a_1,a_2,\dots,a_n$  devine  $a_1,a_2,\dots,a_{p-1},x,a_p,\dots,a_n$ . Dacă  $p$  este  $Fin(L)$  atunci lista devine  $a_1,a_2,\dots,a_n,x$ . Dacă  $p > Fin(L)$  rezultatul este nedefinit;

**3.  $Cauta(x:TipNod,L:TipLista):TipPozitie$ ;** - caută nodul  $x$  în lista  $L$  și returnează poziția nodului. Dacă  $x$  apare de mai multe ori, se furnizează poziția primei apariții. Dacă  $x$  nu apare de loc se returnează valoarea  $Fin(L)$ ;

**4.  $Furnizeaza(p:TipPozite,L:TipLista):TipNod$ ;** - operator care returnează nodul situat pe poziția  $p$  în lista  $L$ . Rezultatul este nedefinit dacă  $p=Fin(L)$  sau dacă în  $L$  nu există poziția  $p$ . Se precizează că tipul operatorului  $Furnizeaza$  trebuie să fie identic cu tipul de bază al listei;

# Tipul de date abstract listă

---

**5. *Suprima(p:TipPozite,L:TipLista)***; - suprimă elementul aflat pe poziția  $p$  în lista  $L$ . Dacă  $L$  este  $a_1, a_2, \dots, a_n$  atunci după suprimare  $L$  devine  $a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n$ . Rezultatul este nedefinit dacă  $L$  nu are poziția  $p$  sau dacă  $p = \text{Fin}(L)$ ;

**6. *Urmator(p:TipPozite,L:TipLista)***:TipPozitie; operator care returnează poziția următoare poziției  $p$  în cadrul listei. Dacă  $p$  este ultima poziție în  $L$  atunci  $\text{Urmator}(p, l) = \text{Fin}(L)$ . Urmator nu este definit pentru  $p = \text{Fin}(L)$ ;

**7. *Anterior(p:TipPozite,L:TipLista)***:TipPozitie; - operatori care returnează poziția anterioară poziției  $p$  în cadrul listei. Dacă  $p$  este prima poziție în  $L$  atunci Anterior nu este definit;

**8. *Initializeaza(L:TipLista)***:TipPozitie;- operator care face lista  $L$  vidă și returnează poziția  $\text{Fin}(L)=0$ ;

**9. *Primul(L:TipLista)***:TipPozitie; - returnează valoarea primei poziții în lista  $L$ . Dacă  $L$  este vidă, poziția returnată este  $\text{Fin}(L)=0$ ;

**10. *TraverseazaLista(L:TipLista,ProcesareNod(...))***; - parcurge nodurile listei  $L$  în ordinea în care apar ele în listă și aplică fiecăruia procedura *ProcesareNod*.

# Tipul de date abstract listă

---

TDA Listă (Set de operatori extins)

**Modelul matematic:** o secvență finită de noduri. Toate nodurile aparțin unui același tip numit tip de bază.

Fiecare nod constă din două părți: o parte de informații și o a doua parte conținând legătura la nodul următor. O variabilă specială indică primul nod al listei.

Notății:

- TipNod - tipul de bază;
- TipLista - tip indicator la tipul de bază;
- TipInfo - partea de informații a lui TipNod;
- TipIndicatorNod - tip indicator la tipul de bază (identic cu TipLista);
- incLista: TipLista - variabilă care indică începutul listei;
- curent,p,pnod: TipIndicatorNod - indică noduri în lista;
- element: TipNod;
- info: TipInfo; parte de informații a unui nod;
- b - valoare booleană; NULL - indicatorul vid.

# Tipul de date abstract listă

---

## Operatori:

- 1. *CreazaListaVida(inclista: TipLista);*** - variabila *inclista* devine NULL.
- 2. *ListaVida(inclista:TipLista):boolean;*** - operator care returnează TRUE dacă lista este vidă respectiv FALSE altfel.
- 3. *Primul(inclista:TipLista,curent:TipIndicatorNod);*** - operator care face ca variabila *curent* să indice primul nod al listei precizată de *inclista*.
- 4. *Ultimul(curent: TipIndicatorNod): boolean;*** - operator care returnează TRUE dacă *curent* indică ultimul element al listei.
- 5. *InserInceput(inclista: TipLista,pnod: TipIndicatorNod);*** - inserează la începutul listei *inclista* nodul indicat de *pnod*.

# Tipul de date abstract listă

---

**6. *InserDupa(curent, pnod: TipIndicatorNod);*** - inserează nodul indicat de pnod după nodul indicat de curent. Se presupune că curent indică un nod din listă.

**7. *InserInFatza(curent, pnod: TipIndicatorNod);*** - insertie în fața nodului curent.

**8. *SuprimaPrimul(incLista: TipLista);*** - suprimă primul nod al listei incLista.

**9. *SuprimaUrm(curent: TipIndicatorNod);*** - suprimă nodul următor celui indicat de curent.

**10. *SuprimaCurent(curent: TipIndicatorNod);*** - suprimă nodul indicat de curent.

**11. *Urmatorul(curent: TipIndicatorNod);*** - curent se poziționează pe următorul nod al listei. Dacă curent indică ultimul nod al listei el va deveni NULL.

**12. *Anterior(curent: TipIndicatorNod);*** - curent se poziționează pe nodul anterior celui curent.

**13. *MemoreazaInfo(curent: TipIndicatorNod, info: TipInfo);*** - atribuie nodului indicat de curent informația info.

# Tipul de date abstract listă

---

- 14. **MemoreazaLeg(curent, p: TipIndicatorNod);** - atribuie câmpului urm (de legătură) al nodului indicat de curent valoarea p.*
- 15. **FurnizeazaInfo(curent: TipIndicatorNod): TipInfo;** - returnează partea de informație a nodului indicat de curent.*
- 16. **FurnizeazaUrm(curent: TipIndicatorNod): TipIndicator Nod;** - returnează legătura nodului curent (valoarea câmpului urm).*
- 17. **TraverseazaLista(inclista: TipLista, ProcesareNod (...));** - parcurge nodurile listei L în ordinea în care apar ele în listă și aplică fiecăruia procedura ProcesareNod.*

# Tehnici de implementare a listelor

---

## Tehnici de implementare a listelor

- De regulă pentru structurile de date fundamentale există construcții de limbaj care le reprezintă, construcții care își găsesc un anumit corespondent în particularitățile hardware ale sistemelor care le implementează.
- Pentru structurile de date avansate însă, care se caracterizează printr-un nivel mai înalt de abstractizare, acest lucru nu mai este valabil.
- De regulă, reprezentarea structurilor de date avansate se realizează cu ajutorul structurilor de date fundamentale, observație valabilă și pentru structura listă.
- Din acest motiv, în cadrul acestui paragraf vor fi prezentate câteva dintre structurile de date fundamentale care pot fi utilizate în reprezentare listelor.
- Funcțiile care implementează operatorii specifici prelucrării listelor vor fi descriși în termenii acestor structuri.

# Tehnici de implementare a listelor

---

## Implementarea listelor cu ajutorul structurii **tablou**

- În cazul implementării listelor cu ajutorul structurii tablou:
- O listă se asimilează cu un tablou.
- Nodurile listei sunt memorate într-o zonă contiguă în locații succesive de memorie.
- În această reprezentare:
  - O listă poate fi ușor traversată.
  - Noile noduri pot fi adăugate în mod simplu la sfârșitul listei.
  - Inserția unui nod în mijlocul listei presupune însă deplasarea tuturor nodurilor următoare cu o poziție spre sfârșitul listei pentru a face loc noului nod.
  - Suprimarea oricărui nod cu excepția ultimului, presupune de asemenea deplasarea tuturor celorlalte în vederea eliminării spațiului creat.
  - Inserția și suprimarea unui nod necesită un efort de execuție  $O(n)$



# Tehnici de implementare a listelor

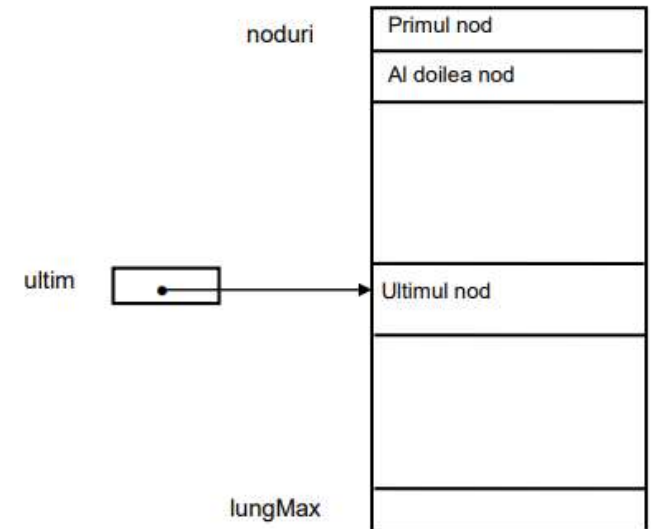
În implementarea bazată pe tablouri, TipLista se definește ca o structură (articol) cu două câmpuri.

(1) **Primul câmp** este un tablou numit **noduri**, cu elemente de TipNod.

- Lungimea acestui tablou este astfel aleasă de către programator încât să fie suficientă pentru a putea păstra cea mai mare dimensiune de listă ce poate apare în respectiva aplicație.

(2) **Cel de-al doilea câmp** este un indicator (**ultim**) care indică poziția în tablou a ultimului nod al listei.

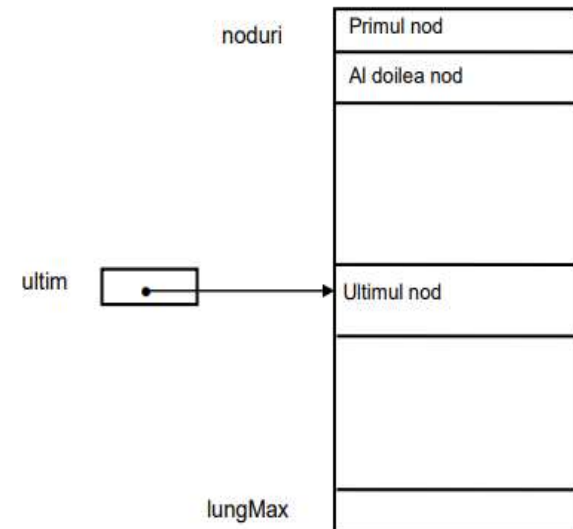
- Cel de-al  $i$ -lea nod al listei se găsește în cel de-al  $i$ -lea element al tabloului, pentru  $1 \leq i \leq \text{ultim}$ .
- Poziția în cadrul listei se reprezintă prin valori întregi, respectiv c de-a  $i$ -a poziție prin valoarea  $i$ .
- Funcția  $\text{Fin}(L)$  returnează valoarea  $\text{ultim}+1$ .



# Tehnici de implementare a listelor

## Variantă implementare C

```
#define lung_max = 100  
typedef struct {  
    tip_nod noduri[lung_max]; //tabloul de elemente  
    tip_indice ultim; //nr efectiv de elemente = ultim+1  
} tip_lista;  
typedef tip_indice tip_pozitie;
```



# Tehnici de implementare a listelor

---

În continuare se vor prezenta câte o variantă de implementare pentru operațiile specifice setului restrâns de operatori: Insereaza, Suprima și Cauta utilizând implementarea bazată pe tablouri a listelor.

- Se fac următoarele precizări:
- Dacă se încearcă inserția unui nod într-o listă care deja a utilizat în întregime tabloul asociat se semnalează un mesaj de eroare.
- Dacă în cursul procesului de căutare nu se găsește elementul căutat, Cauta returnează poziția -1.
- S-a prevăzut parametrul boolean er, care în caz de eroare se returnează cu valoarea adevărat și care poate fi utilizat pentru tratarea erorii sau pentru întreruperea execuției programului

# Tehnici de implementare a listelor

---

```
void insereaza(tiplista* l, tipnod x, tip_pozitie p, boolean* er)
/*plasează pe x în poziția p a listei; performanța O(n)*/
{ tip_pozitie q; *er = false;
  if (l->ultim >= (lung_max-1)) {
    *er = true;
    printf("lista este plina");
  }
  else
    if ((p > l->ultim +1 ) || (p < 0)) {
      *er = true;
      printf("pozie invalida");
    }
  else {
    for (q = l->ultim+1; q > p; q--)
      l->noduri[q] = l->noduri[q - 1]; //se face loc pentru noul element
    l->noduri[p] = x; //se insereaza elementul
    l->ultim++; //se incrementeaza ultima pozitie, si implicit numarul de elemente
  }
}
```

# Tehnici de implementare a listelor

---

```
void suprima(tiplista* l, tip_pozitie p, boolean *er)
/*extrage elementul din poziția p a listei*/
{
    tip_pozitie q; /*performanța O(n)*/
    *er = false;
    if ((p > l->ultim + 1) || (p < 0)) {
        *er = true;
        printf("poziție invalidă");
    }
    else {
        for (q = p; q <= l->ultim; q++)
            l->noduri[q] = l->noduri[q+1];
        l->ultim = l->ultim - 1;
    }
} /*suprima*/
```

//stergem prin copiere la stanga  
//actualizam ultima pozitie

# Tehnici de implementare a listelor

---

```
tip_pozitie cauta(tipnod x, tiplista l)
{
    /*returnează poziția lui x în listă*/
    tip_pozitie q;
    boolean gasit;
    tip_pozitie cauta_result;
    q = 0; gasit = false; /*performanța O(n)*/
    do {
        if (l.noduri[q].info == x.info) { //am gasit elementul
            cauta_result = q;
            gasit = true;
        }
        q = q + 1;
    } while (!(gasit || (q == l.ultim + 1)));
    if (!gasit) cauta_result = -1; //nu s-a gasit elementul
    return cauta_result;
}
```

# Tehnici de implementare a listelor

---

**A doua variantă clasică** de implementare a listelor este cu ajutorul **pointerilor**, folosind alocarea dinamică

O listă înlănțuită este constituită dintr-o serie de obiecte, numite nodurile listei

În acest caz un nod este un obiect de sine stătător, spre deosebire de situația precedentă, în care nodul era reprezentat de o intrare într-un tablou

Deoarece o listă liniară este o structură dinamică ea poate fi definită în termeni recursivi după cum urmează:

```
typedef int tip_info;  
typedef struct {  
    int cheie;  
    tip_info info;  
    struct tip_nod* urm;  
} tip_nod;  
typedef tip_nod * tip_lista;
```

# Tehnici de implementare a listelor

După cum se observă, în cazul definirii unui nod al structurii listă înlănțuită s-au pus în evidență trei câmpuri:

- O cheie care servește la identificarea nodului.
- Un câmp info conținând informația utilă.
- Un pointer de înlănțuire la nodul următor.

Fiecare nod conține o legătură către un alt nod sau valoarea NULL (dacă este vorba de ultimul nod din listă)

O astfel de secvență de noduri formează o listă înlănțuită. Dacă fiecare nod conține doar un câmp de legătură către succesorul său din listă, atunci lista se numește simplu înlănțuită.

În acest caz întreaga listă poate fi reținută, prin folosirea unei singure variabile care să indice începutul listei. În exemplul de mai jos variabila se numește prim:



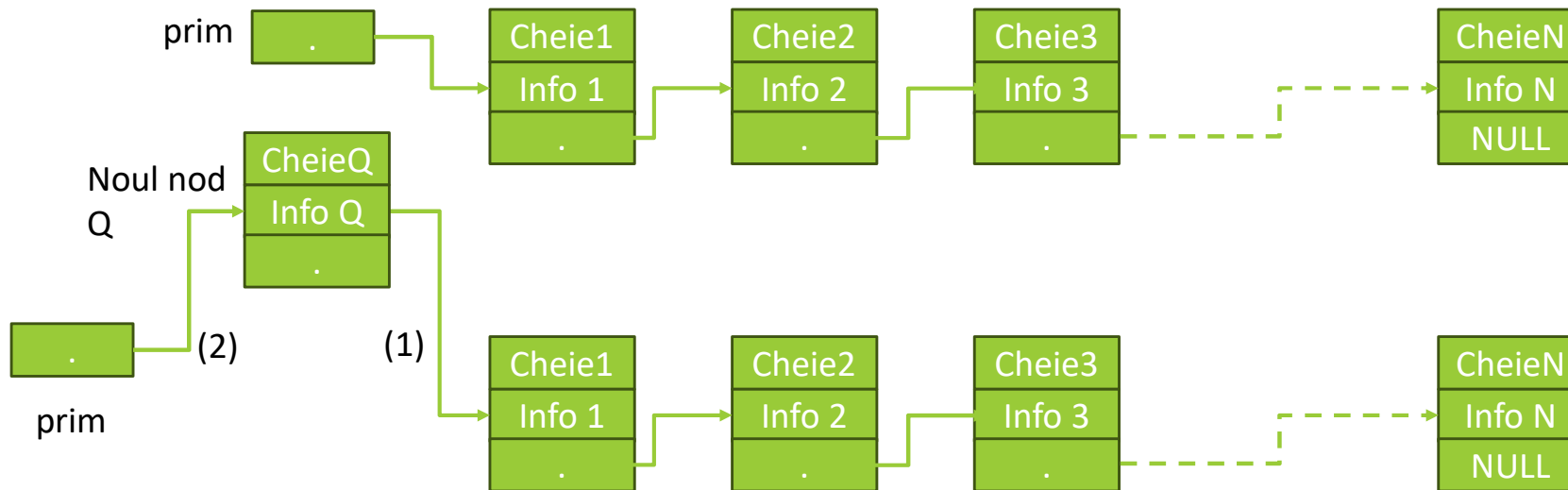


# Tehnici de implementare a listelor

Tehnici de inserție a nodurilor și de creare a listelor înlănțuite

Nodurile se pot insera în diferite poziții în listă

Cea mai simplă variantă de inserție o reprezintă inserția în fața listei



# Tehnici de implementare a listelor

---

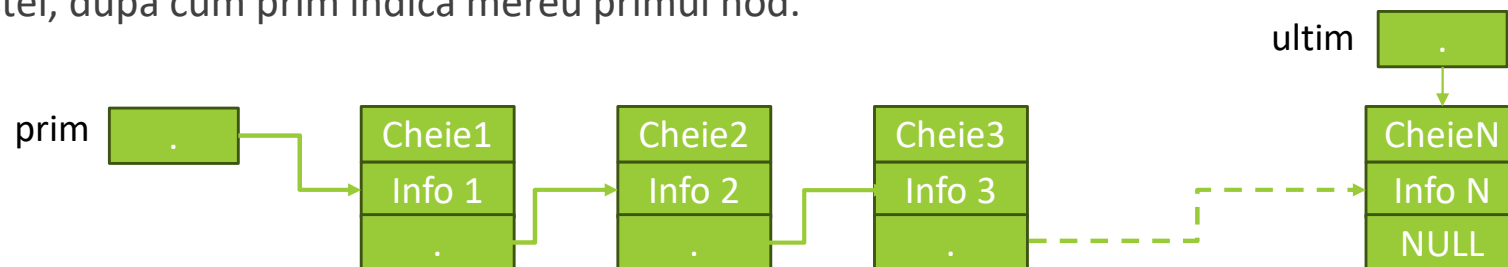
```
tip_lista insertie_fata(int cheie, tip_info info)
{tip_nod* q; //noul nod
  q = (tip_nod*)malloc(sizeof(tip_nod)); // aloc spatiu pentru noul nod
  if (q) //daca s-a alocat spatiu cu succes
  {
    q->cheie = cheie;
    q->info = info; //asignez valorile primite ca parametru
    q->urm = prim; //facem legatura catre inceputul listei
    prim = q; //noul nod devine primul nod;
  }
  return prim;
}

int main()
{ tip_lista prim = NULL; //initializare lista vida
  prim = insertie_fata(1, 1); prim = insertie_fata(2, 2); //apeluri functie
  return 0;
}
```

# Tehnici de implementare a listelor

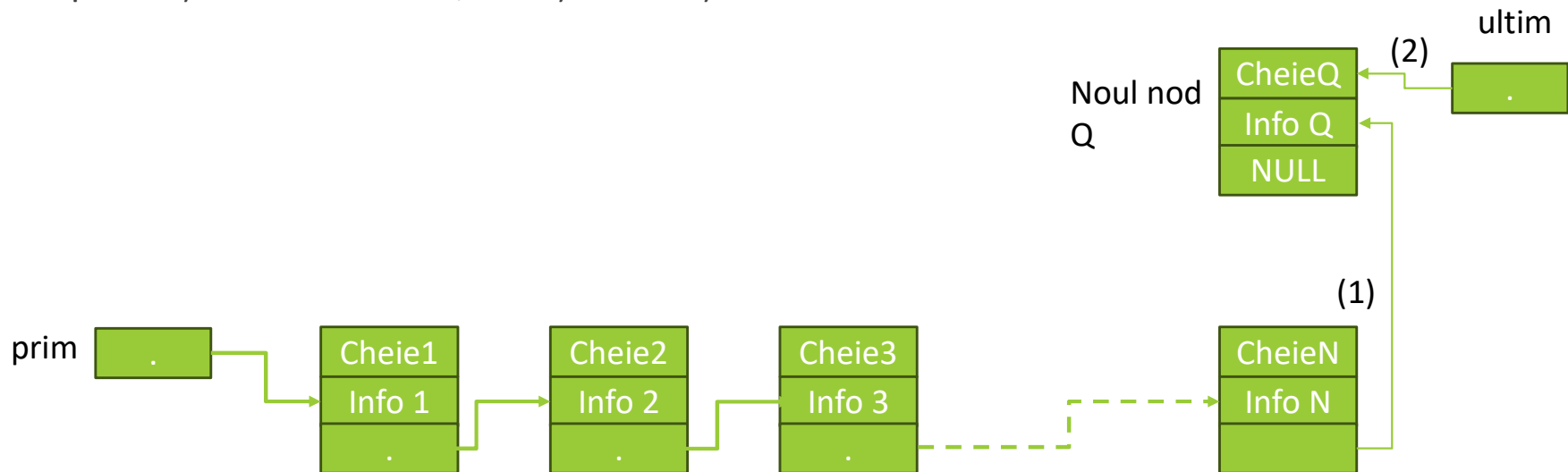
Dacă se dorește crearea listei în ordine naturală, atunci este nevoie de o secvență care inserează un nod la sfârșitul unei liste.

- Această secvență de program se redactează mai simplu dacă se cunoaște locația ultimului nod al listei.
- Teoretic lucrul acesta nu prezintă nici o dificultate, deoarece se poate parcurge lista de la începutul ei (indicat prin inceput) până la detectarea nodului care are câmpul urm = NULL.
- În practică această soluție nu este convenabilă, deoarece parcurgerea de fiecare dată a întregii liste este ineficientă.
- Se preferă să se lucreze cu o variabilă pointer ajutătoare ultim care indică mereu ultimul nod al listei, după cum prim indică mereu primul nod.



# Tehnici de implementare a listelor

În prezența variabilei ultim, inserția la sfârșitul listei devine:



# Tehnici de implementare a listelor

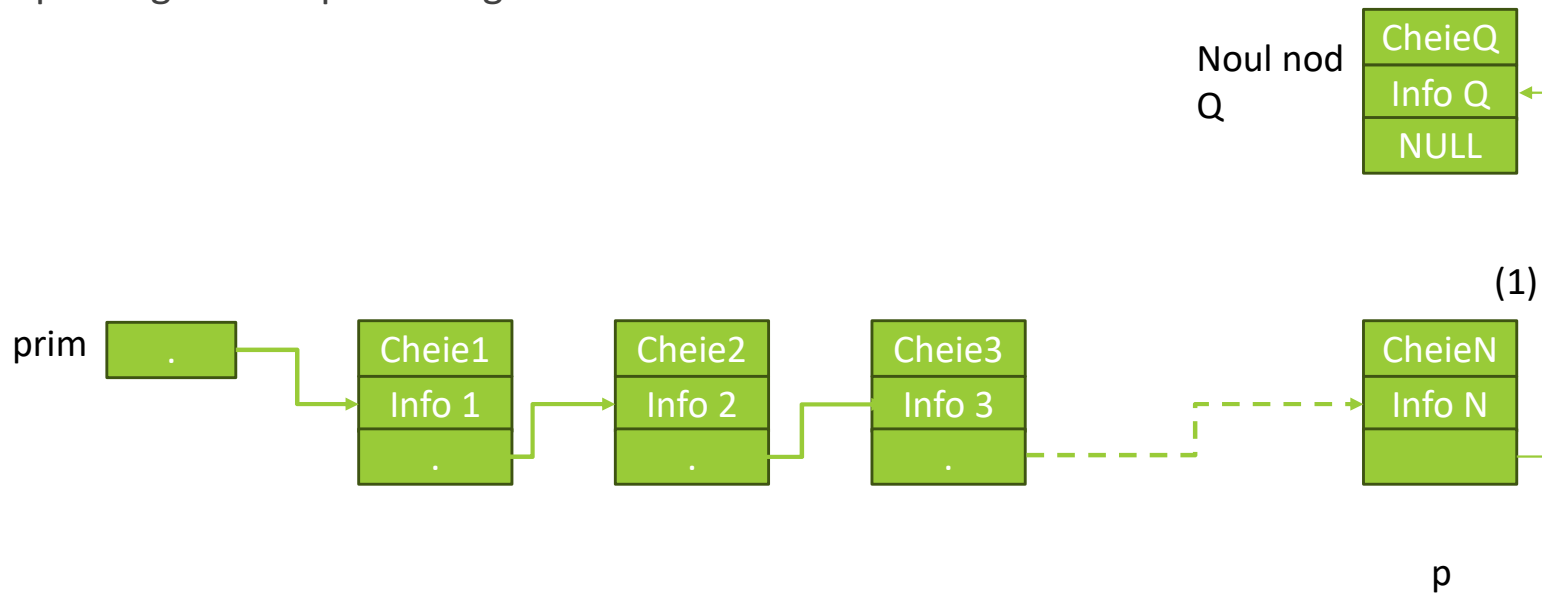
---

```
tip_lista insertie_final(int cheie, tip_info info)
{ tip_nod* q; //noul nod
  q = (tip_nod*)malloc(sizeof(tip_nod)); // aloc spatiu pentru noul nod
  if (q) //daca s-a alocat spatiu cu succes
  {
    q->cheie = cheie;
    q->info = info; //asignez valorile primite ca parametru
    q->urm = NULL;
    if (ultim == NULL) //daca lista a fost vida
      prim = q;
    else ultim->urm= q;      //ultim indica spre q;
    ultim = q; //q devine ultimul nod
  }
  return prim;
}
```

Cum se modifică funcția `insertie_fata`, dacă reținem și ultimul element?

# Tehnici de implementare a listelor

În lipsa variabilei ultim, inserția la sfârșitul listei se folosește de o altă variabilă (p) cu care parcurgem lista pentru a găsi ultimul element



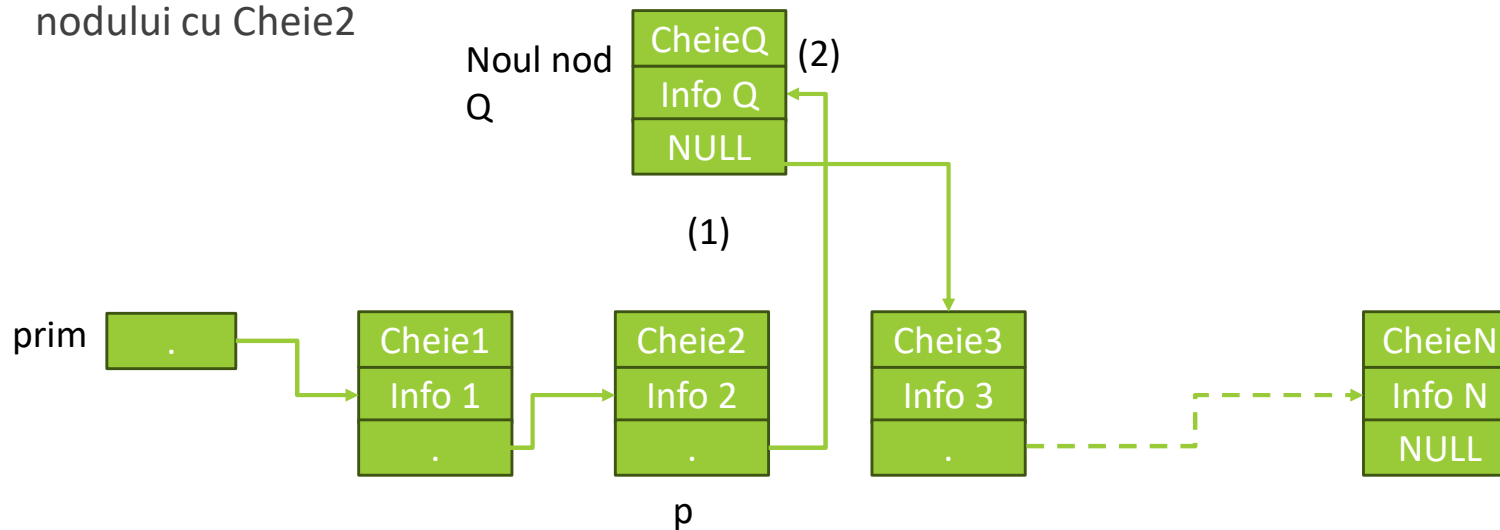
# Tehnici de implementare a listelor

---

```
tip_lista insertie_final2(int cheie, tip_info info)
{tip_nod* q; //noul nod
  q = (tip_nod*)malloc(sizeof(tip_nod)); // aloc spatiu pentru noul nod
  if (q) //daca s-a alocat spatiu cu succes
  {
    q->cheie = cheie;
    q->info = info; //asignez valorile primite ca parametru
    q->urm = NULL;
  }
  tip_nod* p;
  if (prim == NULL) //daca lista era vida
    prim = q;
  else {                //parcurs lista pana la ultimul element
    for (p = prim; p->urm != NULL; p = p->urm);
    p->urm = q;
  }
  return prim;
}
```

# Tehnici de implementare a listelor

În continuare se descrie inserția unui nod nou într-un loc oarecare al unei liste  
Dacă dorim o inserție după nodul cu Cheie2, va trebui să parcurgem lista și să căutăm adresa nodului cu Cheie2





# Tehnici de implementare a listelor

---

```
tip_lista insertie_dupa(int cheie, tip_info info, tip_nod *p)
{
    tip_nod* q; //noul nod
    if (p == NULL) //daca se doreste de fapt insertie in fata listei
        prim = insertie_fata(cheie, info);
    else if (p == ultim) //daca se doreste de fapt adaugare la sfarsit
        prim = insertie_final(cheie,info);
    else
    {
        q = (tip_nod*)malloc(sizeof(tip_nod)); // aloc spatiu pentru noul nod
        if (q) //daca s-a alocat spatiu cu succes
        {
            q->cheie = cheie;
            q->info = info; //asignez valorile primite ca parametru
            q->urm = p->urm; //fac legatura de la noul nod la succesorul sau
        }
        p->urm = q;          //fac legatura catre noul nod
    }
    return prim;
}
```

# Tehnici de implementare a listelor

---

Pentru a găsi poziția nodului p în listă ne-am folosit de o funcție de căutare

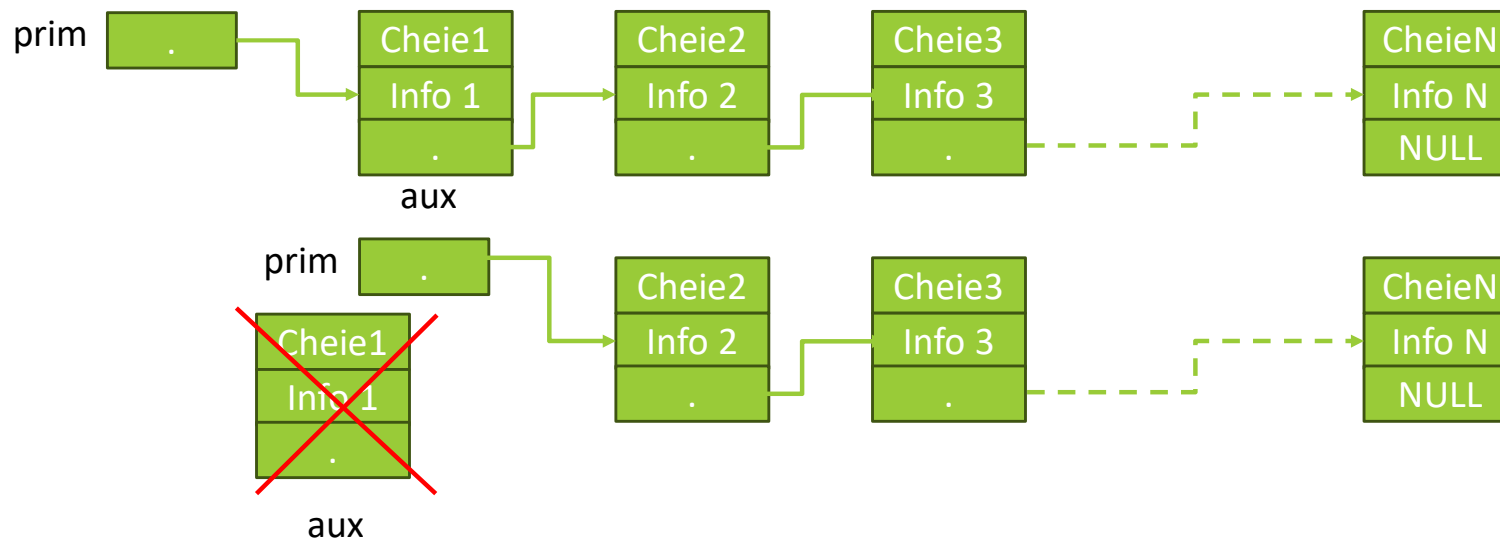
```
tip_nod* cauta(int cheie)
{
    tip_nod* p;
    /*cat timp nu am gasit cheia si nu am ajuns la capatul listei
    avansez in lista */
    for (p = prim; (p != NULL)&&(p->cheie != cheie); p = p->urm);
    /*atentie: conteaza ordinea celor doua contitii, deoarece in C
    expresiile logice se evalueaza in scurtcircuit*/
    return p; //functia returneaza NULL in caz ca nu a gasit cheia
}
```

# Tehnici de implementare a listelor

Tehnici de suprimare a nodurilor:

În funcție de poziția nodului de suprimat, în listă avem mai multe situații ce vor fi tratate în mod diferit

Cea mai simplă situație este când avem de șters un nod aflat la începutul listei



# Tehnici de implementare a listelor

---

În cazul **ștergerii primului element**, se execută următorii pași în ordine:

1. Se reține primul element într-o variabilă auxiliară
2. Se actualizează variabila prim, care va indica nodul următor (prim->urm)
3. Se dealocă spațiul pentru elementul reținut în variabila auxiliară

Toate cele trei operații implică un timp constant de execuție și nu depind de lungimea listei, astfel complexitatea acestei ștergeri este  $O(1)$

Pentru ca algoritmul să funcționeze în toate cazurile, este nevoie să adăugăm o verificare suplimentară ca lista să nu fie vidă. Dacă lista este vidă nu facem nimic.

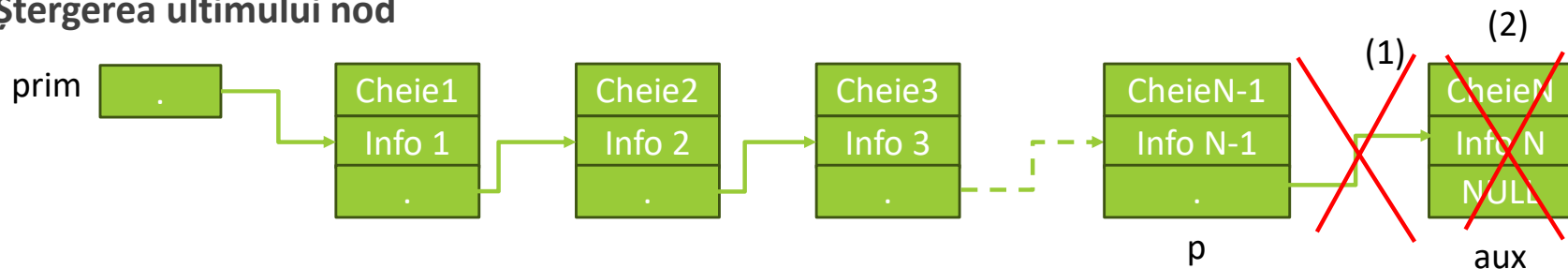
**0.** Se verifică dacă lista este vidă și dacă da, nu se mai execută niciunul din pașii următori

1. Se reține primul element într-o variabilă auxiliară
2. Se actualizează variabila prim, care va indica nodul următor (prim->urm)
3. Se dealocă spațiul pentru elementul reținut în variabila auxiliară

Ce se întâmplă dacă lista are un singur nod? Dacă reținem și ultimul nod, cum se schimbă algoritmul?

# Tehnici de implementare a listelor

## Ștergerea ultimului nod



În cazul ștergerii ultimului element, se execută următorii pași în ordine:

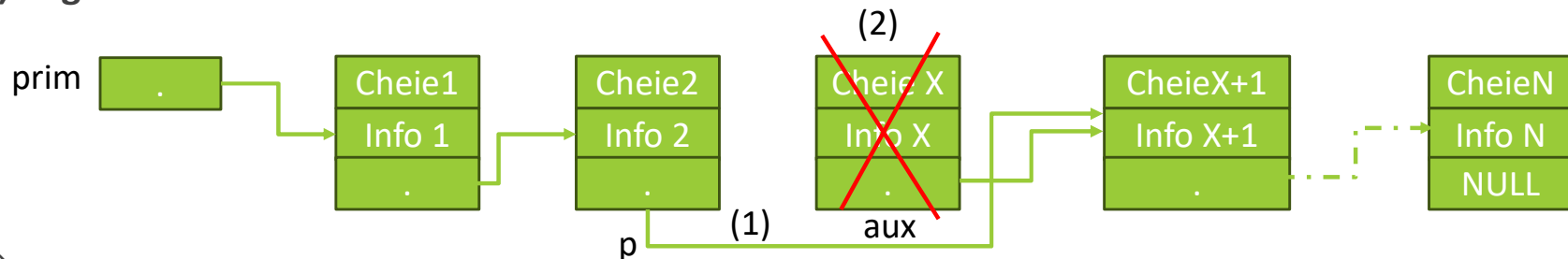
1. Se parcurge lista cu două variabile și se reține atât ultimul nod, cât și predecesorul său
2. Se actualizează câmpul următor pentru predecesorul nodului de șters ca fiind NULL
3. Se dealocă spațiul pentru elementul reținut în variabila auxiliară
- (4.) Dacă avem o listă pentru care se reține și ultimul element, atunci acesta se actualizează cu valoarea predecesorului nodului de șters.

Găsirea nodului de șters presupune o parcurgere a listei, astfel algoritmul are o complexitate de  $O(n)$ .

ATENȚIE: Și în acest caz trebuie verificat dacă lista este vidă!

# Tehnici de implementare a listelor

## Ștergerea unui nod oarecare



În cazul ștergerii unui nod oarecare, se procedează asemănător cu situația în care se șterge ultimul nod, după ce se verifică dacă lista nu este vidă:

1. Se parcurge lista cu două variabile și se reține atât nodul de șters, cât și predecesorul său, în cazul în care predecesorul nu există avem o situație de ștergere a primului element
2. Se actualizează câmpul următor pentru predecesorul nodului de șters ca fiind nodul următor nodului de șters
3. Se dealocă spațiul pentru elementul reținut în variabila auxiliară

Găsirea nodului de șters presupune, cazul cel mai defavorabil, o parcurgere a întregii liste, astfel algoritmul are o complexitate de  $O(n)$ .

# Tehnici de implementare a listelor

---

Traversarea unei liste înlănțuite. **Căutarea** unui nod într-o listă înlănțuită

Prin traversarea unei liste se înțelege executarea în manieră ordonată a unei anumite operații asupra tuturor nodurilor listei.

- Fie pointerul prim care indică primul nod al listei și fie curent o variabilă pointer auxiliară.
- Dacă curent este un nod oarecare al listei se notează cu Prelucrare(curent) operația amintită, a cărei natură nu se precizează.

O operație care apare frecvent în practică, este căutarea adică depistarea unui nod care are cheie egală cu o valoare dată x

- Căutarea este de fapt o traversare cu caracter special a unei liste

```
curent = prim;  
while ((curent != NULL) && (curent->cheie != x))  
    curent = curent->urm;  
return curent;
```

# Tehnici de implementare a listelor

---

## Implementarea listelor cu ajutorul cursorilor

- În anumite limbaje de programare ca și FORTRAN sau ALGOL **nu** există definit tipul pointer.
- De asemenea, în anumite situații (de exemplu pe microcontrolerele cu resurse limitate) este mai avantajos pentru programator din punctul de vedere al performanței codului implementat să evite utilizarea pointerilor
- În astfel de cazuri, pointerii pot fi simulați cu ajutorul cursorilor care sunt valori întregi ce indică poziții în tablouri.



# Tehnici de implementare a listelor

---

În accepțiunea acestei implementări:

- (1) Pentru toate listele ale căror noduri sunt de TipNod, se crează un tablou (Zona) având elementele de TipElement.
- (2) Fiecare element al tabloului conține un câmp nodLista:TipNod și un câmp urm:TipCursor definit ca **subdomeniu** al tipului întreg.
- (3) TipLista este în aceste condiții identic cu

# Tehnici de implementare a listelor

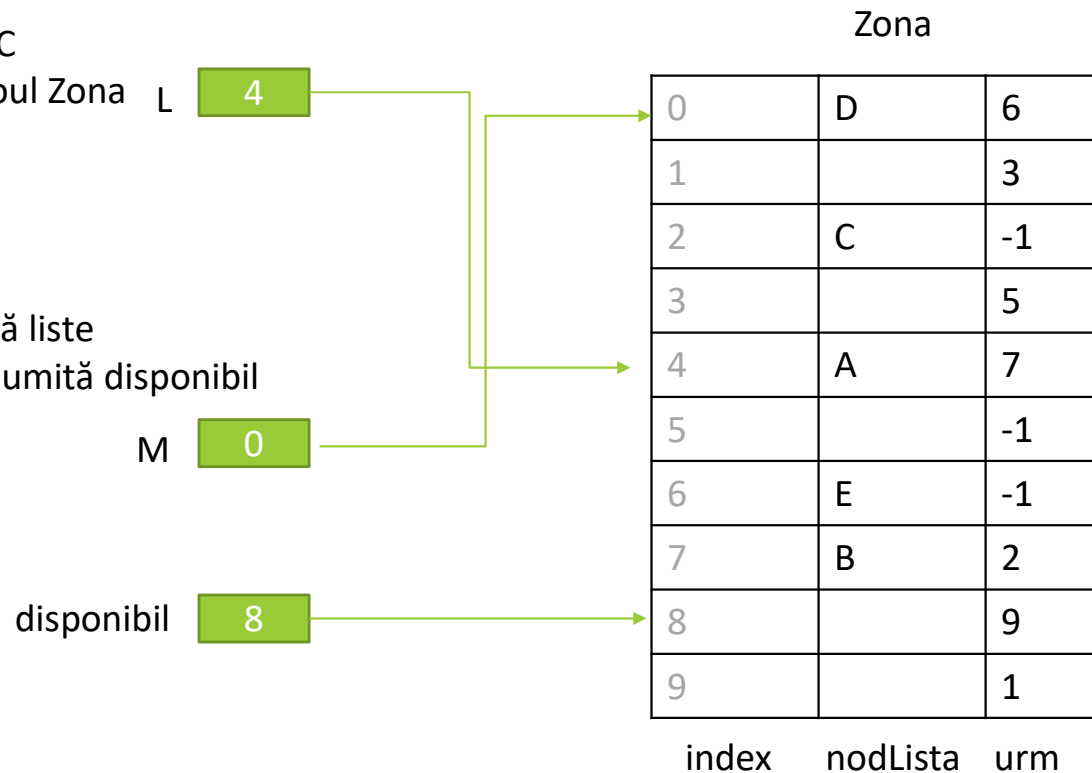
---

```
enum { lung_max = 10}; /*implementarea listelor cu ajutorul cursorilor*/
typedef unsigned char tip_cursor;
typedef unsigned boolean;
#define true (1)
#define false (0)
typedef int tip_nod;
typedef struct tip_element {
    tip_nod nod_lista;
    tip_cursor urm;
} tip_element
typedef tip_cursor tip_lista;
tip_element zona[lung_max];
tip_lista L,M,disponibil;
```

# Tehnici de implementare a listelor

Exemplu de două liste L = A, B, C  
și M = D, E care partajează tabloul Zona  
cu lungimea maximă 10.

Acele locații ale tabloului care  
**nu** apar în nici una din cele două liste  
sunt înlănțuite într-o altă listă numită disponibil



# Tehnici de implementare a listelor

---

Spre exemplu dacă `L.TipLista` este un cursor care indică începutul unei liste atunci:

- Valoarea lui `Zona[L].nodLista` reprezintă **primul** nod al listei `L`.
- `Zona[L].urm` este indexul (cursorul) care indică cel de-al doilea element al listei `L`, ș.a.m.d.
- Valoarea `-1` a unui cursor, semnifică legătura vidă, adică faptul că nu urmează nici un element

Se observă că:

- Acele locații ale tabloului care **nu** apar în nici una din cele două liste sunt înlănțuite într-o altă listă numită disponibil.
- Lista disponibil este utilizată fie:
  - Pentru a **furniza o nouă locație** în vederea realizării unei **insertii**.
  - Pentru a **depozita o locație** a tabloului rezultată din **suprimarea** unui nod în vederea unei reutilizări ulterioare

# Tehnici de implementare a listelor

---

Pentru a **insera** un nod  $x$  în lista  $L$ :

- (1) Se suprimă prima locație a listei disponibil.
- (2) Se inserează această locație în poziția dorită a listei  $L$ , actualizând valorile cursorilor implicați.
- (3) Se asignează câmpul `nodLista` al acestei locații cu valoarea lui  $x$ .

- **Suprimarea** unui element  $x$  din lista  $L$  presupune:

- (1) Suprimarea locației care îl conține pe  $x$  din lista  $L$ .
- (2) Inserția acestei locații în capul listei disponibil.

# Tehnici de implementare a listelor

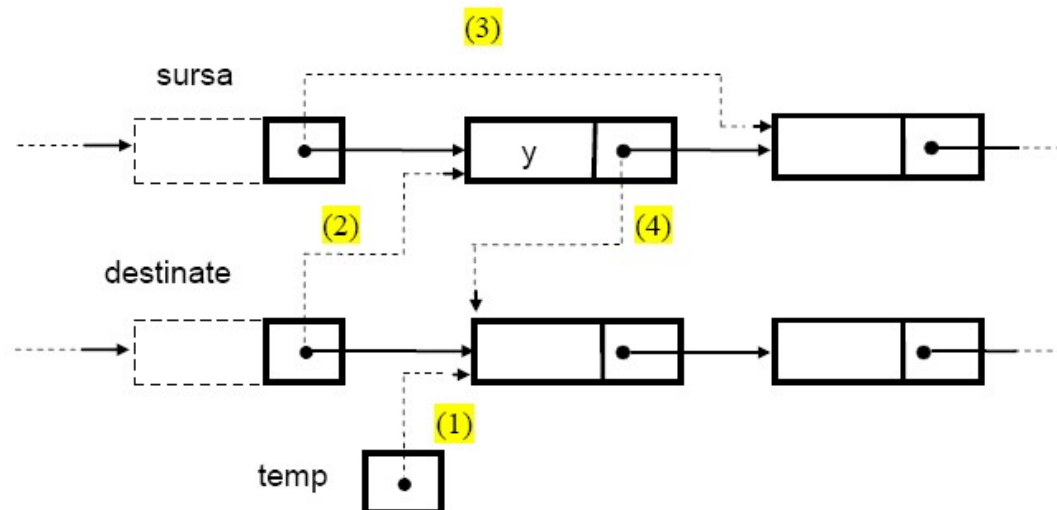
---

Atât **inserția** cât și **suprimarea** sunt de fapt cazuri speciale ale următoarei **situații**:

- Fie două liste precizate prin cursorii sursa și destinație.
- Fie y prima locație a listei indicate de sursa.
- Se **suprimă** y din lista indicată de sursa și se **înlănțuie** pe prima poziție a listei indicate de destinație. Acest lucru se realizează astfel:
  - (1) Se salvează valoarea cursorului destinație în locația auxiliară temp.
  - (2) Se atribuie valoarea cursorului sursa cursorului destinație care astfel îl va indica pe y.
  - (3) Se atribuie lui sursa valoarea legăturii lui y.
  - (4) Legătura lui y se asignează cu fosta valoare a lui destinație

# Tehnici de implementare a listelor

Reprezentarea grafică a acțiunilor anterioare este reprezentată mai jos, unde sunt prezentate legăturile înainte (linie continuă) și după (linie întreruptă) desfășurarea ei.



# Tehnici de implementare a listelor

---

```
boolean muta(tip_cursor* sursa,tip_cursor* destinatie)
/*mută elementul indicat de sursă în fața celui indicat de destinație*/
{tip_cursor temp;
boolean muta_result;
if (*sursa==0) /*performanța O(1)*/
{
    printf("locația nu există");
    muta_result= false;
}
else
{
    temp= *destinatie;
    *destinatie=*sursa;
    *sursa= zona[*destinatie-1].urm;
    zona[*destinatie-1].urm= temp;
    muta_result= true;
}
return muta_result;
}
```



# Tehnici de implementare a listelor

---

```
void insereaza(tip_nod x, tip_cursor p, tip_cursor* inceput)
{ /*performanța O(1)*/
  if (p==0){ /*se inserează pe prima poziție*/
    if (muta(&disponibil,inceput))
      zona[*inceput-1].nodlista= x;
  }
  else /*se inserează într-o poziție diferită de prima*/
    if (muta(&disponibil,&zona[p-1].urm))
      /*locația pentru x va fi indicată de către zona[p].urm*/
      zona[zona[p-1].urm-1].nodlista= x;
} /*insereaza*/
```

# Tehnici de implementare a listelor

---

```
void suprima(tip_cursor p, tip_cursor* inceput)
{
    if (p==0) /*performanța O(1)*/
        muta(inceput, &disponibil);
    else
        muta(&zona[p-1].urm, &disponibil);
}

void init()
/*inițializează elementele zonei înlănțuindu-le în lista de disponibili*/
{
    tip_cursor i; /*performanța O(n)*/
    for( i=lung_max-1; i >= 1; i --)
        zona[i-1].urm= i+1;
    disponibil= 1;
    zona[lung_max-1].urm= 0;
}
```

# Tehnici de implementare a listelor

---

Comparație între metodele de implementare a listelor

- Este greu de precizat care dintre metodele de implementare a listelor este mai bună, deoarece răspunsul depinde de:

- (1) Limbajul de programare utilizat.
- (2) Operațiile care se doresc a fi realizate.
- (3) Frecvența cu care sunt invocați operatorii.
- (4) Constrângerile de timp de acces, de memorie, de performanță, ș.a.

# Tehnici de implementare a listelor

---

Observații:

(1) Implementarea bazată pe tablouri sau pe cursori necesită specificarea dimensiunii maxime a listei în momentul compilării.

- Dacă nu se poate determina o astfel de limită superioară a dimensiunii listei se recomandă implementarea bazată pe pointeri sau referințe.

(2) Aceiași operatori pot avea performanțe diferite în implementări diferite.

- Spre exemplu inserția sau ștergerea unui nod precizat au o durată constantă la implementarea înlănțuită ( $O(1)$ ), dar necesită o perioadă de timp proporțională cu numărul de noduri care urmează nodului în cauză în implementarea bazată pe tablouri ( $O(n)$ ).

- În schimb operatorul Anterior necesită un timp constant în implementarea prin tablouri ( $O(1)$ ) și un timp care depinde de lungimea totală, respectiv de poziția nodului în implementarea bazată pe pointeri, referințe sau cursori ( $O(n)$ ).

# Tehnici de implementare a listelor

---

Observații:

(3) Implementarea bazată pe tablouri sau pe cursori poate fi ineficientă din punctul de vedere al utilizării memoriei, deoarece ea ocupă tot timpul spațiul maxim solicitat, indiferent de dimensiunea reală a listei la un moment dat.

(4) Implementarea înlănțuită utilizează în fiecare moment spațiul de memorie strict necesar lungimii curente a listei, dar necesită în plus spațiul pentru înlănțuire în cadrul fiecărui nod.

În funcție de circumstanțe una sau alta dintre implementări poate fi mai mult sau mai puțin avantajoasă.

# Structuri de date derivate din TDA listă

---

- În cadrul acestui subcapitol vor fi prezentate câteva dintre structurile de date care derivă din structura de date listă, fiind considerate liste speciale.
- Este vorba despre: listele circulare, listele dublu înlănțuite, stivele și cozile.

# Structuri de date derivate din TDA listă

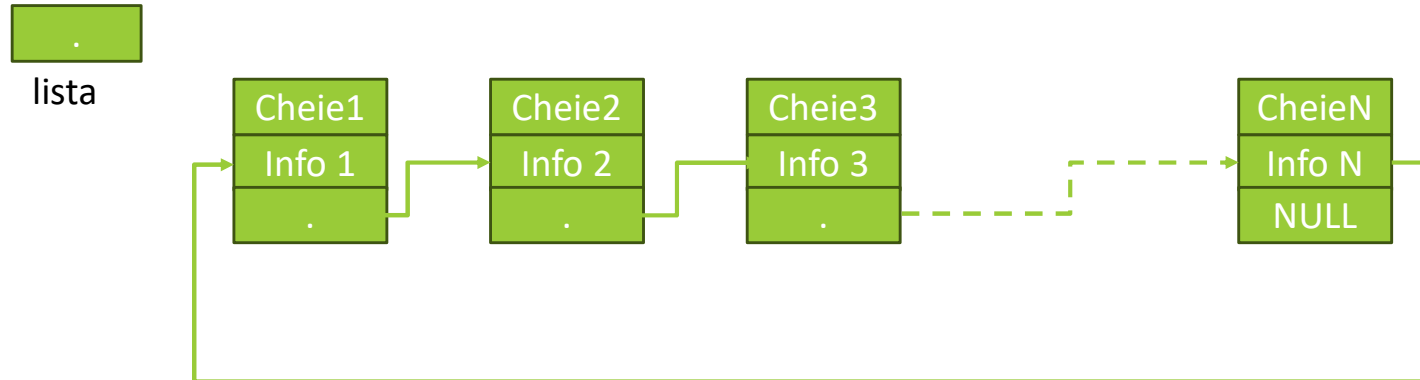
---

## Liste circulare

- Listele circulare sunt liste înlănțuite ale căror înlănțuiri se închid.
- În aceste condiții se pierde noțiunea de început și sfârșit, lista fiind referită de un pointer care se deplasează de-a lungul ei
- Listele circulare ridică unele probleme referitoare la inserția primului nod în listă și la suprimarea ultimului nod.

# Structuri de date derivate din TDA listă

Liste circulare

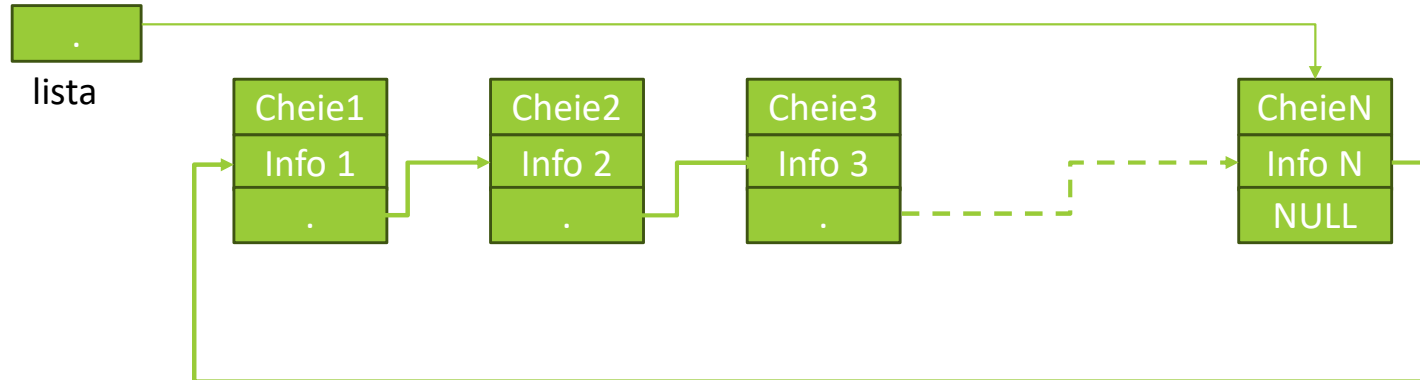


Ce nod ar trebui să refere nodul lista (primul, ultimul, orice nod)?



# Structuri de date derivate din TDA listă

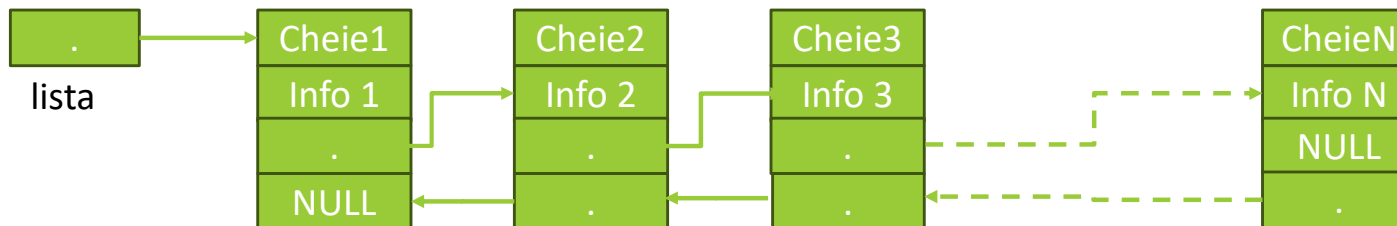
Liste circulare



Ce caz de inserție și de ștergere este cel mai eficient? Care este complexitatea pentru acel caz?

# Structuri de date derivate din TDA listă

- Unele aplicații necesită traversarea listelor în ambele sensuri.
- Cu alte cuvinte fiind dat un element oarecare al listei trebuie determinat cu rapiditate atât succesorul cât și predecesorul acestuia.
- Maniera cea mai rapidă de a realiza acest lucru este aceea de a memora în fiecare nod al listei referințele către nodul următor și către nodul anterior.
- Această abordare conduce la structura listă **dublu înlănțuită**.



# Structuri de date derivate din TDA listă

---

- Prețul care se plătește este:

(1) Prezența unui câmp suplimentar de tip pointer în fiecare nod.

(2) O oarecare creștere a complexității funcțiilor care implementează operatorii de bază care prelucrează astfel de liste.

- Dacă implementarea acestor liste se realizează cu pointeri se pot defini tipurile de date din secvența

```
typedef struct {  
    TipCheie cheie;  
    TipInfo info;  
    struct NodDubla * urm, *ant;  
}NodDubla;
```

# Structuri de date derivate din TDA listă

Exemplu de operații asupra listei dublu înlanțuite:

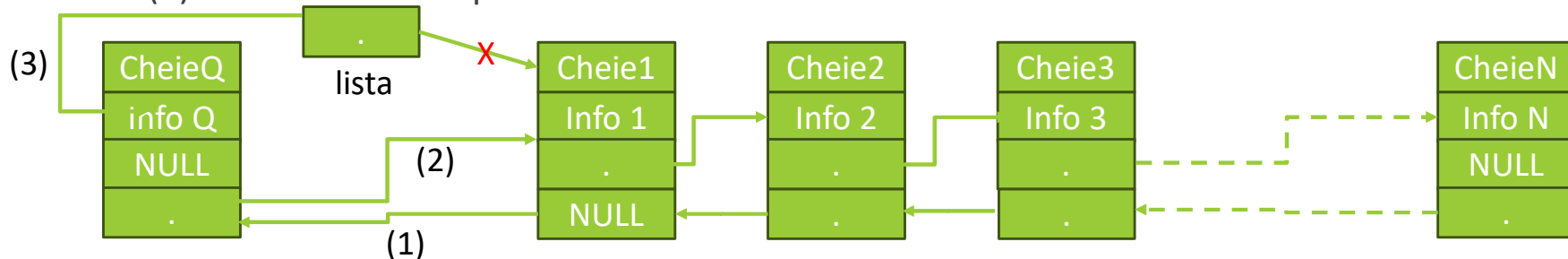
## Inserarea unui element pe prima poziție:

(1) Se alocă spațiu pentru noul nod și se completează câmpurile de informație și cheie ale acestuia

(2) Se pune pe NULL câmpul anterior

(3) Se face legătura de la primul nod al listei vechi, dacă acesta există, spre noul nod

(4) Noul nod devine primul nod



# Structuri de date derivate din TDA listă

---

O variantă de implementare în C este:

```
NodDubla* insertie_inceputDubla(NodDubla * listaD,int cheie, int
info)
{
    NodDubla* q = (NodDubla*)malloc(sizeof(NodDubla));
    if (q) //daca alocarea a avut loc cu succes
    {
        q->cheie = cheie;
        q->info = info;
        q->urm = listaD;
        q->ant = NULL;
        if (listaD)
            listaD->ant = q;
        listaD = q;
    }
    return listaD;
}
```

# Structuri de date derivate din TDA listă

---

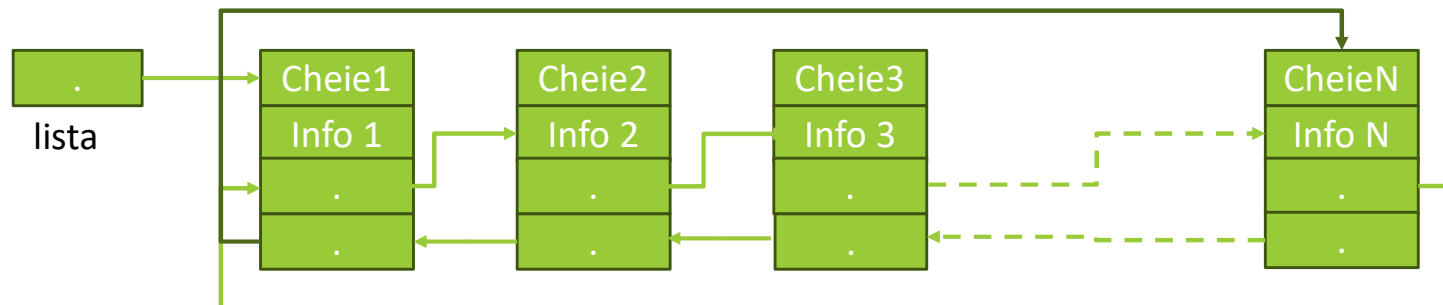
Exemplu de operații asupra listei dublu înlănțuite:

## **Suprimarea elementului situat în poziția p a unei liste dublu înlănțuite:**

- Considerând că nodul suprimat nu este nici primul nici ultimul nod al listei:
  - (1) Se localizează nodul ant(erior) și se face câmpul urm(ător) al acestuia să indice nodul care urmează celui indicat de p.
  - (2) se modifică câmpul anterior al nodului care urmează celui indicat de p astfel încât el să indice nodul precedent celui indicat de p.
  - (3) Nodul suprimat este indicat în continuare de p, ca atare spațiul de memorie afectat lui poate fi reutilizat în regim de alocare dinamică a memoriei.

# Structuri de date derivate din TDA listă

Listele dublu înlanțuite pot fi implementate și ca liste circulare



# Stive

O **stivă** este o structură liniară care poate fi accesată doar la un capăt al său pentru insera și pentru a extrage date.

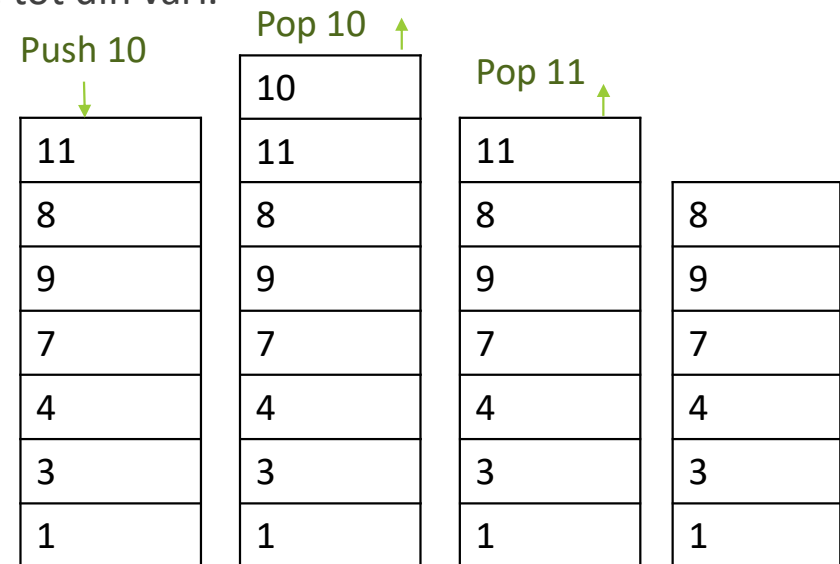
Structura se numește stivă deoarece funcționează în mod similar cu o stivă de cărți. Se poate adăuga doar dus, în vârful stivei și se poate extrage tot din vârf.

În limba engleză folosim termenul de stack sau

LIFO (last in, first out)

**Definirea TDA Stivă** presupune precizarea:

- (1) Modelului matematic asociat.
- (2) Notațiilor utilizate.
- (3) Operatorilor definiți pentru acest tip.





# Stive

---

## TDA Stivă

Modelul matematic: o secvență finită de noduri. Toate nodurile aparțin unui același tip numit tip de bază. O stivă este de fapt o listă specială în care toate inserțiile și toate suprimările se fac la un singur capăt care se numește vârful stivei.

### Notații:

s: TipStiva;

x: TipElement.

### Operatori:

- 1. *Initializeaza(s:TipStiva){:TipPozitie};*** - face stiva s vidă.
- 2. *VarfSt(s:TipStiva):TipElement;*** - furnizează elementul din vârful stivei s.
- 3. *Pop(s:TipStiva);*** - suprimă elementul din vârful stivei.
- 4. *Push(x:TipElement,s:TipStiva);*** - inserează elementul x în vârful stivei s.
- 5. *Stivid(s:TipStiva):boolean;*** - returnează valoarea adevărat dacă stiva s este vidă și fals în caz contrar.

# Stive

---

În secvența apare un exemplu de implementare a TDA Stivă utilizând TDA Listă varianta restrânsă.

```
{Implementarea TDA Stivă bazată pe TDA Lista (varianta restrânsă)}  
TYPE TipStiva = TipLista;  
VAR s: TipStiva;  
p: TipPozitie;  
x: TipNod;  
b: boolean;  
{Initializeaza(s:TipStiva);} p:= Initializeaza(s);  
{VarfSt(s);} x:= Furnizeaza(Primul(s),s);  
{Pop(s);} Suprima(Primul(s),s);  
{Push(x,s);} Insereaza(s,x,Primul(s));  
{Stivid(s);} b:= Fin(s)=0;
```

# Stive

---

În secvența apare un exemplu de implementare a TDA Stivă utilizând TDA Listă varianta restrânsă.

```
typedef tip_lista tip_stiva;  
tip_stiva s;  
tip_pozitie p;  
tip_nod x;  
boolean b;  
/*Initializeaza(s:TipStiva);*/p= initializeaza(s);  
/*VarfSt(s);*/ x= furnizeaza(primul(s),s);  
/*Pop(s);*/ suprima(&primul(s),s);  
/*Push(x,s);*/ insereaza(s,x,primul(s));  
/*Stivid(s);*/ b= fin(s)==0;
```

# Stive

---

Acesta este în același timp și un exemplu de **implementare ierarhică** a unui **tip de date abstract** care ilustrează:

- Pe de o parte **flexibilitatea** și **simplitatea** unei astfel de abordări.
- Pe de altă parte, **invarianța** ei în raport cu **nivelurile ierarhiei**.
- Cu alte cuvinte, **modificarea** implementării **TDA Listă** nu afectează sub nici o formă implementarea **TDA Stivă** în accepțiunea păstrării nemodificate a **prototipurilor operatorilor definiți**.
- Utilizarea deosebit de frecventă și cu mare eficiență a **structurii de date stivă** în domeniul programării, a determinat **evoluția** acesteia de la statutul de **structură de date avansată** spre cel de **structură fundamentală**.
- Această tendință s-a concretizat în **implementarea hardware** a acestei structuri în toate sistemele de calcul moderne și în includerea operatorilor specifici tipului de date abstract stivă în **setul de instrucții cablate** al procesoarelor actuale.

# Stive

---

Structura stivă, implementare bazată pe tablouri:

```
typedef struct{
    tip_element elemente[lungime_maxima];
    int varf;
}tip_stiva;
/*varianta implementare stiva care porneste de la lungime maxima
si adauga elemente spre 0*/

void initializeaza(tip_stiva* s)
{
    s->varf = lungime_maxima + 1;
} /*initializeaza*/
```

# Stive

---

```
boolean stivid(tip_stiva s)
{
    boolean stivid_result;
    if (s.varf>lungime_maxima)
        stivid_result = true;
    else stivid_result = false;
    return stivid_result;
} /*stivid*/
tip_element varfst(tip_stiva* s)
{
    boolean er;tip_element varfst_result;
    if (stivid(*s)){
        er = true;
        printf("stiva este vida");
    }
    else varfst_result = s->elemente[s->varf - 1];
    return varfst_result;
}
```

# Stive

```
void pop(tip_stiva* s, tip_element* x)
{
    boolean er;
    if (stivid(*s)){
        er = true;
        printf("stiva este vida");}
    else{
        *x = s->elemente[s->varf - 1];
        s->varf = s->varf + 1;}
} /*pop*/
```

```
void push(tip_element x, tip_stiva* s)
{
    boolean er;
    if (s->varf == 1){
        er = true;
        printf("stiva este plina");}
    else{
        s->varf = s->varf - 1;
        s->elemente[s->varf - 1] = x;}
} /*push*/
```

# Stive

---

Ce implică o implementare cu pointeri?



Cum s-ar implementa cel mai eficient operațiile de pop și push?



# Stive

---

Avantaje și exemple de utilizare:

- Atât operația de push cât și cea de pop au o complexitate de  $O(1)$
- Structura stivă are numeroase aplicații:
  - în recursivitate și tehnici de eliminare a acestora
  - pentru evaluarea expresiilor
  - pentru managementul memoriei

# Cozi

---

Asemenea stivelor, **cozile** sunt tot structuri derivate din liste, care oferă acces restricționat la elementele lor

Elementele sunt inserate la un capăt (**spate**) și sunt suprimate la celălalt (**început**).

- Cozile se mai numesc liste "**FIFO**" ("**first-in first-out**") adică liste de tip "**primul-venit-primul-servit**".
- Operațiile care se execută asupra **cozii** sunt analoage celor care se realizează asupra **stivei** cu diferența că inserările se fac la **spatele** cozii și **nu** la **începutul** ei

În cazul cozilor se utilizează ambele capete, spre deosebire de stivă unde atât operațiile de inserție cât și de suprimare aveau loc prin intermediul aceluiași capăt numit vârf.

Practic cozile se comportă ca o coadă de așteptare, în care primul venit este servit și iese din coadă, iar noii veniti se aseză rând pe rând la capătul din spate al cozii

# Cozi

---

## Tipul de date abstract Coadă

- În acord cu abordările anterioare, la fel ca în cazul stivei, se definesc **două** variante ale **TDA Coadă**.
- În continuare se prezintă un exemplu de implementare a **TDA Coadă** bazat pe **TDA Listă**.

## TDA Coadă

Modelul matematic: o secvență finită de noduri. Toate nodurile aparțin unui același tip numit tip de bază. O coadă este de fapt o listă specială în care toate inserțiile se fac la un capăt (spate) și toate ștergerile se fac la celălalt capăt (cap).

## Notății:

C: TipCoadă;

x: TipElement; b: boolean;

# Cozi

---

## Operatori (setul 1):

1. **Initializeaza**( $C: \text{TipCoadă}$ ){:  $\text{TipPozitie}$ }; - face coada  $C$  vidă.
2. **Cap**( $C: \text{TipCoadă}$ ):  $\text{TipElement}$ ; - funcție care returnează primul element al cozii  $C$ .
3. **Adauga**( $x: \text{TipElement}, C: \text{TipCoadă}$ ); - inserează elementul  $x$  la spatele cozii  $C$ .
4. **Scoate**( $C: \text{TipCoadă}$ ); - suprimă primul element al lui  $C$ .
5. **Vid**( $C: \text{TipCoadă}$ ):  $\text{boolean}$ ; - este adevărat dacă și numai dacă  $C$  este o coadă vidă.

# Cozi

---

## Operatori (setul 2):

1. **CreazaCoadăVidă**(*C: TipCoadă*); - face coada *C* vidă.
2. **CoadăVidă**(*C: TipCoadă*): *boolean*; - este adevărat dacă și numai dacă *C* este o coadă vidă.
3. **CoadăPlină**(*C: TipCoadă*): *boolean*; - este adevărat dacă și numai dacă *C* este o coadă plină (operator dependent de implementare).
4. **InCoadă**(*C: TipCoadă x: TipElement*); - inserează elementul *x* la spatele cozii *C* (**EnQueue**).
5. **DinCoadă**(*C: TipCoadă, x: TipElement*); - suprimă primul element al lui *C* (**DeQueue**).

# Cozi

---

**{Exemplu de implementare a TDA Coadă cu ajutorul TDA Listă (setul 1 de operatori)}**

**TYPE** TipCoadă = TipLista;

**VAR** C: TipCoadă;

p: TipPozitie;

x: TipElement;

b: boolean;

**{Initializeaza(C);}**

**{Cap(C);}**

**{Adauga(x,C);}**

**{Scoate(C);}**

**{Vid(C,C);}**

p:= **Initializeaza(C);**

x:= **Furnizeaza(Primul(C),C);**

**Insereaza(C,x,Fin(C));**

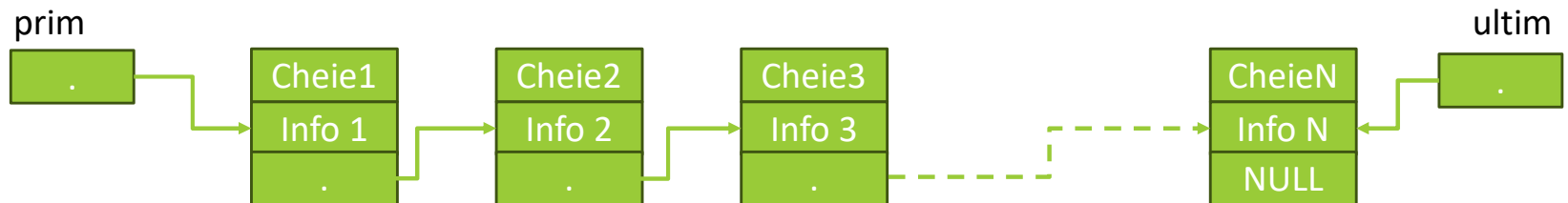
**Suprima(Primul(C),C);**

b:= **Fin(C)=0;**

# Cozi

Ca și în cazul stivelor, orice **implementare** a listelor este valabilă și pentru **cozi**.

- Pornind însă de la observația că inserțiile se fac numai la **spatele** cozii, funcția **Adauga** poate fi concepută mai eficient.
- Astfel, în loc de a parcurge de fiecare dată coada de la început la sfârșit atunci când se dorește adăugarea unui element, se va păstra un **pointer** la **ultimul element al cozii**.
- De asemenea, ca și la toate tipurile de liste, se va păstra și pointerul la **primul element al listei** utilizat în execuția comenzilor **Cap** și **Scoate**.



# Cozi

---

```
/*Exemplu de implementare a TDA Coada cu ajutorul TDA Lista
(setul 1 de operatori).*/
typedef int tip_element;
typedef struct
{
    tip_element element;
    struct tip_nod* urm;
}tip_nod;
typedef struct
{
    tip_nod *prim;
    tip_nod *ultim;
}tip_coada;
```



# Cozi

---

```
void initializeaza(tip_coadă* c)
{
    c->prim= NULL;
    c->ultim= NULL;
    /*nici prim nici spate nu indica spre vreun element*/
} /*initializeaza*/
boolean vid(tip_coadă c)
{
    if (c.prim == NULL)
        return true;
    else return false;
} /*vid*/
tip_element cap(tip_coadă c)
{
    return c.prim->element;
} /*cap*/
```

# Cozi

---

```
void adauga(tip_coada* c, tip_element x)
{tip_nod *nou = (tip_nod*)malloc(sizeof(tip_nod));
if (nou) /*daca s-a alocat spatiu pentru nou cu succes*/
{nou->element = x;
nou->urm = NULL;
if (c->prim== NULL)
/* daca lista este vida, noul nod este si primul si ultimul */
{  c->prim= nou;
    c->ultim= nou;
}
else {
    c->ultim->urm = nou; /*se adauga un nod nou la sfarsitul cozii*/
    c->ultim= c->ultim->urm; /*actualizam spatele cozii*/
}
}
} /*adauga*/
```

# Cozi

---

```
void scoate(tip_coadă* c, int * er)
{
    if (vid(*c))
        *er = true; // coada este vida, nu se poate face suprimarea
    else
    {
        *er = false;
        c->prim= c->prim->urm;
    }
} /*scoate*/
```

# Cozi

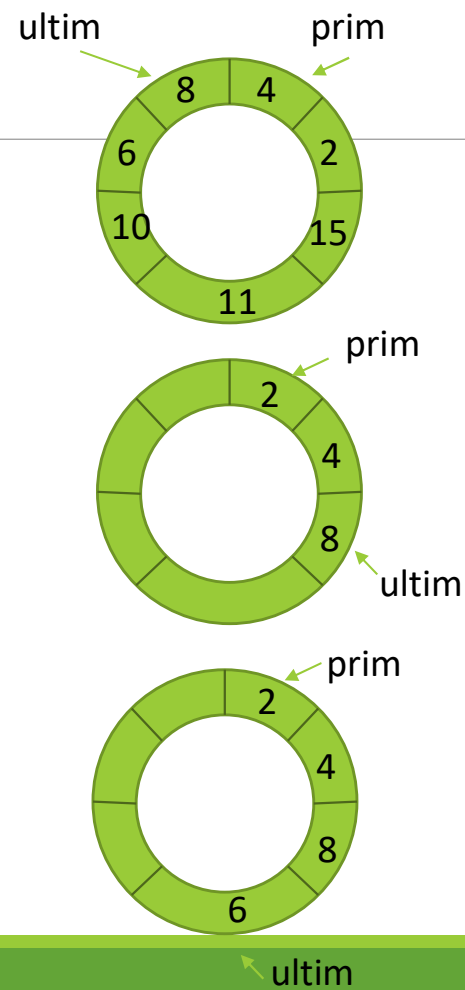
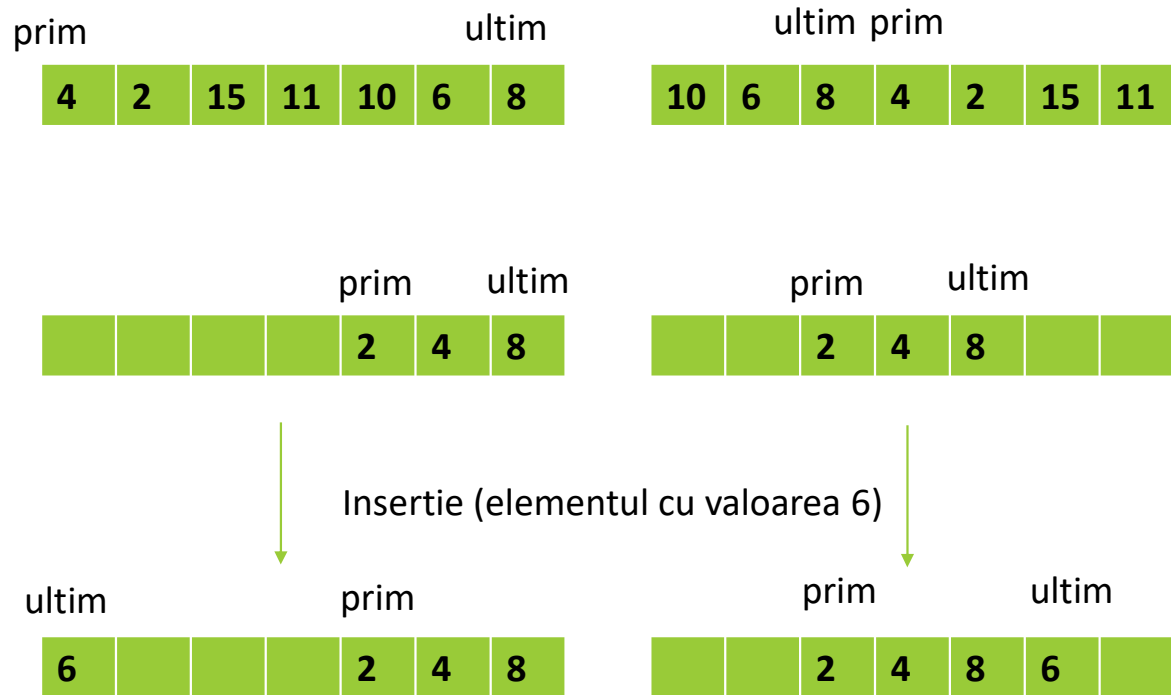
---

## Cozi circulare

Reprezentarea **listelor** cu ajutorul **tablourilor**, **poate** fi utilizată și pentru **cozi**, dar **nu** este foarte eficientă.

- Într-adevăr, indicatorul la ultimul element al listei, permite implementarea **simplă**, într-un număr fix de pași, a operației **Adauga**.
- Execuția operației **Scoate** presupune însă **mutarea** întregii cozi cu o poziție în tablou, operație care necesită un timp  $O(n)$ , dacă coada are lungimea  $n$ .
- Pentru a depăși acest dezavantaj se poate utiliza **o structură de tablou circular** în care prima poziție urmează ultimei

# Cozi



# Cozi

---

- **Adăugarea** unui element presupune mișcarea indicatorului C.ultim cu o poziție în sensul acelor de ceasornic și introducerea elementului în această poziție.
- **Scoaterea** unui element din coadă, presupune simpla mutare a indicatorului C.prim cu o poziție în sensul rotirii acelor de ceasornic.
- Astfel coada se rotește în sensul rotirii acelor de ceasornic după cum se adaugă sau se scot elemente din ea.
- În aceste condiții atât funcția Adauga cât și Scoate pot fi redactate astfel încât să presupună un număr fix de pași de execuție, cu alte cuvinte să fie  **$O(1)$** .
- În cazul implementării ce apare în continuare, indicatorul C.prim indică **primul element al cozii**, iar indicatorul C.ultim **ultimul element al cozii**.

# Cozi

---

- Acest mod de implementare ridică însă o problemă referitoare la sesizarea **cozii vide** și a **celelalte pline**.
- Presupunând că structura **coadă** este **plină** (conține LungimeMaxima elemente), indicatorul C.ultim va indica **poziția adiacentă** lui C.prim la parcurgerea în sensul arcelor de ceasornic.
- Pentru a preciza reprezentarea **cozii vide** se presupune o coadă formată dintr-un singur element.
- În acest caz C.prim și C.ultim indică aceeași poziție.
- Dacă se scoate **singurul** element, C.prim avansează cu o poziție înainte (în sensul acelor de ceasornic) indicând **coada vidă**.
- Se observă însă că această situație este **identică** cea anterioară care indică coada plină, adică C.prim se află cu o poziție în fața lui C.ultim, la parcurgerea în sensul arcelor de ceasornic a cozii.

# Cozi

---

- În vederea rezolvării acestei situații, în tablou se vor introduce **doar**  $LungimeMaxima - 1$  elemente, deși în tablou există  $LungimeMaxima$  poziții.
- Astfel testul de **coadă plină** conduce la o valoare adevărată dacă  $C.ultim$  devine egal cu  $C.prim$  după **două** avansări succesive.
- Testul de **coadă vidă** conduce la o valoare adevărată dacă  $C.ultim$  devine egal cu  $C.prim$  după avansul cu o poziție.
- Evident avansările se realizează în sensul acelor de ceasornic al parcurgerii cozii.



# Cozi

---

O variantă de implementare

```
typedef unsigned boolean;
#define true (1)
#define false (0)
#define lungime_maxima 5
/*Exemplu de implementare a TDA Coada cu ajutorul TDA Lista
(setul 1 de operatori).*/
typedef int tip_element;
typedef struct coada_c
{
    tip_element coada[lungime_maxima + 1];
    int dimensiune;
    int prim;
    int ultim;
}tip_coada;
```

# Cozi

---

```
void initializeaza(tip_coadă* c)
{
    c->dimensiune = lungime_maxima + 1;
    c->prim = 1;
    c->ultim = 0;
}
tip_element cap(tip_coadă c)
{
    return c.coadă[c.prim];
}
boolean vid(tip_coadă c)
{
    if ((c.ultim + 1) % c.dimensiune) == c.prim)
        return true;
    else return false;
}
```

# Cozi

---

```
void adauga(tip_coda* c, tip_element el)
{
    if (((c->ultim + 2) % c->dimensiune) == c->prim)
        printf("Coda este plina\n");
    else
    {
        c->ultim = (c->ultim + 1) % c->dimensiune;
        c->coada[c->ultim] = el;
    }
}

void scoate(tip_coda* c, int* er)
{
    if (vid(*c))
        *er = true;
    else
    {
        *er = false;
        c->prim = (c->prim + 1) % c->dimensiune;
    }
}
```

# Cozi

---

## Cozi bazate pe prioritate

- Coadă bazată pe prioritate ("priority queue") este structura de date abstractă care permite inserția unui element (poate fi sau nu ordonată în funcție de prioritatea sa) și **suprimarea celui mai prioritar element**.
- O astfel de structură diferă atât față de structura coadă (din care se suprimă primul venit, deci cel mai vechi) cât și față de stivă (din care se suprimă ultimul venit, deci cel mai nou).
- De fapt cozile bazate pe prioritate pot fi concepute ca și structuri care generalizează cozile și stivele.
- Aplicațiile cozilor bazate pe prioritate sunt:
  - Sisteme de simulare - unde cheile pot corespunde unor evenimente "de timp" ce trebuie să decurgă în ordine temporală.
  - Planificarea firelor de execuție de către un sistem de operare.
  - Traversări speciale ale unor structuri de date

# Cozi

---

## TDA Coadă bazată pe prioritate

- Considerând **coada bazată pe prioritate** drept o structură de date abstractă ale cărei elemente sunt articole cu **chei** afectate de **priorități**, definirea acesteia poate fi:

**Modelul matematic:** o secvență finită de noduri. Toate nodurile aparțin unui același tip numit tip de bază. Fiecare nod are o asociată o prioritate. O coadă bazată **pe prioritate** este de fapt o listă specială în care se permite inserția ordnată după un criteriu și suprimarea doar a celui mai prioritar nod.

### Notății:

*q*: *TipCoadăBazataPePrioritate*;

*x*: *TipElement*;

# Cozi

---

## Operatori:

1. **Initializează**( $q$ : *TipCoadaBazataPePrioritate*); - face coada  $q$  vidă.
2. **Inseerează**( $x$ : *TipElement*,  $q$ : *CoadaBazataPePrioritate*); - inserția unui nou element  $x$  în coada  $q$ .
3. **Extrage**( $q$ : *CoadaBazataPePrioritate*): *TipElement*; - extrage cel mai prioritar element al cozii  $q$ .
4. **Inlocuiește**( $q$ : *CoadaBazataPePrioritate*,  $x$ : *TipElement*): *TipElement*; - înlocuiește cel mai prioritar element al cozii  $q$  cu elementul  $x$ , mai puțin în situația în care noul element este cel mai prioritar element. Returnează cel mai prioritar element.
5. **Schimbă**( $q$ : *TipCoadaBazataPePrioritate*,  $x$ : *TipElement*;  $p$ : *TipPrioritate*); - schimbă prioritatea elementului  $x$  al cozii  $q$  și îi conferă valoarea  $p$ .
6. **Suprimă**( $q$ : *TipCoadaBazataPePrioritate*,  $x$ : *TipElement*); - suprimă elementul  $x$  din coada  $q$ .
7. **Vid**( $q$ : *TipCoadaBazataPePrioritate*): *boolean*; - operator care returnează **true** dacă și numai dacă  $q$  este o coadă vidă.

# Cozi

---

Operatorul **Inlocuiește** constă dintr-o **inserție** urmată de **suprimarea** celui mai prioritar element.

- Este o operație diferită de succesiunea suprimare-inserție deoarece necesită creșterea pentru moment a dimensiunii cozii cu un element.
- Acest operator se definește separat deoarece în anumite implementări poate fi conceput foarte eficient.
- În mod analog operatorul **Schimbă** poate fi implementat ca și o **suprimare**, urmată de o **inserție**, iar generarea unei cozi ca și o succesiune de operatori **Inserează**.
- **Cozile bazate pe prioritate** pot fi în general implementate în diferite moduri unele bazate pe structuri simple, altele pe structuri de date avansate, fiecare presupunând însă performanțe diferite pentru operatorii specifici.

# Cozi

---

## **Coadă bazată pe prioritate:**

Implementări și complexitate:

- Tablou unidimensional (vector) neordonat – Inserție  $O(1)$ , Ștergere  $O(n)$
- Listă cu pointeri neordonată - Inserție  $O(1)$ , Ștergere  $O(n)$
- Tablou unidimensional (vector) ordonat – Inserție  $O(n)$ , Ștergere  $O(1)$
- Listă cu pointeri ordonată - Inserție  $O(n)$ , Ștergere  $O(1)$
- Arbori (nu face obiectul acestui curs) – Inserție  $O(\log n)$ , Ștergere  $O(\log n)$



# Multiliste

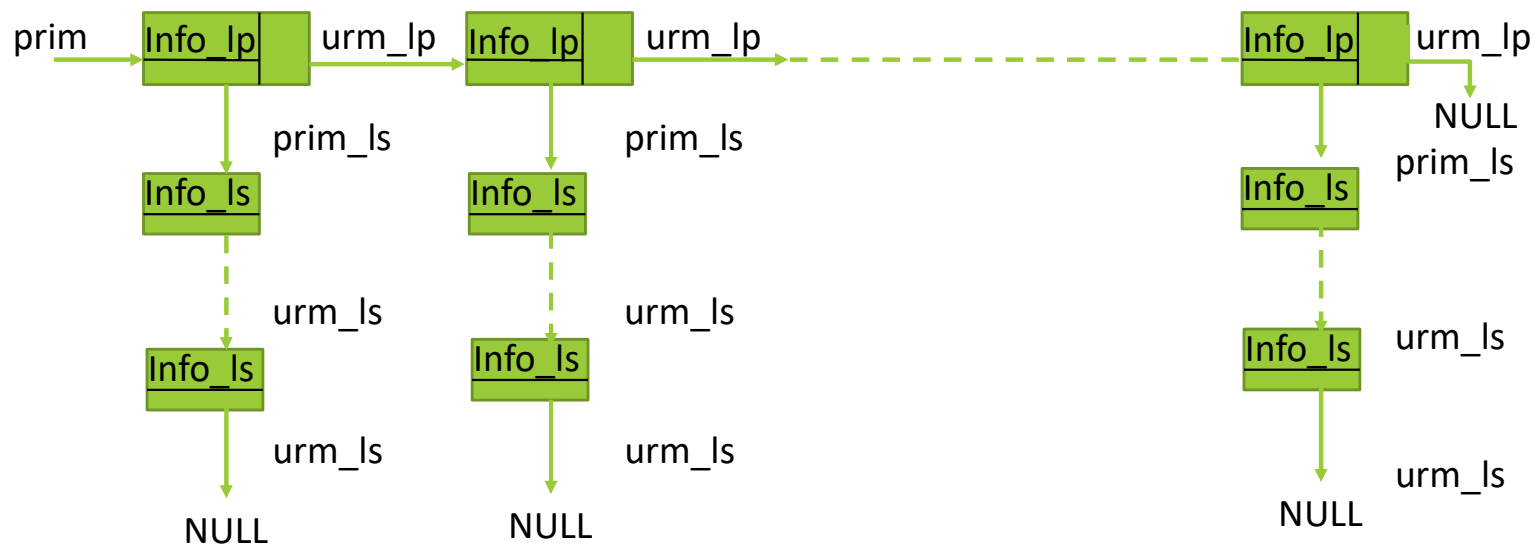
---

## Structura de date **multilistă**

- Se numește multilistă, o structură de date ale cărei noduri conțin mai multe câmpuri de înlănțuire.
- Cu alte cuvinte, un nod al unei astfel de structuri poate aparține în același timp la mai multe liste înlănțuite simple.
- În literatura de specialitate termenii consacrați pentru a desemna o astfel de structură sunt:
- "Multilist".
- "Multiply linked list"

# Multiliste

Un exemplu de multilistă în care avem o listă principală și fiecare nod al listei principale indică către o listă secundară:



# Multiliste

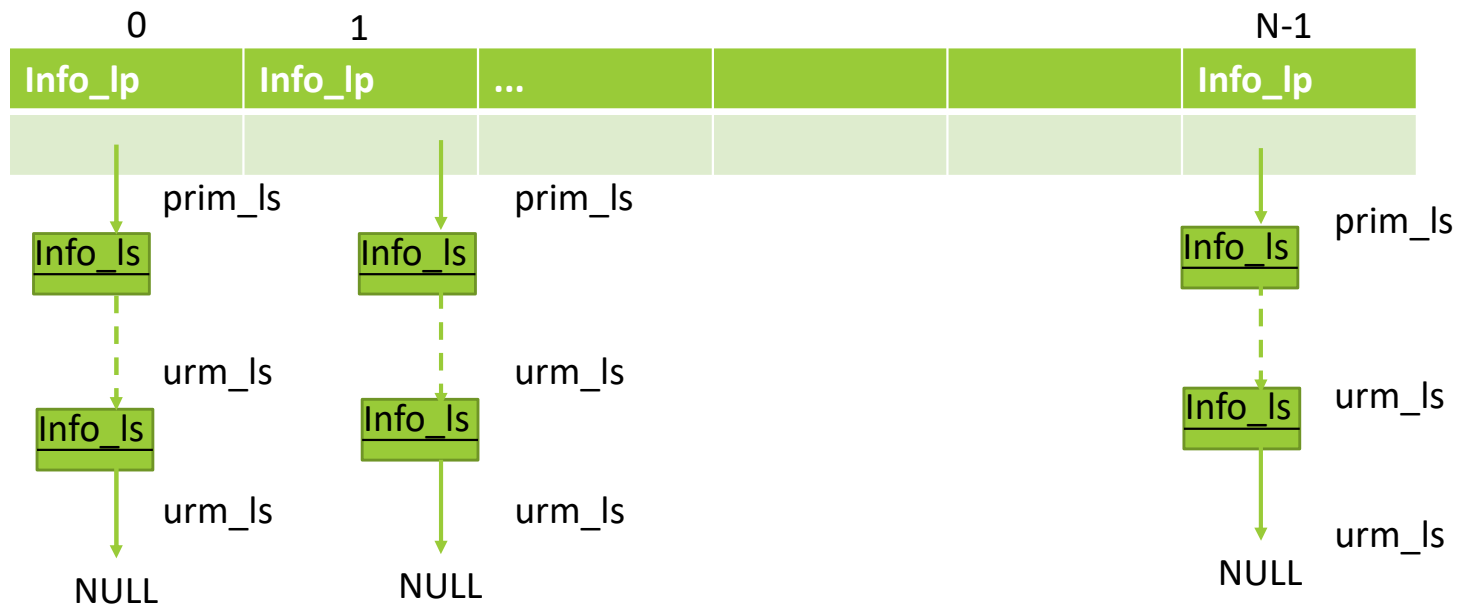
---

În cazul în care atât lista principală și listele secundare sunt implementate cu **pointeri**, atunci putem avea următoarele structuri:

```
typedef struct Nod_ls{
    TipCheie cheie;
    TipInfo info_lp;
    struct Nod_p *urm_ls;
}Nod_ls; //nodul listei secundare
typedef struct Nod_lp{
    TipCheie cheie;
    TipInfo info_lp;
    struct Nod_lp *urm_lp;
    struct Nod_ls *prim_ls;
}Nod_lp; //nodul listei principale
Nod_lp *prim; //un pointer catre lista de liste
```

# Multiliste

Un exemplu de lista principală implementată cu ajutorul tablourilor și liste secundare implementate cu ajutorul pointerilor:



# Multiliste

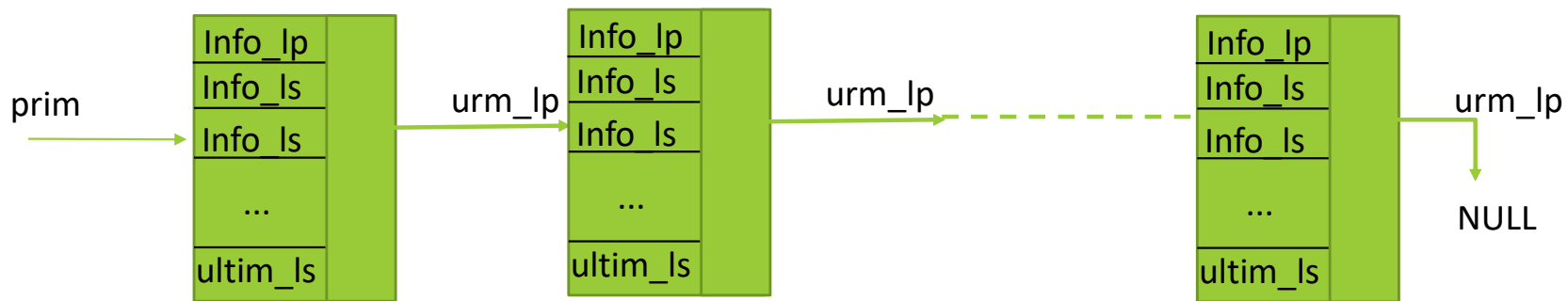
---

În cazul în care **lista principală** este implementată cu ajutorul **tablourilor**, iar **listele secundare** sunt implementate cu ajutorul **pointerilor**, atunci vom avea un tablou de liste, fiecare element din tablou indicând spre începutul unei liste secundare.

```
typedef struct Nod_ls{
    TipCheie cheie;
    TipInfo info_lp;
    struct Nod_p *urmator_ls;
}Nod_ls; //nodul listei secundare
typedef struct Nod_lp{
    TipCheie cheie;
    TipInfo info_lp;
    struct Nod_ls *inceput_ls;
}Nod_lp; //nodul listei principale
#define LungMax 100
typedef struct {
    Nod_lp tablou[LungMax];
    int ultim; } Lista;
```

# Multiliste

Un exemplu în care lista principală este implementată cu ajutorul pointerilor, iar listele secundare sunt implementate cu ajutorul tablourilor:



# Multiliste

---

În cazul în care lista principală este implementată cu ajutorul pointerilor, iar listele secundare sunt implementate cu ajutorul tablourilor, avem:

```
typedef struct Nod_ls{
    TipCheie cheie;
    TipInfo info_lp;
}Nod_ls; //nodul listei secundare
typedef struct Nod_lp{
    TipCheie cheie;
    TipInfo info_lp;
    struct Nod_ls[MAX];
    int ultim_ls;
    struct Nod_lp *urmator_lp;
}Nod_lp; //nodul listei principale
```

# Multiliste

---

**Avantajul** utilizării unor astfel de structuri este evident.

- Prezența mai **multor înlănțuiri** într-un același nod, respectiv **apartenența simultană** a aceluiași nod la mai multe liste, asigură acestei structuri de date o **flexibilitate** deosebită.
- Acest avantaj, coroborat cu o manipulare relativ **facilă** specifică structurilor înlănțuite este exploatat la implementarea **bazelor de date**, cu precădere a celor relaționale.
- Aria de utilizare a structurilor multilistă este însă mult mai extinsă.
- Spre exemplu, o astfel de structură poate fi utilizată cu succes la memorarea **matricelor rare**.
- Este cunoscut faptul că **matricele rare** sunt matrice de mari dimensiuni care conțin de regulă un **număr redus** de elemente restul fiind poziționate de obicei pe zero.
- Din acest motiv memorarea lor în forma obișnuită a tablourilor bidimensionale presupune o **risipă** mare de memorie.



# Exerciții

---

E1. Să se implementeze o funcție de ștergere a unui nod cu o cheie dată dintr-o listă simplu înlănțuită. Funcția trebuie să ia în considerare toate cazurile de ștergere.

Ex2. Să se implementeze o funcție de ștergere a unui nod cu o cheie dată dintr-o listă dublu înlănțuită. Funcția trebuie să ia în considerare toate cazurile de ștergere.

Ex3. Să se implementeze o structură de tip coadă cu priorități folosind o listă simplu înlănțuită ordonată după priorități. Pentru această structură să se implementeze operatorii: inițializează, inserează și extrage, definiți în acest curs.

Ex4. Să se implementeze cât mai eficient o funcție care să inverseze ordinea elementelor într-o listă simplu înlănțuită dată.

Ex5. Să se implementeze o funcție care inversează conținutul unei stive, folosind doar operații de push și pop, fără a folosi alte structuri auxiliare definite de utilizator. (Obs: Ne putem folosi de recursivitate)

Ex6. Definiți și implementați, cât mai eficient, o structură asemănătoare unei cozi, cu proprietatea că elementele pot fi adăugate atât în față, cât și în spate.

# Bibliografie selectivă

---

- Drozdek, A. (2012). *Data Structures and algorithms in C++*. Cengage Learning.
- Shaffer, C. A. (2012). Data structures and algorithm analysis.
- Crețu, V. Structuri de date și algoritmi, Editura Orizonturi Universitare Timișoara, 2011