Curs_8 - Liste

Introducere

Limitarile tablourilor

```
-> Nevoia de a cunoaste dimensiunea acestora
-> Nu putem efectua si insetie si stergere cu complexitatea O(1) , aceste operatii avand o complexitate O(n) in cazul cel mai defavorabil
```

Solutii

```
-> Aceste limitari pot fi depasite prin intermediul listelor inlantuite

-> O structura inlantuita = o colectie de noduri in care retinem informatii utile si legaturi catre alte noduri
-> Nodurile pot fi stocate oriunde in memorie, iar legatura catre alt nod se face prin retinerea adresei nodului spre care dorim sa retinem legatura
```

Tipul de date abstract listă

Structura de date lista

```
-> structura de date avansata
-> structura dinamica, definita pornind de la notiunea de vector
-> are elemente de acelasi tip => structura de date omogena
-> toate elementele sunt inregistrate in memoria centrala a sistemului de calcul
-> numarul componentelor poate fi nul / variabil
-> structura flexibila particulara, care poate creste sau descreste in functie de necesitati si ale carei elemente pot fi referite, inserate sau sterse in orice pozitie din cadrul listei
-> doua sau mai multe liste pot fi concatenate sau scindate in subliste
```

TDA Lista

• Din punct de vedere matematic, o listă este o secvenţă de zero sau mai multe elemente numite noduri aparţinând unui anumit tip numit tip de bază.

• Formal, o listă se reprezintă de regulă ca o secvența de elemente:

$$a_1, a_2,...,a_n$$

• unde n ≥ 0 şi fiecare ai aparţine tipului de bază.

• Numărul n al nodurilor se numeşte lungimea listei.

• Presupunând că n ≥ 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

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

• De regulă, se spune că nodul a_i se află pe poziţia i.

• Se spune că a_i precede pe a_{i+1} pentru i=1,2,...,n-1.

• Se spune că a_i succede (urmează) lui a_{i-1} pentru i=2,3,4, ...,n.

• Este de asemenea convenabil să se postuleze existenţa poziţiei următoare ultimului element al listei.

• În această idee se introduce funcţia FIN(L): TipPozitie care returnează poziţia următoare poziţiei n în lista L având n elemente.

• Se observă că FIN(L) are o distanţă variabilă faţă de începutul listei, funcţie de faptul că lista creşte sau se reduce, în timp ce alte poziţii au o distanţă fixă faţă de începutul listei

Ne amintim: Pentru a defini un tip de date abstract, în cazul de faţă 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 aplicaţiile, iar pe de altă parte natura setului depinde esenţial atât de maniera de implementare cât şi de cea de utilizare a listelor.

• În continuare se prezintă două seturi reprezentative de operatori care acţionează asupra listelor, unul restrâns şi altul extins.

Set de operatori restransi

Modelul matematic : o secventa formata din zero sau mai multe elemente numite noduri toate incadrate intr-un anumit tip -> tip de baza

NOTATII :

```
L : TipLista;
p : TipPozitie;
x : TipNod;
```

OPERATORI:

1.Fin(L:TipLista):TipPozitie; - operator care returnează poziţia următoare ultimului nod al listei, adică poziţia următoare sfârşitului ei. În cazul listei vide Fin(L)=0;
2.Insereaza(L:TipLista,x:TipNod,p:TipPozitie); - inserează în lista L, nodul x în poziţia p. Toate nodurile care urmează acestei poziţii se mută cu un pas spre poziţiile superioare, astfel încât a1,a2,...,an devine a1,a2,...,ap-1,x,ap,...an. Dacă p este Fin(L) atunci lista devine a1,a2,...an,x. Dacă p>Fin(L) rezultatul este nedefinit;
3.Cauta(x:TipNod,L:TipLista):TipPozitie; - caută nodul x în lista L şi returnează poziţia nodului. Dacă x apare de mai multe ori, se furnizează poziţia primei apariţii. Dacă x nu apare de loc se returnează valoarea Fin(L);
4.Furnizeaza(p:TipPozite,L:TipLista):TipNod; - operator care returnează nodul situat pe poziţia p în lista L. Rezultatul este nedefinit dacă p=Fin(L) sau dacă în L nu există poziţia p. Se precizează că tipul operatorului Furnizeaza trebuie să fie identic cu tipul de bază al listei;
5.Suprima(p:TipPozite,L:TipLista); - suprimă elementul aflat pe poziţia p în lista L. Dacă L este a1,a2,...,an atunci după suprimare L devine a1,a2,...ap-1,ap+1,...,an. Rezultatul este nedefinit dacă L nu are poziţia p sau dacă p=Fin(L);
6.Urmator(p:TipPozite,L:TipLista):TipPozitie; operator care returnează poziţia următoare poziţiei p în cadrul listei. Dacă p este ultima poziţie în L atunci Urmator(p,l)=Fin(L). Urmator nu este definit pentru p=Fin(L);
7.Anterior(p:TipPozite,L:TipLista):TipPozitie; - operatori care returnează poziţia anterioară poziţiei p în cadrul listei. Dacă p este prima poziţie în L atunci Anterior nu este definit;
8.Initializeaza(L:TipLista):TipPozitie;- operator care face lista L vidă şi returnează poziţia Fin(L)=0;
9.Primul(L:TipLista):TipPozitie; - returnează valoarea primei poziţii în lista L. Dacă L este vidă, poziţia returnată este Fin(L)=0;
10.TraverseazaLista(L:TipLista,ProcesareNod(...)); - parcurge nodurile listei L în ordinea în care apar ele în listă şi aplică fiecăruia procedura ProcesareNod.

OPERATORI EXTINSI

Modelul matematic: o secvenţă finită de noduri. Toate nodurile aparţin unui acelaşi tip numit tip de bază.

Fiecare nod constă din două părţi: o parte de informaţii şi o a doua parte conţinând legătura la nodul următor. O variabilă specială indică primul nod al listei.

Notaţii:
- TipNod - tipul de bază;
- TipLista - tip indicator la tipul de bază;
- TipInfo - partea de informaţii 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 informaţii a unui nod;
- b - valoare booleană; NULL - indicatorul vid.

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.*

*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 faţa 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 poziţionează pe următorul nod al listei. Dacă curent indică ultimul nod al listei el va deveni NULL.*

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

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

*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 informaţie 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 şi aplică fiecăruia procedura ProcesareNod.*

Tehnici de implementare a listelor

• De regulă pentru structurile de date fundamentale există construcţii de limbaj care le reprezintă, construcţii care îşi găsesc un anumit corespondent în particularităţile 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, observaţie valabilă şi 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 descrişi în termenii acestor structuri.

Implementarea listelor cu ajutorul structurii tablou

```
-> O lista se asimileaza cu un tablou
-> Nodurile listei sunt memorate într-o zonă contiguă în locaţii succesive
de memorie.
-> În această reprezentare:
        ◦ O listă poate fi uşor traversată.
        ◦ Noile noduri pot fi adăugate în mod simplu la sfârşitul listei.
        ◦ Inserţia unui nod în mijlocul listei presupune însă deplasarea
tuturor nodurilor următoare cu o poziţie spre sfârşitul listei pentru a face
loc noului nod.
        ◦ Suprimarea oricărui nod cu excepţia ultimului, presupune de
asemenea deplasarea tuturor celorlalte în vederea eliminării spaţiului
creat.
        ◦ Inserţia şi suprimarea unui nod necesită un efort de execuţie O(n)
```

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

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

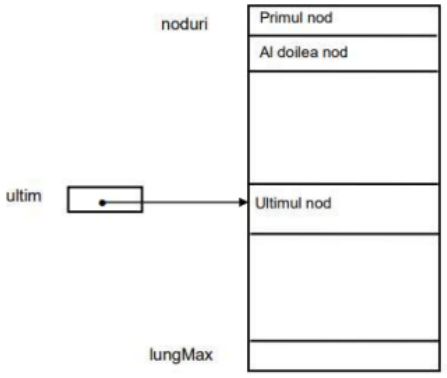
• Lungimea acestui tablou este astfel aleasă de către programator încât să fie suficientă pentru a putea păstra cea mai mare dimensiune de listă ce poate apare în respectiva aplicaţie.

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

• Cel de-al i-lea nod al listei se găseşte în cel de-al i-lea element al tabloului, pentru $1 \le i \le$ ultim.

• Poziţia în cadrul listei se reprezintă prin valori întregi, respectiv c de-a i-a poziţie prin valoarea i.

• Funcţia Fin(L) returnează valoarea ultim+1.



```
#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;

/*
        Se fac următoarele precizări:
                • Dacă se încearcă inserţia 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 găseşte
elementul căutat, Cauta returnează p
*/

void insereaza(tip_lista *l, tip_nod x, tip_pozitie p, boolean *er){
        // plaseaza pe x in pozitia p a listei
        // performanta 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("Pozitie 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
                        }
                }
        }
}

void suprima(tiplista* l,tip_pozitie p, boolean *er) {
        /*extrage elementul din poziţia p a listei*/
        tip_pozitie q; /*performanţa O(n)*/
        *er = false;
        if ((p > l->ultim +1) || (p < 0)) {
                *er = true;
```

```
                printf("pozitie invalida");
        }
        else {
                for (q = p; q <= l->ultim; q++)
                        l->noduri[q] = l->noduri[q+1]; //stergem prin
copiere la stanga
                l->ultim = l->ultim - 1; //actualizam ultima pozitie
        }
} //suprima

tip_pozitie cauta(tipnod x, tiplista l) {
        /*returnează poziţia lui x în listă*/
        tip_pozitie q;
        boolean gasit;
        tip_pozitie cauta_result;
        q = 0;
        gasit = false; /*performanţa 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;
}
```

Implementarea listelor cu ajutorul pointerilor si alocarii dinamice

```
-> O lista inlantuita este constituita dintr-o serie de obiecte, numite
nodurile listei
-> Un nod = obiect de sine statator
```

```
typedef int tip_info;

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

typedef tip_nod * tip_lista;
```

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

- O cheie care serveşte la identificarea nodului.
- Un câmp info conţinând informaţia utilă.
- Un pointer de înlănţuire la nodul următor.

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

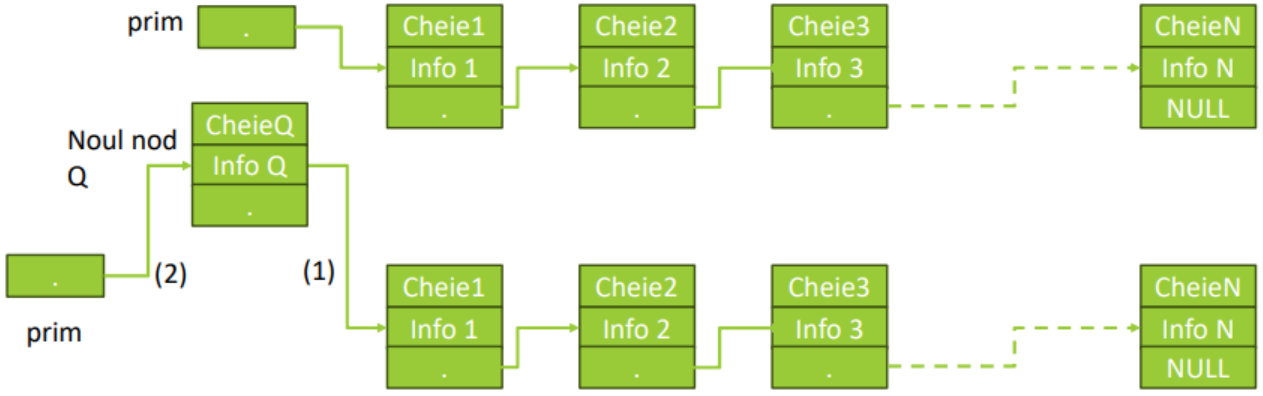
O astfel de secvență de noduri formează o listă înlănțuită. Dacă fiecare nod conține doar un câmp de legătură către succesorul său din listă, atunci lista se numește simplu înlănțuită.

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 inserţie a nodurilor şi de creeare a listelor înlănţuite

Nodurile se pot insera în diferite poziții în listă

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

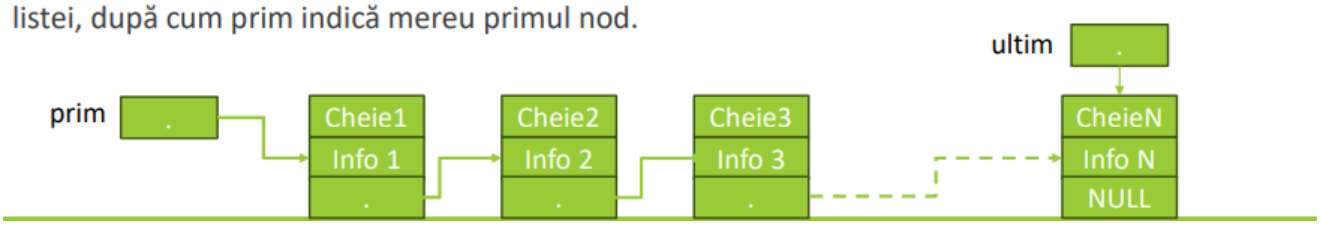



```
tip_lista insertie_fata(int cheie, tip_info info) {
        tip_nod* q; //noul nod
        q = (tip_nod*)malloc(sizeof(tip_nod)); // aloc spatiu pentru noul
nod
        if (q) {
                //daca s-a alocat spatiu cu succes
                q->cheie = cheie;
                q->info = info; //asignez 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);
```

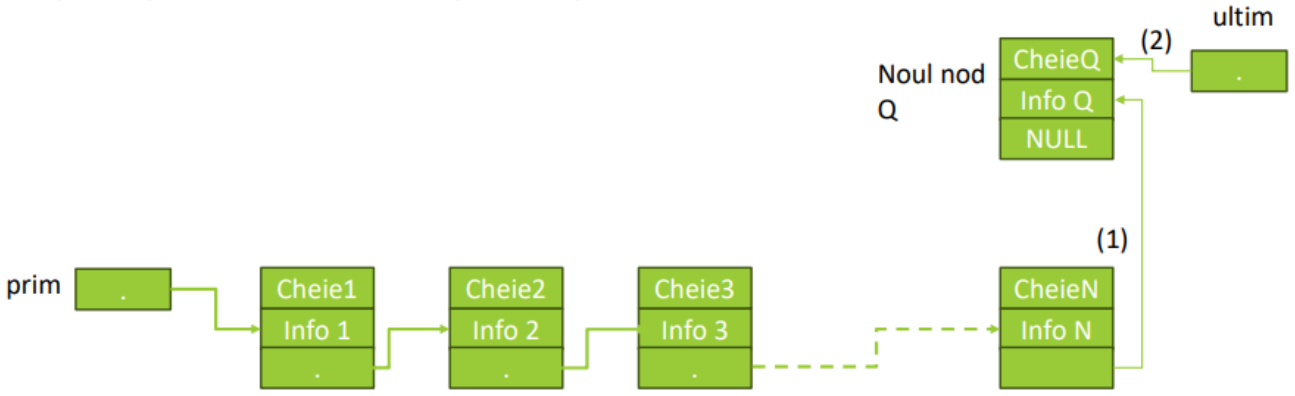
```
        prim = insertie_fata(2, 2);
        return 0;
}
```

Dacă se doreşte crearea listei în ordine naturală, atunci este nevoie de o secvenţă care inserează un nod la sfârşitul unei liste.

- Această secvenţă de program se redactează mai simplu dacă se cunoaşte locaţia 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 soluţie 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.



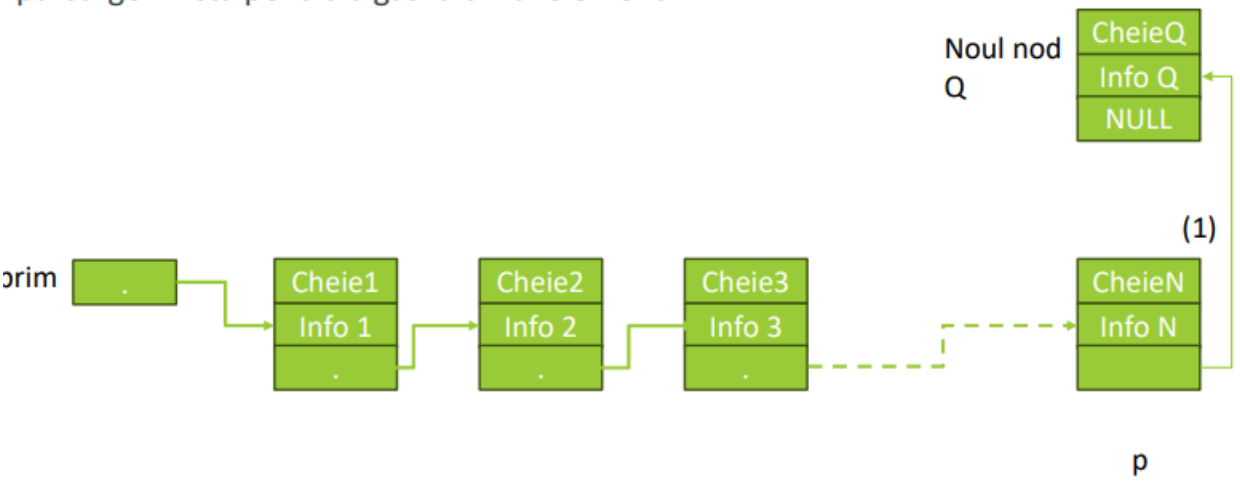
În prezența variabilei ultim, inserția la sfârșitul listei devine:




```
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) {
                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;
}
```

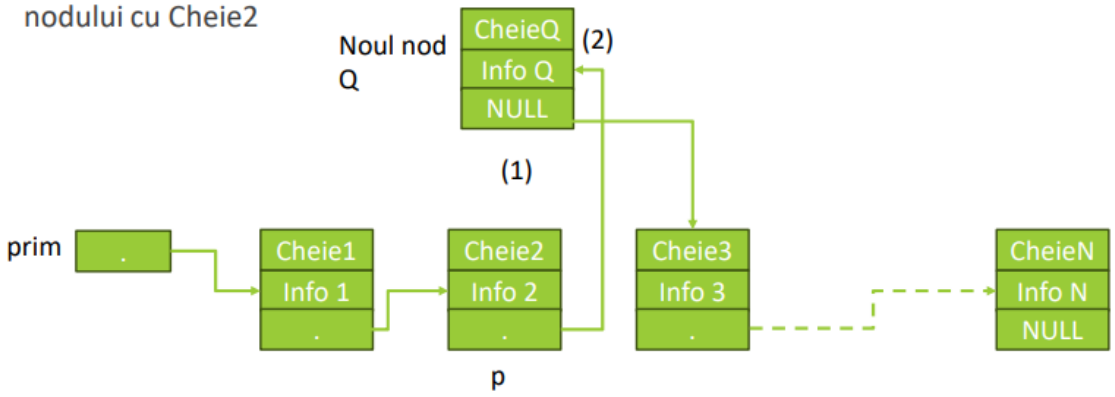
Cum se modifică funcția insertie_fata, dacă reținem și ultimul element?

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




```
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 {
                //parcurg lista pana la ultimul element
                for (p = prim; p->urm != NULL; p = p->urm);
                        p->urm = q;
        }

        return prim;
}
```

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

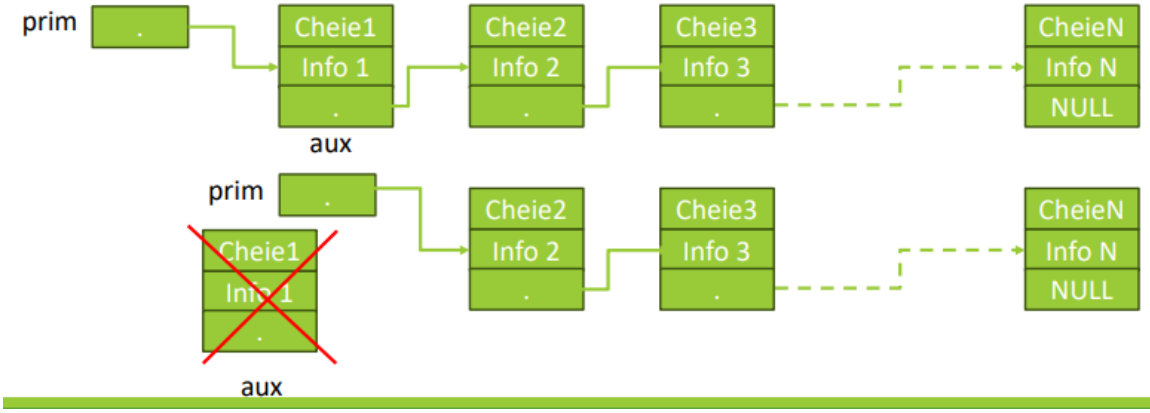



```
tip_lista insertie_dupa(int cheie, tip_info info, tip_nod *p) {
        tip_nod* q; //noul nod
        if (p == NULL) //daca se doreste de fapt insertie in fata listei
                prim = insertie_fata(cheie, info);
        else if (p == ultim) //daca se doreste de fapt adaugare la sfarsit
                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;
}

// Pentru a gasi pozitia nodului p in lista ne-am folosit de o functie de
cautare
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;
}
```

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



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

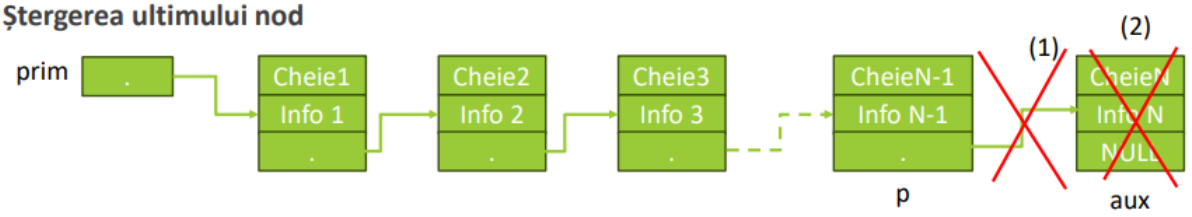
Toate cele trei operații implică un timp constant de execuție și nu depind de lungimea listei, astfel complexitatea acestei ștergeri este O(1)

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.

1. Se verifică daca lista este vidă și dacă da, nu se mai execută niciunul din pașii următori
2. Se reține primul element într-o variabilă auxiliară
3. Se actualizează variabila prim, care va indica nodul următor (prim->urm)
4. 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?

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

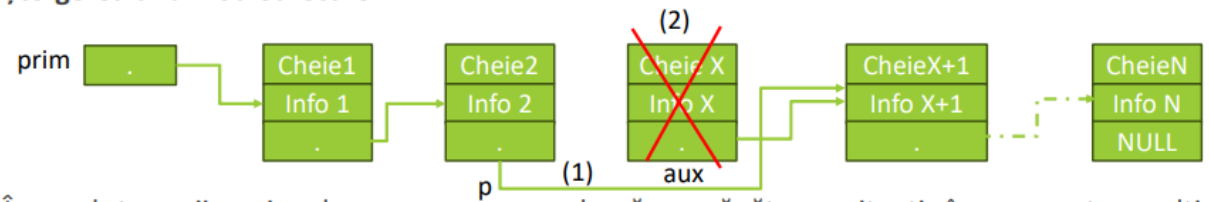
3. Se dealocă spațiul pentru elementul reținut în variabila auxiliară

(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!

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).

Cautarea

Traversarea unei liste înlănţuite. **Căutarea** unui nod într-o listă înlănţuită

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

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

O operaţie 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;
```

Implementarea listelor cu ajutorul cursorilor

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

În accepţiunea acestei implementări:

(1) Pentru toate listele ale căror noduri sunt de TipNod, se crează un tablou (Zona) având elementele de TipElement.

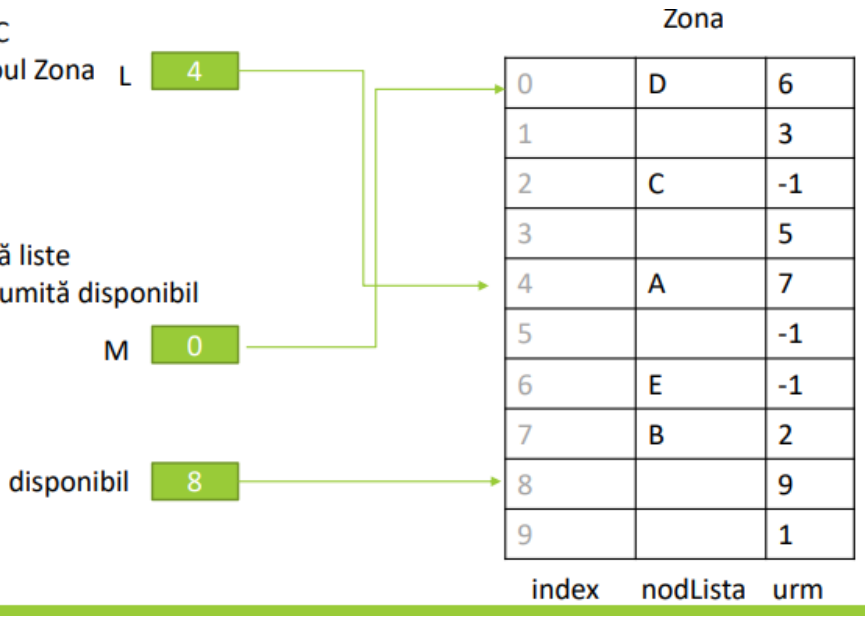
(2) Fiecare element al tabloului conţine un câmp nodLista:TipNod şi un câmp urm:TipCursor definit ca **subdomeniu** al tipului întreg.

(3) TipLista este în aceste condiţii identic cu

```
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;
```

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

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



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, ş.a.m.d.
- Valoarea -1 a unui cursor, semnifică legătura vidă, adică faptul că nu urmează nici un element

Se observă că:

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

Operatii:

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

(1) Se suprimă prima locaţie a listei disponibil.

(2) Se inserează această locaţie în poziţia dorită a listei L, actualizând valorile cursorilor implicaţi.

(3) Se asignează câmpul nodLista al acestei locaţii cu valoarea lui x.

• **Suprimarea** unui element x din lista L presupune:

(1) Suprimarea locaţiei care îl conţine pe x din lista L.

(2) Inserţia acestei locaţii în capul listei disponibil.

Atât **inserţia** cât şi **suprimarea** sunt de fapt cazuri speciale ale următoarei **situaţii**:

• Fie două liste precizate prin cursorii sursa şi destinatie.

• Fie y prima locaţie a listei indicate de sursa.

• Se **suprimă** y din lista indicată de sursa şi se **înlănţuie** pe prima poziţie a listei indicate de destinatie. Acest lucru se realizează astfel:

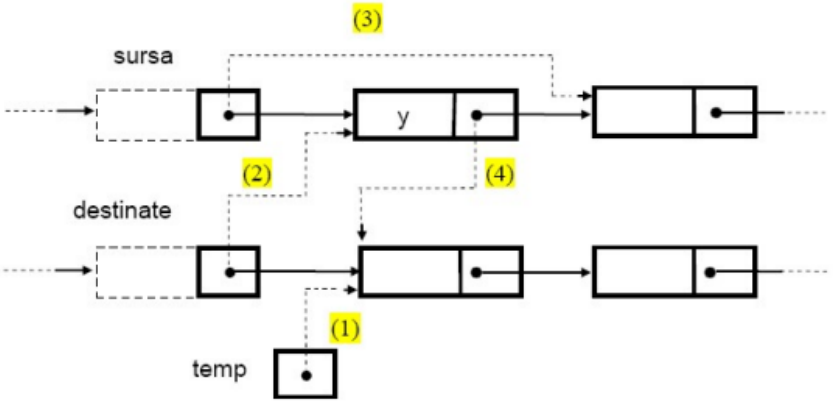
(1) Se salvează valoarea cursorului destinatie în locaţia auxiliară temp.

(2) Se atribuie valoarea cursorului sursa cursorului destinatie 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 destinatie

Reprezentarea grafică a acţiunilor anterioare este reprezentată mai jos, unde sunt prezentate legăturile înainte (linie continuă) şi după (linie întreruptă) desfăşurarea ei.




```
boolean muta(tip_cursor* sursa,tip_cursor* destinatie) {
        /*mută elementul indicat de sursă în faţa celui indicat de
destinaţie*/
        tip_cursor temp;
        boolean muta_result;
        if (*sursa==0) {
                /*performanţa O(1)*/
                printf("locaţia 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;
}

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

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

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

Comparaţie 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) Operaţiile care se doresc a fi realizate.

(3) Frecvenţa cu care sunt invocaţi operatorii.

(4) Constrângerile de timp de acces, de memorie, de performanţă, ş.a.

Observatii

Observaţii:

(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 referinţe.

(2) Aceiaşi operatori pot avea performanţe diferite în implementări diferite.

• Spre exemplu inserţia sau suprimarea unui nod precizat au o durată constantă la implementarea înlănţuită (O(1)), dar necesită o perioadă de timp proporţională 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)) şi un timp care depinde de lungimea totală, respectiv de poziţia nodului în implementarea bazată pe pointeri, referinţe sau cursori (O(n)).

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

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

În funcţie de circumstanţe una sau alta dintre implementări poate fi mai mult sau mai puţin avantajoasă.

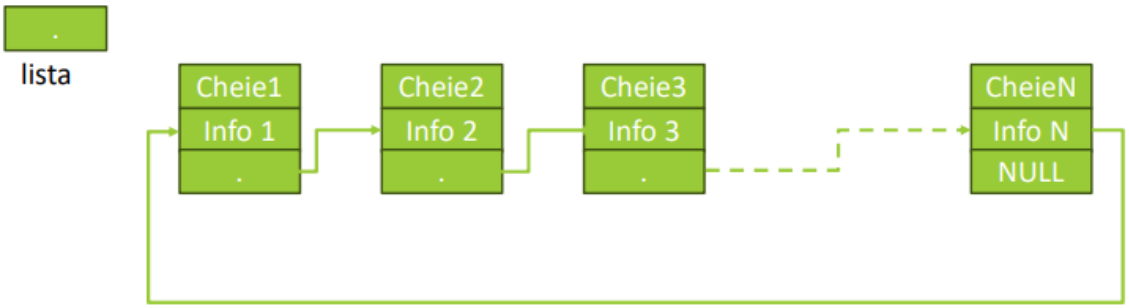
Structuri de date derivate din TDA listă

1. Liste circulare
2. Lste dublu inlantuite
3. Stiva
4. Coada

Liste circulare

-> sunt liste inlantuite ale caror inlantuiri se inchid
-> În aceste condiţii se pierde noţiunea de început şi sfârşit, lista fiind referită de un pointer care se deplasează de-a lungul ei
-> Listele circulare ridică unele probleme referitoare la inserţia primului nod în listă şi la suprimarea ultimului nod.

Liste circulare

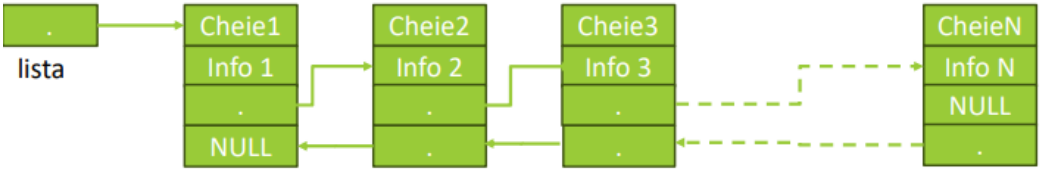


Ce nod ar trebui să refere nodul lista (primul, ultmul, orice nod)?

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

Liste dublu inlantuite

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



- Preţul care se plăteşte este:

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

(2) O oarecare creştere a complexităţii funcţiilor 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 secvenţa

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

Exemplu de operații asupra listei dublu înlănțuite:

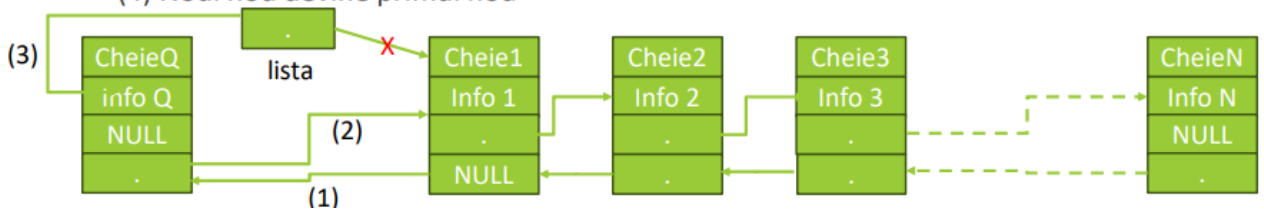
Inserarea unui element pe prima poziție:

(1) Se alocă spațiu pentru noul nod și ce 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




```
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;
}
```

Exemplu de operații asupra listei dublu înlănțuite:

Suprimarea elementului situat în poziţia p a unei liste dublu înlănţuite:

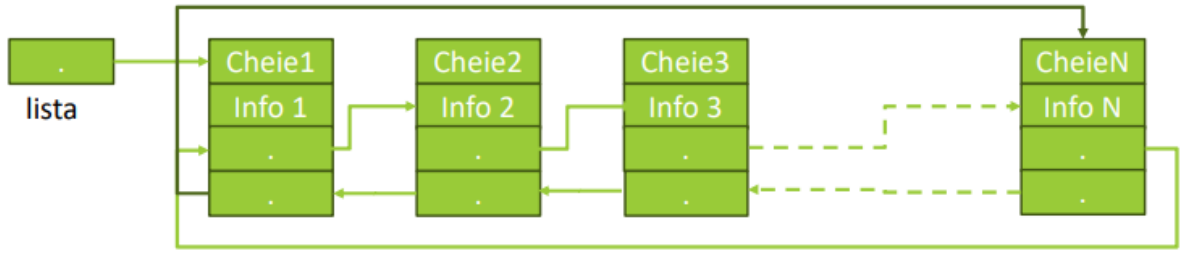
• Considerând că nodul suprimat nu este nici primul nici ultimul nod al listei:

(1) Se localizează nodul ant(erior) şi 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 spaţiul de memorie afectat lui poate fi reutilizat în regim de alocare dinamică a memoriei.

Listele dublu înlănțuite pot fi implementate și ca liste circulare



Stive

```
-> Structura liniara care poate fi accesata doar la un capat al sau pentru a insera si a extrage niste 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.

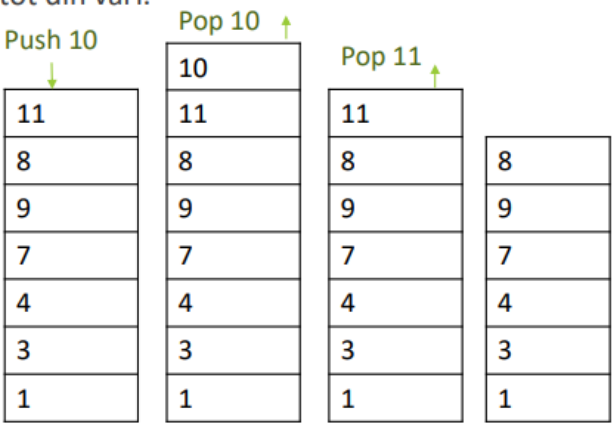
În limba engleză folosim termenul de stack sau LIFO (last in, first out)

Definirea TDA Stivă presupune precizarea:

(1) Modelului matematic asociat.

(2) Notaţiilor utilizate.

(3) Operatorilor definiţi pentru acest tip.



TDA Stivă

Modelul matematic: o secvenţă finită de noduri. Toate nodurile aparţin unui acelaşi tip numit tip de bază. O stivă este de fapt o listă specială în care toate inserţiile şi toate suprimările se fac la un singur capăt care se numeşte vârful stivei.

Notaţii:

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 şi fals în caz contrar.*

```
// Varianta restransa
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;
```

```
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;
```

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

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

```
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*/

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;
```

```
}

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*/

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*/
```

Ce implică o implementare cu pointeri?

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

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 acesteia
  - 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**) şi sunt suprimate la celălalt (**început**).

• Cozile se mai numesc liste "**FIFO**" ("**first-in first-out**") adică liste de tip "**primul-venit-primul-servit**".

• Operaţiile care se execută asupra **cozii** sunt analoage celor care se realizează asupra **stivei** cu diferenţa că inserările se fac la **spatele** cozii şi **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.

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

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 secvenţă finită de noduri. Toate nodurile aparţin unui aceluiaşi tip numit tip de bază. O coadă este de fapt o listă specială în care toate inserţiile se fac la un capăt (spate) şi toate suprimările se fac la celălalt capăt (cap).

Notaţii:

C: TipCoada;

x: TipElement; b: boolean;

Operatori

Set 1

1.***Initializeaza****(C: TipCoada){: TipPozitie};* - face coada *C* vidă.

2.***Cap****(C: TipCoada): TipElement;* - funcţie care returnează primul element al cozii *C*.

3.***Adauga****(x: TipElement, C: TipCoada);* - inserează elementul *x* la spatele cozii *C*.

4.***Scoate****(C: TipCoada);* - suprimă primul element al lui *C*.

5.***Vid****(C: TipCoada): boolean;* - este adevărat dacă şi numai dacă *C* este o coadă vidă.

Set 2

1.***CreazaCoadaVida****(C: TipCoada);* - face coada *C* vidă.

2.***CoadaVida****(C: TipCoada): boolean;* - este adevărat dacă şi numai dacă *C* este o coadă vidă.

3.***CoadaPlina****(C: TipCoada): boolean;* - este adevărat dacă şi numai dacă *C* este o coadă plină (operator dependent de implementare).

4.***InCoada****(C: TipCoada x: TipElement);* - inserează elementul *x* la spatele cozii *C* (***EnQueue*****).**

5.***DinCoada****(C: TipCoada, x: TipElement);* - suprimă primul element al lui *C* (***DeQueue****).*

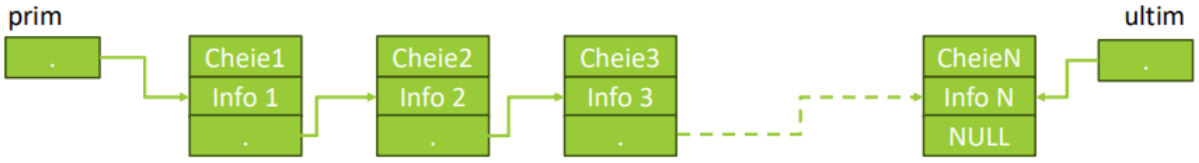
```
// Set 1
TYPE TipCoada = TipLista;
VAR C: TipCoada;
p: TipPozitie;
x: TipElement;
b: boolean;
{Initializeaza(C);}     p:= Initializeaza(C);
{Cap(C);}               x:= Furnizeaza(Primul(C),C);
{Adauga(x,C);}          Insereaza(C,x,Fin(C));
{Scoate(C);}            Suprima(Primul(C),C);
{Vid(C),C);}            b:= Fin(C)=0;
```

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

• Pornind însă de la observaţia că inserţiile se fac numai la **spatele** cozii, funcţia ***Adauga*** poate fi concepută mai eficient.

• Astfel, în loc de a parcurge de fiecare dată coada de la început la sfârşit atunci când se doreşte adăugarea unui element, se va păstra un **pointer** la **ultimul element al cozii**.

• De asemenea, ca şi la toate tipurile de liste, se va păstra şi pointerul la **primul element al listei** utilizat în execuţia comenzilor ***Cap*** şi ***Scoate***.




```
/*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;
```

```
void initializeaza(tip_coada* c) {
        c->prim= NULL;
        c->ultim= NULL;
        /*nici prim nici spate nu indica spre vreun element*/
}
/*initializeaza*/

boolean vid(tip_coada c) {
        if (c.prim == NULL)
                return true;
        else
                return false;
}
/*vid*/

tip_element cap(tip_coada c) {
        return c.prim->element;
}
/*cap*/

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

void scoate(tip_coada *c, int *er){
        if(vid(*c))
                *er = true;
        else{
                *er = false;
                c->prim = c->prim->urm;
        }
}
// scoate
```

Cozi circulare

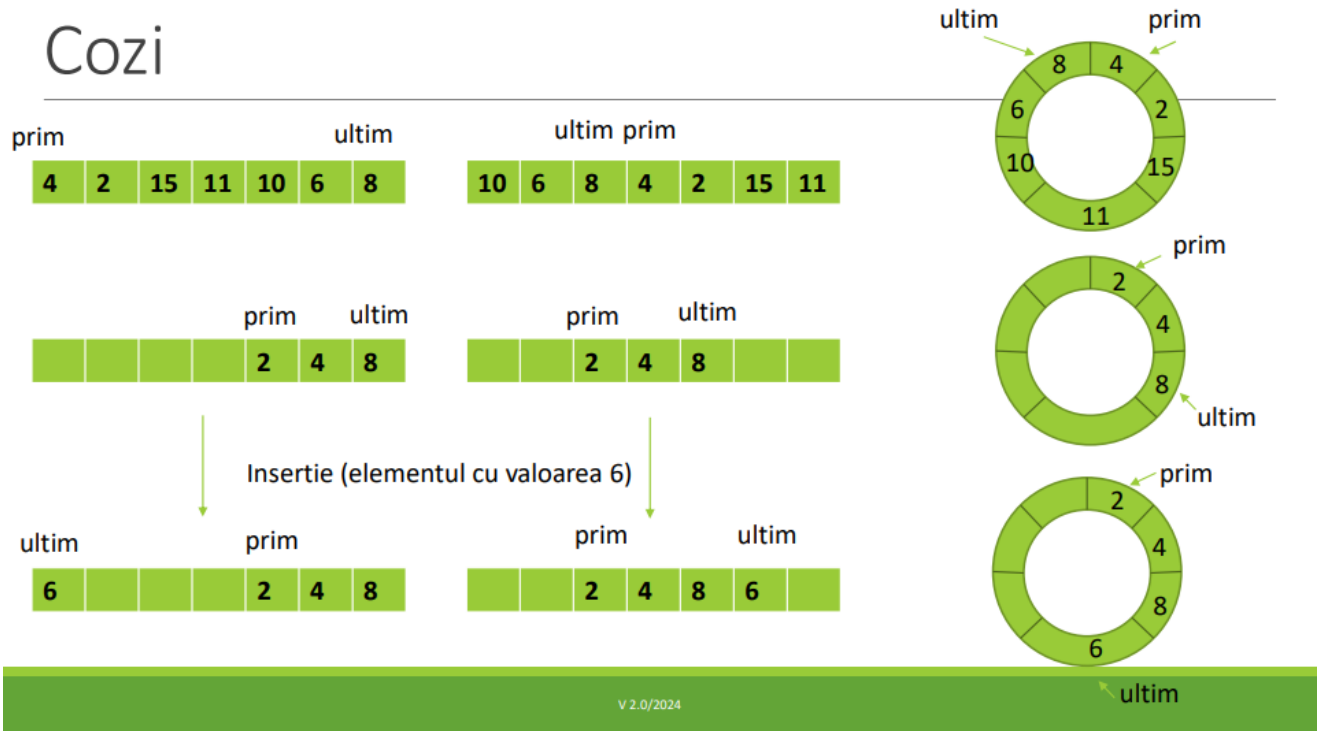
Cozi circulare

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

• Într-adevăr, indicatorul la ultimul element al listei, permite implementarea **simplă**, într-un număr fix de paşi, a operaţiei ***Adauga***.

• Execuţia operaţiei ***Scoate*** presupune însă **mutarea** întregii cozi cu o poziţie în tablou, operaţie care necesită un timp O*(n)*, dacă coada are lungimea n.

• Pentru a depăşi acest dezavantaj se poate utiliza **o structură de tablou circular** în care prima poziţie urmează ultimei



• **Adăugarea** unui element presupune mişcarea indicatorului C.ultim cu o poziţie în sensul acelor de ceasornic şi introducerea elementului în această poziţie.

• **Scoaterea** unui element din coadă, presupune simpla mutare a indicatorului C.prim cu o poziţie în sensul rotirii acelor de ceasornic.

• Astfel coada se roteşte în sensul rotirii acelor de ceasornic după cum se adaugă sau se scot elemente din ea.

• În aceste condiţii atât funcţia Adauga cât şi Scoate pot fi redactate astfel încât să presupună un număr fix de paşi de execuţie, 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**.

• Acest mod de implementare ridică însă o problemă referitoare la sesizarea **cozii vide** şi a **celei pline**.

• Presupunând că structura **coadă** este **plină** (conţine LungimeMaxima elemente), indicatorul C.ultim va indica **poziţia 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 şi C.ultim indică aceeaşi poziţie.

• Dacă se scoate **singurul** element, C.prim avansează cu o poziţie înainte(în sensul acelor de ceasornic) indicând **coada vidă**.

• Se observă însă că această situaţie este **identică** cea anterioară care indică coada plină, adică C.prim se află cu o poziţie în faţa lui C.ultim, la parcurgerea în sensul arcelor de ceasornic a cozii.

• În vederea rezolvării acestei situaţii, în tablou se vor introduce **doar** LungimeMaxima-1 elemente, deşi în tablou există LungimeMaxima poziţii.

• 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 poziţie.

• Evident avansările se realizează în sensul acelor de ceasornic al parcurgerii cozii.

```
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;
```

```
void initializeaza(tip_coada* c) {
        c->dimensiune = lungime_maxima + 1;
        c->prim = 1;
        c->ultim = 0;
}

tip_element cap(tip_coada c) {
        return c.coada[c.prim];
}

boolean vid(tip_coada c) {
        if (((c.ultim + 1) % c.dimensiune) == c.prim)
                return true;
        else return false;
}

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

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

Cozi bazate pe prioritate

• Coada bazată pe prioritate (“priority queue”) este structura de date abstractă care permite inserţia unui element (poate fi sau nu ordonată în funcție de prioritatea sa) şi **suprimarea celui mai prioritar element**.

• O astfel de structură diferă atât faţă de structura coadă (din care se suprimă primul venit, deci cel mai vechi) cât şi faţă de stivă (din care se suprimă ultimul venit, deci cel mai nou).

• De fapt cozile bazate pe prioritate pot fi concepute ca şi structuri care generalizează cozile şi stivele.

• Aplicaţiile 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

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 **priorităţi**, definirea acesteia poate fi:

Modelul matematic: o secvenţă finită de noduri. Toate nodurile aparţin unui aceluiaşi 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 inserţia ordnată după un criteriu şi suprimarea doar a celui mai prioritar nod.

Notaţii:

q: TipCoadaBazataPePrioritate;

x: TipElement;

Operatori:

1.***Initializează***(*q: TipCoadaBazataPePrioritate);* - face coada *q* vidă.

2.***Inserează***(*x: TipElement, q: CoadaBazataPePrioritate);* - inserţia unui nou element *x* în coada *q.*

3.***Extrage***(*q: CoadaBazataPePrioritate): TipElement*; - extrage cel mai prioritar element al cozii *q*.

4.***Inlocuieşte***(*q: CoadaBazataPePrioritate, x: TipElement): TipElement;* - înlocu- ieşte cel mai prioritar element al cozii *q* cu elementul *x,* mai puţin în situaţia 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* şi 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 şi numai dacă *q* este o coadă vidă.

Operatorul ***Inlocuieşte*** constă dintr-o **inserţie** urmată de **suprimarea** celui mai prioritar element.

• Este o operaţie diferită de succesiunea suprimare-inserţie deoarece necesită creşterea pentru moment a dimensiunii cozii cu un element.

• Acest operator se defineşte separat deoarece în anumite implementări poate fi conceput foarte eficient.

• În mod analog operatorul ***Schimbă*** poate fi implementat ca şi o **suprimare**, urmată de o **inserţie**, iar generarea unei cozi ca şi o succesiune de operatori ***Insereză***.

• **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 performanţe diferite pentru operatorii specifici.

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 numeşte multilistă, o structură de date ale cărei noduri conţin mai multe câmpuri de înlănţuire.

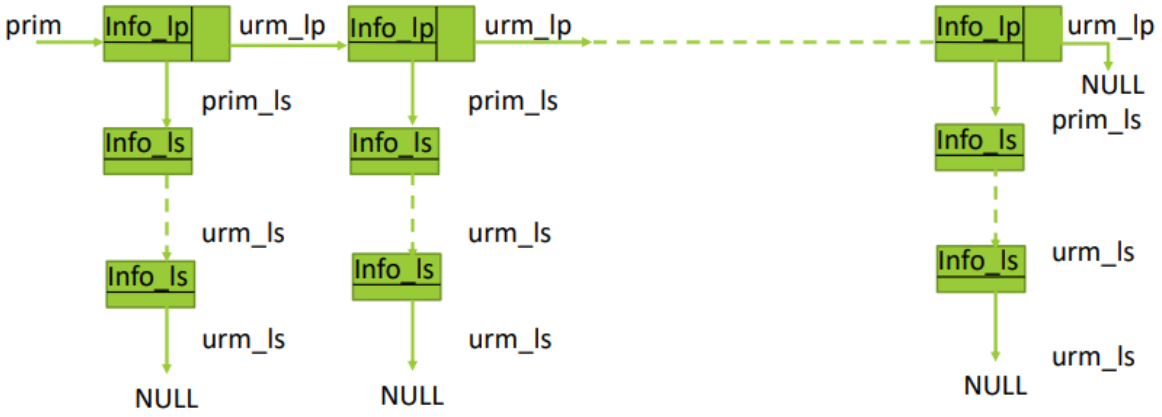
• Cu alte cuvinte, un nod al unei astfel de structuri poate aparţine în acelaşi timp la mai multe liste înlănţuite simple.

• În literatura de specialitate termenii consacraţi pentru a desemna o astfel de structură sunt:

• "Multilist".

• "Multiply linked list"

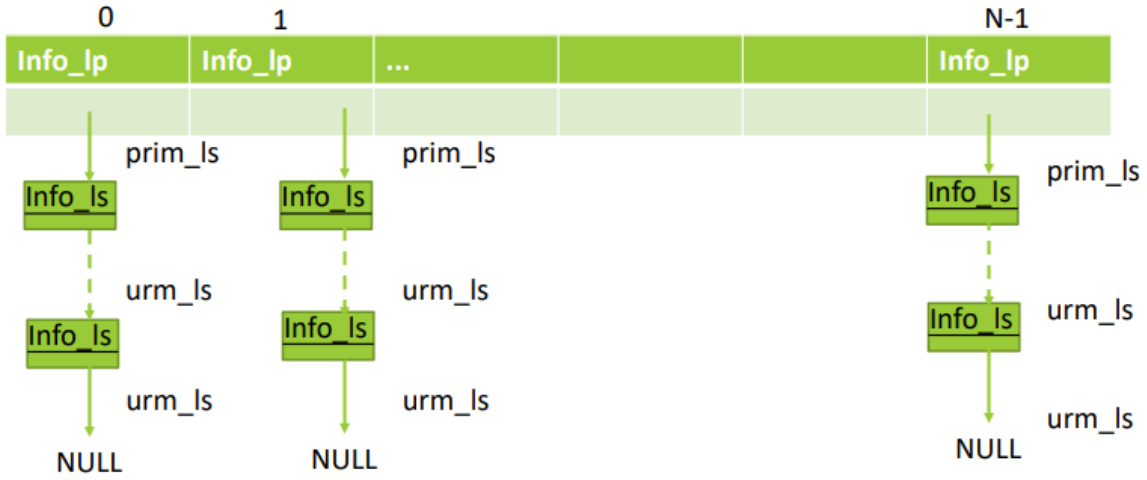
Un exemplu de multilistă în care avem o listă principală și fiecare nod al listei principale indică către o listă secundară:



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
```

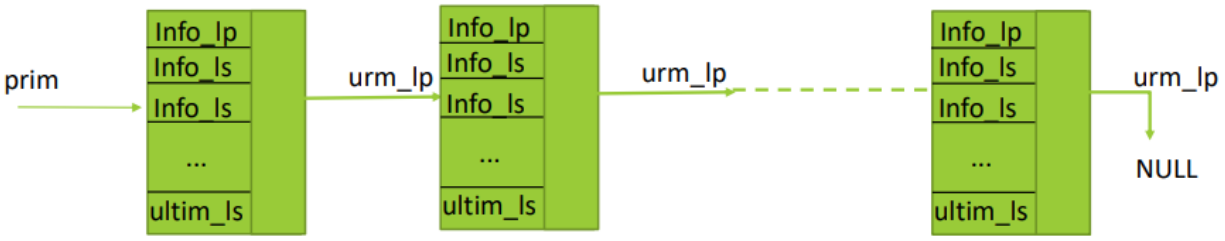
Un exemplu de lista principală implementată cu ajutorul tablourilor și liste secundare implementate cu ajutorul pointerilor:



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;
```

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



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
```

Avantajul utilizării unor astfel de structuri este evident.

• Prezenţa mai **multor înlănţuiri** într-un acelaşi nod, respectiv **apartenenţa simultană** a aceluiaşi nod la mai multe liste, asigură acestei structuri de date o **flexibilitate** deosebită.

• Acest avantaj, coroborat cu o manipulare relativ **facilă** specifică structurilor înlănţuite este exploatat la implementarea **bazelor de date**, cu precădere a celor relaţionale.

• 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 conţin de regulă un **număr redus** de elemente restul fiind poziţionate de obicei pe zero.

• Din acest motiv memorarea lor în forma obişnuită 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 in 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 si 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 adaugate atât în față, cât și în spate.