



# LISTE

Șl. Dr. Ing. Șerban Radu

Departamentul de Calculatoare

Facultatea de Automatică și Calculatoare

# Liste înlănțuite

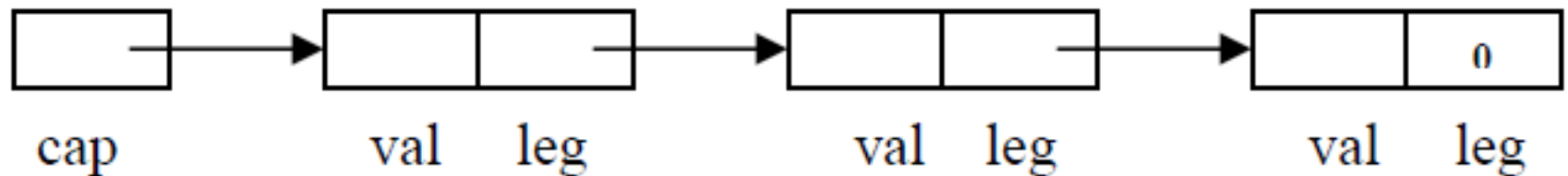
- O listă înlănțuită (**Linked List**) este o colecție de elemente, alocate dinamic, dispersate în memorie, dar legate între ele prin pointeri, ca într-un lanț
- O listă înlănțuită este o structură dinamică, flexibilă, care se poate extinde continuu, fără ca utilizatorul să fie preocupat de posibilitatea depășirii unei dimensiuni estimate initial (singura limită este mărimea zonei "heap" din care se alocă memorie)

# Liste simplu înlănțuite

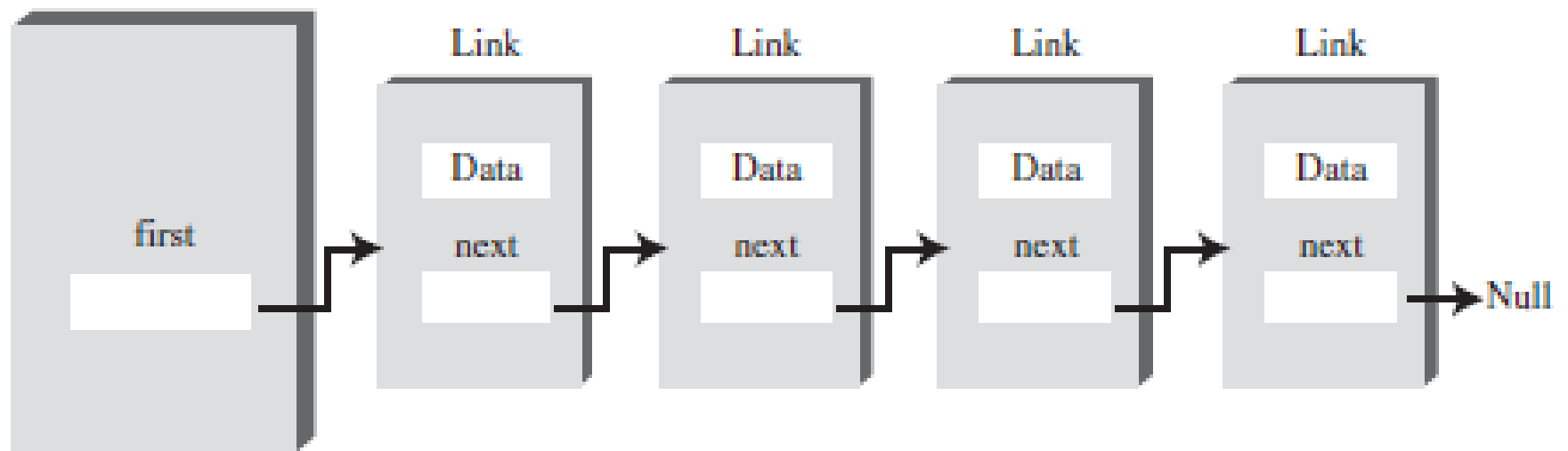
- Într-o **listă simplu înlănțuită** fiecare element al listei conține **adresa elementului următor din listă**
- Ultimul element poate conține ca adresă de legătură fie constanta NULL (un pointer către nicăieri), fie adresa primului element din listă (dacă este o listă circulară), fie adresa unui element terminator cu o valoare specială

# Liste simplu înlănțuite

- Adresa primului element din listă este memorată într-o variabilă pointer cu nume (alocată la compilare) și numită cap de listă (list head)

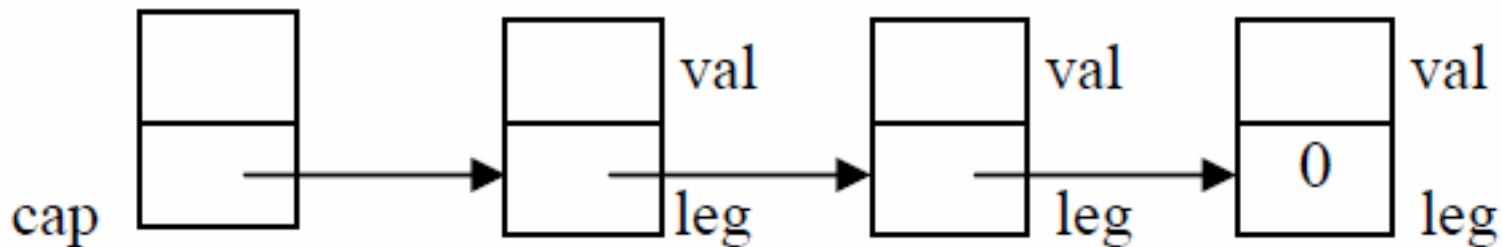


## Linked List



# Liste simplu înlănțuite

- Este posibil ca variabila cap de listă să fie tot o structură, și nu un pointer



# Liste simplu înlănțuite


- Un element din listă (numit și nod de listă) este de un tip **structură** și are (cel puțin) **două câmpuri**:
  - un câmp de date (sau mai multe)
  - un câmp de legătură
- Exemplu:

```
typedef int T;                // orice tip numeric
typedef struct nod {
    T val ;                   // câmp de date
    struct nod *leg ;         // câmp de legătură
} Nod;
```

Conținutul și tipul câmpului de date depind de  
Informațiile memorate în listă, deci de aplicația  
care o folosește

Toate funcțiile care urmează sunt direct aplicabile  
dacă tipul de date nedefinit  $T$  este un tip numeric  
(aritmetic)





Tipul “List” poate fi definit ca un  
tip pointer sau ca un tip structură:

```
typedef Nod* List;      // listă ca pointer  
typedef Nod List;      // listă ca structură
```

O listă înlănțuită este complet caracterizată de  
variabila **cap de listă**, care conține adresa  
primului nod (sau a ultimului nod, într-o listă circulară)

Variabila care definește o listă este de obicei  
o **variabilă pointer**, dar poate fi și o **variabilă structură**


# Operații cu liste înlănțuite

- Inițializare listă (a variabilei cap de listă)
  - *initL (List &)*
- Adăugarea unui nou element la o listă
  - *addL (List &, T)*
- Eliminarea unui element dintr-o listă
  - *dell (List &, T)*
- Căutarea unei valori date într-o listă
  - *findL (List &, T)*



# Operații cu liste înlănțuite

- Test de listă vidă
  - emptyL (List)
- Determinarea dimensiunii listei
  - sizeL (List)
- Parcurgerea tuturor nodurilor din listă (traversare listă)



Accesul la elementele unei liste cu legături este strict secvențial, pornind de la primul element și trecând prin toate nodurile precedente celui căutat, sau pornind din elementul "curent" al listei, dacă se memorează și adresa elementului curent al listei


Pentru parcurgere se folosește o variabilă cursor, de tip **pointer către nod**, care se inițializează cu adresa cap de listă

Pentru a avansa la următorul element din listă se folosește adresa din câmpul de legătură al nodului curent:

```
Nod *p, *prim;
    p = prim;           // adresa primului element
    ...
    p = p→leg;          // avans la următorul nod
```


Exemplu de afișare a unei liste înlănțuite  
definite prin adresa primului nod:

```
void printL ( Nod* lst) {
    while (lst != NULL) {
        // repetă cât timp există ceva la adresa lst
        printf ("%d ", lst→val);
        // afișare date din nodul de la adresa lst
        lst = lst→leg;
        // avans la nodul următor din listă
    }
}
```



Căutarea secvențială a unei valori date într-o listă este asemănătoare operației de afișare, dar are ca rezultat adresa nodului ce conține valoarea căutată:

```
// căutare într-o listă neordonată
Nod* findL (Nod* lst, T x) {
    while (lst != NULL && x != lst→val)
        lst = lst→leg;
    return lst;      // NULL dacă x nu e găsit
}
}
```




Funcțiile de adăugare, ștergere și inițializare a listei modifică adresa primului element (nod) din listă

Dacă lista este definită printr-un pointer, atunci funcțiile primesc un pointer și modifică (uneori) acest pointer

Dacă lista este definită printr-o variabilă structură, atunci funcțiile modifică structura

În varianta **listelor cu element santinelă**, nu se mai modifică variabila cap de listă, deoarece conține mereu adresa elementului santinelă, creat la inițializare



Operația de inițializare a unei liste stabilește adresa de început a listei, fie ca NULL pentru liste fără santinelă, fie ca adresă a elementului santinelă

Crearea unui nou element de listă necesită alocarea de memorie, prin funcția **malloc**

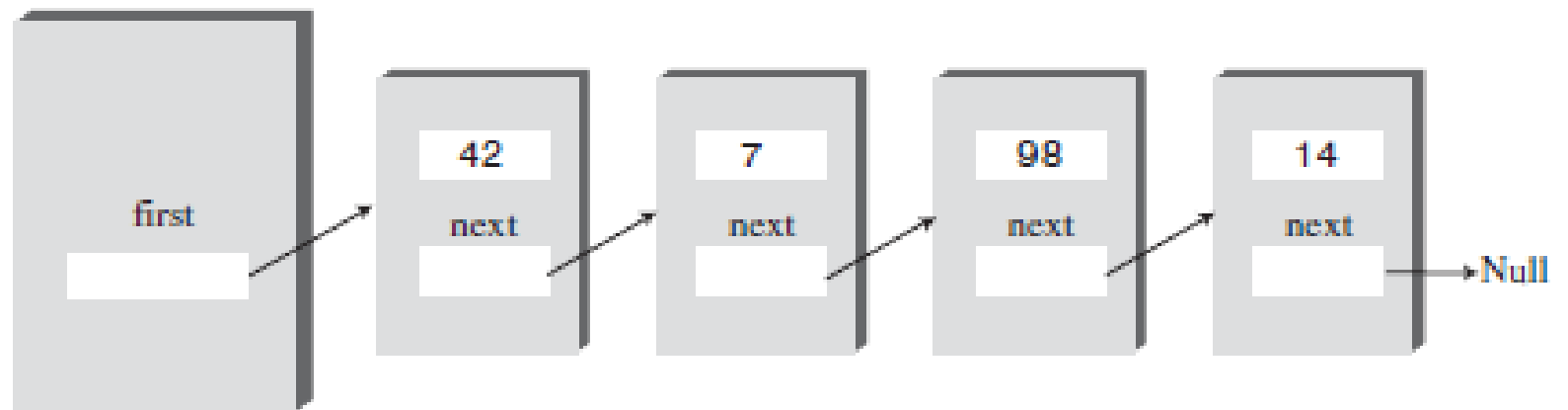
Verificarea rezultatului cererii de alocare (NULL, dacă alocarea este imposibilă) se poate face printr-o instrucțiune **if**

```
nou = (Nod*) malloc( sizeof(Nod));
```

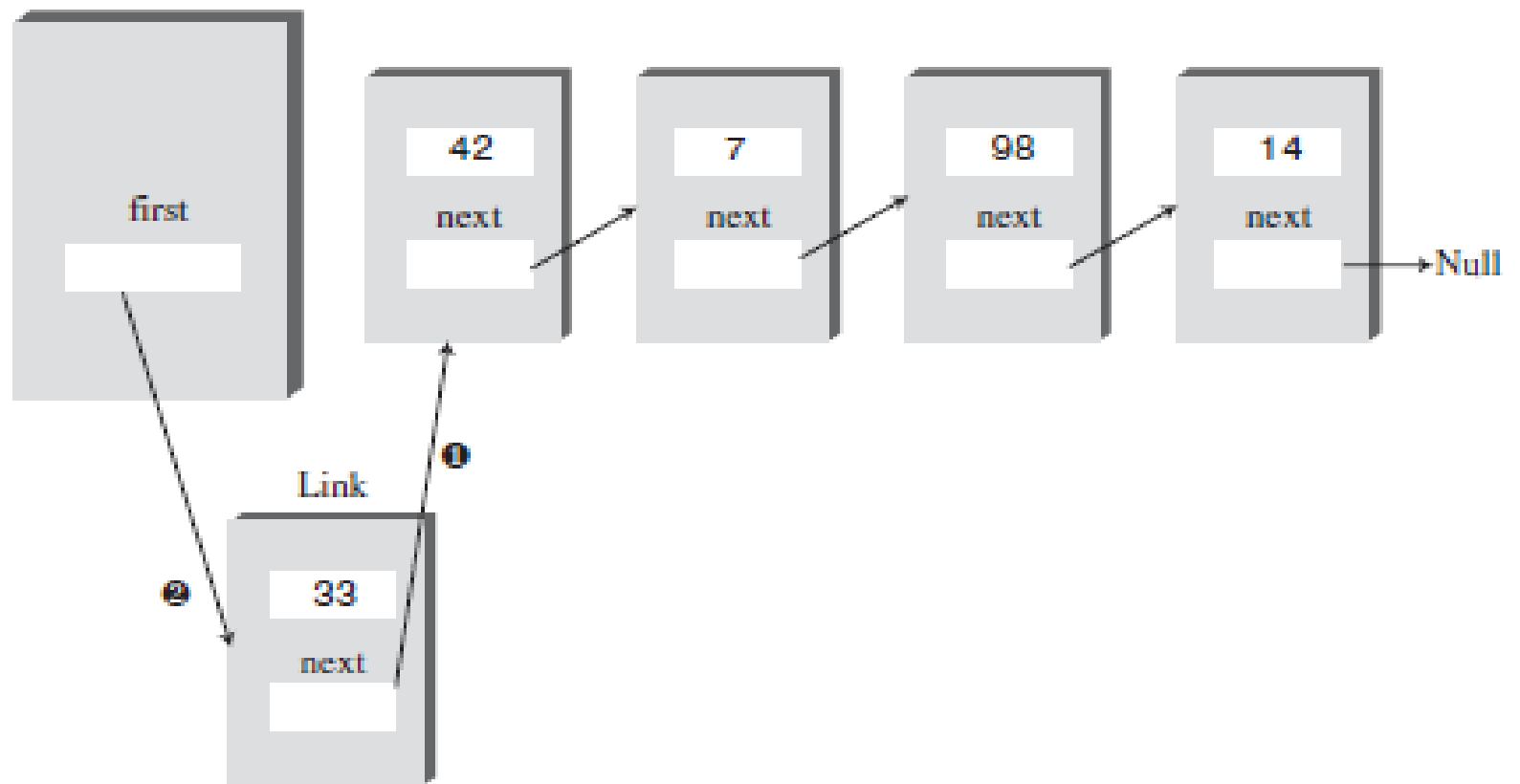


# Adăugarea unui element la o listă înlănțuită

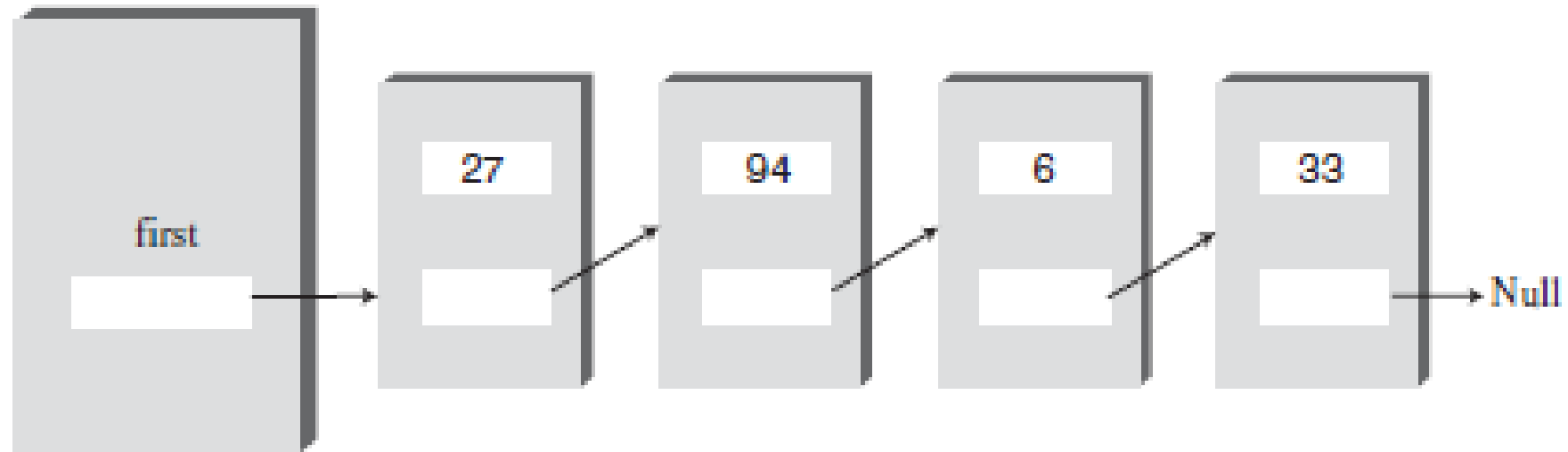
- Mereu la începutul listei
- Mereu la sfârșitul listei
- Într-o poziție determinată de valoarea noului element
- Dacă ordinea datelor din listă este indiferentă pentru aplicație, atunci cel mai simplu este ca adăugarea să se facă numai la începutul listei
- Afișarea valorilor din listă se face în ordine inversă introducerii în listă



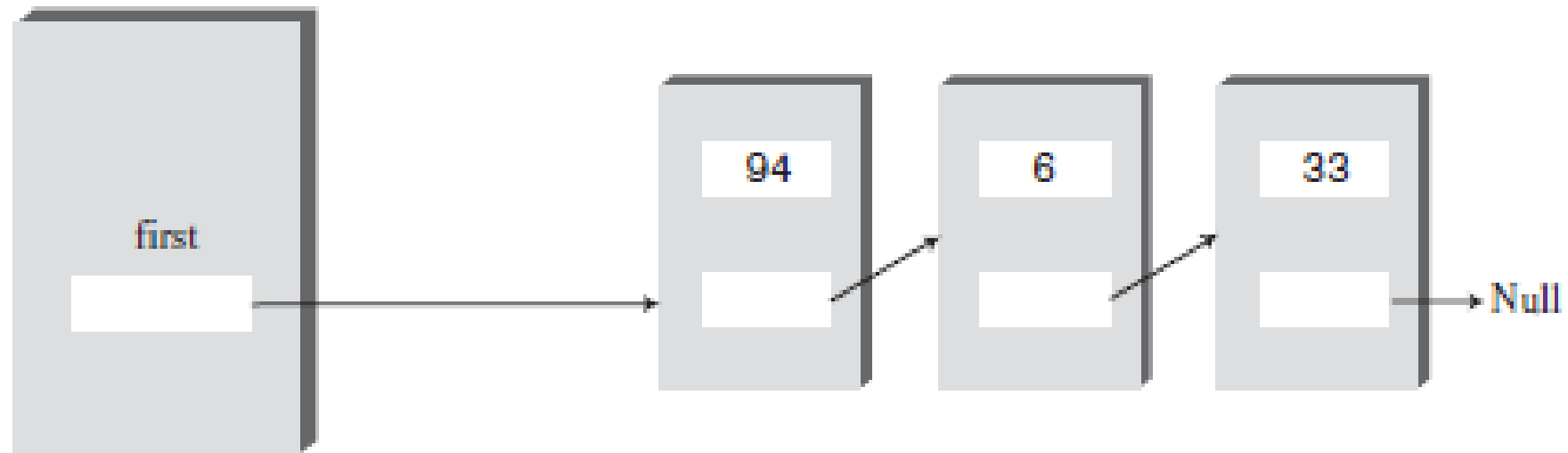
a) Before Insertion



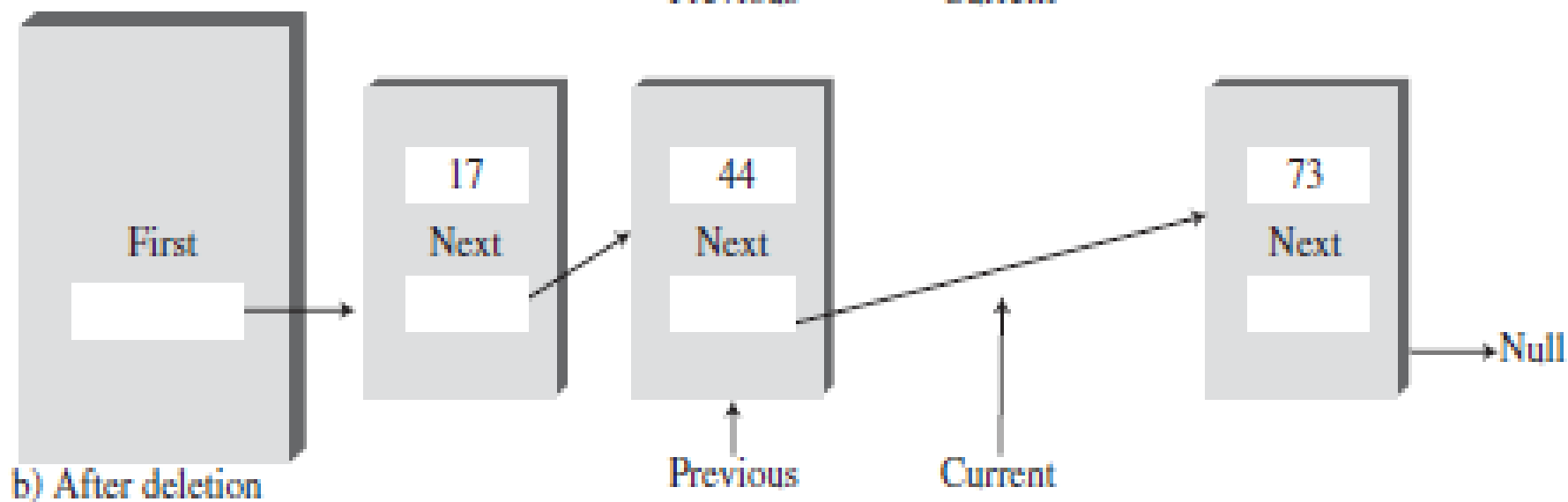
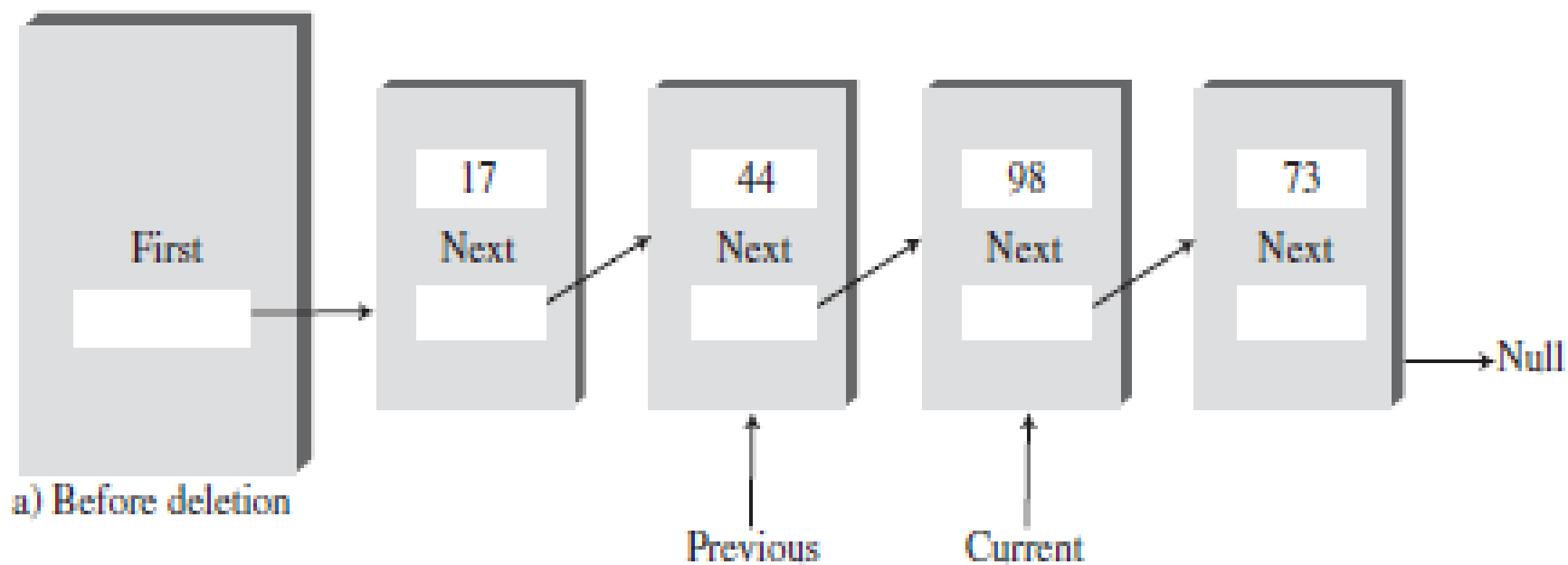
b) After Insertion



a) Before Deletion



b) After Deletion





Exemplu de creare și afișare a unei liste înlănțuite,  
cu adăugare la început de listă

Lista va conține valori numerice, care sunt introduse  
de la tastatură, pe rând, până când se introduce o literă

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
#include <stdlib.h>
```

```
typedef int T; // orice tip numeric
```


```
typedef struct nod {
```

```
    T val; // câmp de date
```

```
    struct nod *leg; // câmp de legătură
```


```
} Nod;
```

```
typedef Nod* List; // pt a permite redefinirea tipului "List"
```



```
int main () {  
    List lst;  
    int x;  
    Nod* nou;      // nou=adresa element nou  
    lst = NULL;    // inițializare lista vidă  
    printf("Introduceti valoarea elementului din lista = ");
```

```
while (scanf("%d", &x) > 0) {  
    nou = (Nod*)malloc(sizeof(Nod)); // alocă memorie  
    nou->val = x;  
    nou->leg = lst;  
    lst = nou;           // noul element este primul  
    printf("Introduceti valoarea elementului din lista = ");  
}  
while (lst != NULL) {           // afișare listă  
    printf("%d\n", lst->val);  
    // în ordine inversă celei de adaugare  
    lst = lst->leg;  
}  
getch();  
}
```




Operațiile elementare cu liste se scriu ca funcții,  
pentru a fi reutilizate în diferite aplicații  
Pentru comparație se prezintă trei dintre  
posibilitățile de programare a acestor funcții  
pentru liste, cu **adăugare și eliminare de la  
începutul listei**

Vezi demonstrația LinkList

Prima variantă este pentru o listă definită printr-o  
variabilă structură, de tip **Nod**:



```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef int T;                                // orice tip numeric
typedef struct nod {
    T val;                                    // câmp de date
    struct nod *leg;                          // câmp de legătură
} Nod;
void initS ( Nod * s) {                      // initializare listă
    s->leg = NULL;                            // s=var. cap de listă
}
//testează dacă lista e vidă
int emptyS(Nod * s) {
    return (s->leg == NULL);
}
```



```
// pune în lista un element  
void push (Nod * s, int x) {  
    Nod * nou = (Nod*)malloc(sizeof(Nod));  
    nou->val = x;  
    nou->leg = s->leg;  
    s->leg = nou;  
}
```



// scoate din lista un element

```
int pop (Nod * s) {
```

```
    Nod * p;
```

```
    int rez;
```

```
    p = s->leg;      // adresa primului element
```


```
    rez = p->val;     // valoarea primului element
```

```
    s->leg = p->leg; // adresa element urmator
```


```
    free (p) ;
```

```
    return rez;
```

```
}
```



```
// utilizzare
int main () {
    Nod st;
    int x;
    initS(&st);
    for (x = 0; x < 11; x++)
        push(&st, x);
    while (! emptyS(&st))
        printf ( "%d\n", pop(&st));
    getch();
}
```



A doua variantă folosește un pointer ca variabilă cap de listă și nu folosește argumente de tip referință:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef int T;           // orice tip numeric
typedef struct nod {
    T val;               // câmp de date
    struct nod *leg;     // câmp de legătură
} Nod;
```

```
void initS ( Nod ** sp) {
```

```
    *sp = NULL;
```

```
}
```

```
//testează daca lista e vidă
```

```
int emptyS(Nod * s) {
```

```
    return (s == NULL);
```

```
}
```

```
// pune in stiva un element
```

```
void push (Nod ** sp, int x) {
```


```
    Nod * nou = (Nod*)malloc(sizeof(Nod));
```

```
    nou->val = x;
```


```
    nou->leg = *sp;
```

```
    *sp = nou;
```

```
}
```




```
// scoate din stivă un element
int pop (Nod ** sp) {
    Nod * p;
    int rez;
    rez = (*sp)->val;
    p = (*sp)->leg;
    free (*sp) ;
    *sp = p;
    return rez;
}
```




```
// utilizzare
int main () {
    Nod* st;
    int x;
    initS(&st);
    for (x = 0; x < 11; x++)
        push(&st, x);
    while (! emptyS(st))
        printf( "%d\n", pop(&st));
    getch();
}
```






A treia variantă va fi cea folosită în aplicații și utilizează argumente de tip referință pentru o listă definită printr-un pointer:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef int T;                                // orice tip numeric
typedef struct nod {
    T val;                                    // câmp de date
    struct nod *leg;                          // câmp de legătură
} Nod;
```




```
void initS ( Nod* & s) {  
    s = NULL;  
}
```


```
//testează dacă lista e vidă  
int emptyS(Nod* & s) {  
    return (s == NULL);  
}
```




```
// pune în stivă un element  
void push (Nod* & s, int x) {  
    Nod * nou = (Nod*)malloc(sizeof(Nod));  
    nou->val = x;  
    nou->leg = s;  
    s = nou;  
}
```



```
// scoate din stivă un element
int pop (Nod* & s) {
    Nod * p;
    int rez;
    rez = s->val;    // valoare din primul nod
    p = s->leg;      // adresa nod următor
    free (s) ;
    s = p;           // adresa vârf stivă
    return rez;
}
```



```
// utilizzare
int main () {
    Nod* st;
    int x;
    initS(st);
    for (x = 0; x < 11; x++)
        push(st,x);
    while (! emptyS(st))
        printf ( "%d\n", pop(st));
    getch();
}
```



Structura de listă înlănțuită poate fi definită ca o structură recursivă: o listă este formată dintr-un element, urmat de o altă listă, eventual vidă. Acest punct de vedere poate conduce la funcții recursive pentru operații cu liste, dar fără niciun avantaj față de funcțiile iterative.



Exemplu de afișare recursivă a unei liste:

```
void printL ( Nod* lst) {  
    if (lst != NULL) {          // daca (sub)lista nu e vidă  
        printf ("%d\n",lst->val);    // afișarea primului element  
        printL (lst->leg);  
        // afișare sublistă de după primul element  
    }  
}
```

# Liste înlănțuite ordonate

- Listele înlănțuite ordonate se folosesc în aplicațiile care fac multe operații de adăugare și/sau ștergere la/din listă și care necesită menținerea permanentă a ordinii în listă
- Pentru liste, adăugarea cu păstrarea ordinii este mai eficientă decât pentru vectori, dar reordonarea unei liste înlănțuite este o operație ineficientă



# Liste înlănțuite ordonate

- În comparație cu adăugarea la un vector ordonat, adăugarea la o listă ordonată este mai rapidă și mai simplă, deoarece nu necesită mutarea unor elemente în memorie
- Pe de altă parte, căutarea unei valori într-o listă înlănțuită ordonată nu poate fi la fel de eficientă cum este căutarea într-un vector ordonat (căutarea binară nu se poate aplica la liste)

# Liste înlănțuite ordonate

- Crearea și afișarea unei liste înlănțuite ordonate poate fi considerată și ca o metodă de ordonare a unei colecții de date
- Operația de adăugare a unei valori la o listă ordonată este precedată de o căutare a locului unde se face inserția, adică de găsirea nodului de care se va lega noul element



# Liste înlănțuite ordonate

- Mai exact, se caută primul nod cu valoare mai mare decât valoarea care se adaugă
- Căutarea folosește o funcție de comparare care depinde de tipul datelor memorate și de criteriul de ordonare al elementelor



# Liste înlănțuite ordonate

- După căutare pot exista 3 situații:
  - Noul element se introduce înaintea primului element din listă
  - Noul element se adaugă după ultimul element din listă
  - Noul element se intercalează între două noduri existente
- Prima situație necesită modificarea capului de listă și de aceea este tratată separat

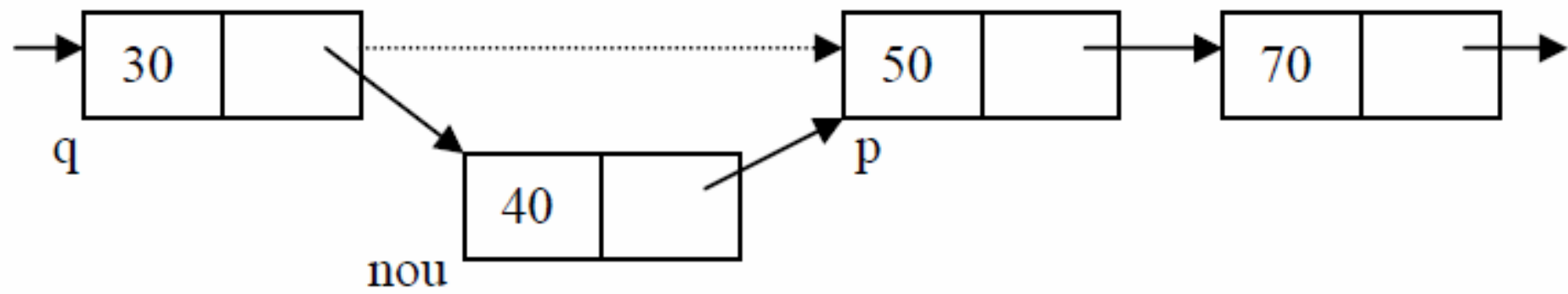


## *Exemplu*

Pentru inserarea valorii 40 într-o listă cu nodurile 30, 50, 70, se caută prima valoare mai mare ca 40 și se inserează 40 înaintea nodului 50

Operația presupune modificarea adresei de legătură a nodului precedent (cu valoarea 30), deci trebuie să dispunem și de adresa lui

Se folosește o variabilă pointer  $q$ , pentru a reține mereu adresa nodului anterior nodului  $p$ , unde  $p$  este nodul a cărui valoare se compară cu valoarea de adăugat (avem mereu  $q \rightarrow \text{leg} == p$ )






Adăugarea unui nod la o listă ordonată  
necesită:

- 1) crearea unui nod nou - alocare de memorie  
și completare câmp de date
- 2) căutarea poziției din listă unde trebuie legat  
noul nod
- 3) legarea efectivă prin modificarea a doi  
pointeri - adresa de legătură a nodului  
precedent  $q$  și legătura noului nod (cu excepția  
adăugării înaintea primului nod):


$q \rightarrow \text{leg} = \text{nou};$

$\text{nou} \rightarrow \text{leg} = p;$




```
// inserție în listă ordonată, cu doi pointeri
void insL (List & lst, T x) {
    Nod *p,*q, *nou ;
    nou = (Nod*)malloc(sizeof(Nod));
        // creare nod nou pentru inserarea valorii x
    nou->val = x;
        // completare cu date nod nou
    if ( lst == NULL || x < lst->val) {
        // daca lista vidă sau x mai mic ca primul element
        nou->leg = lst;
        lst = nou;           // adaugă nou la început de listă
    }
}
```







```
else {           // altfel caută locul unde trebuie inserat x
p = q = lst;      // q este nodul precedent lui p
while ( p != NULL && p->val < x) {
    q = p;
    p=p->leg;      // avans cu pointerii q și p
}
nou->leg = p;
q->leg = nou;      // nou se introduce între q și p
}
}
```




Funcția următoare folosește un singur pointer  $q$ :  
căutarea se oprește pe nodul  $q$ , precedent celui  
cu valoare mai mare ca  $x$  (*nou* se leagă între  $q$  și  
 $q \rightarrow \text{leg}$ )



```
void insL (List & lst, T x) {  
    Nod* q, *nou ;  
    nou = (Nod*)malloc(sizeof(Nod)); // creare nod nou  
    nou->val = x;  
    if ( lst==NULL || x < lst->val) {  
        // dacă listă vidă sau x mai mic ca primul element  
        nou->leg = lst;  
        lst= nou;           // adăugare la început de listă  
        return;  
    }  
}
```



```
q = lst;  
// ca să nu se modifice începutul listei /st  
while ( q->leg != NULL && x > q->leg->val)  
// până când  $x < q->leg->val$   
    q = q->leg;  
nou->leg = q->leg;  
q->leg = nou;    // nou între q și q->leg  
}
```



Ștergerea unui element cu valoare dată dintr-o listă începe cu căutarea elementului în listă, urmată de modificarea adresei de legătură a nodului precedent celui șters

Fie  $p$  adresa nodului ce trebuie eliminat și  $q$  adresa nodului precedent

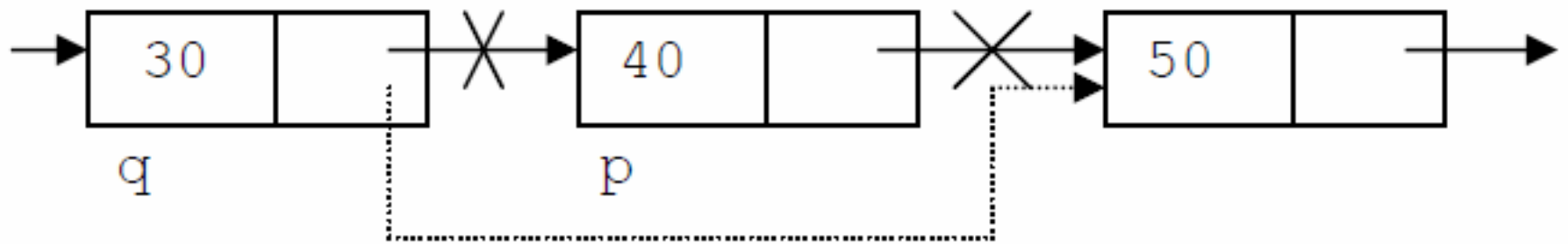
Eliminarea unui nod  $p$  (diferit de primul) se realizează prin următoarele operații:

```
 $q \rightarrow \text{leg} = p \rightarrow \text{leg};$ 
```

```
// succesorul lui p devine succesorul lui q
```

```
 $\text{free}(p);$ 
```


```
// se eliberează memoria alocată lui p
```





Dacă se șterge chiar primul nod, atunci trebuie modificată și adresa de început a listei (primită ca argument de funcția respectivă)


Funcția următoare elimină nodul cu valoarea  $x$ , folosind doi pointeri



```
void delL (List & lst, T x) {  
    // elimină elementul cu valoarea x din lista /lst  
    Nod* p = lst, *q = lst;  
    while ( p != NULL && x > p->val ) {  
        // caută pe x in lista (x de tip numeric)  
        q = p;  
        p = p->leg;  
        // q->leg == p (q înainte de p)  
    }  
}
```



```
if (p->val == x) {           // daca x este găsit
    if (q == p)
        // daca p este primul nod din lista
        lst = lst->leg;
        // modifică adresa de inceput a listei
    else // x găsit la adresa p
        q->leg = p->leg;
        // dupa q urmează acum succesorul lui p
    free(p);
// eliberare memorie ocupată de elem. eliminat
}
```



Funcția următoare de eliminare folosește un singur pointer:

```
void delL (List & lst, T x) {  
    Nod* p = lst;  
    Nod* q;                // q = adresă nod eliminat  
    if (x == lst->val) {    // dacă x este primul element  
        q = lst;  
        lst = lst->leg;  
        free(q);           // necesar pentru eliberare memorie  
        return;  
    }  
}
```

```
while ( p->leg != NULL && x > p->leg->val)
    p = p->leg;
if (p->leg ==NULL || x != p->leg->val) return;
    // x nu există în lista
q = p->leg;        // adresă nod de eliminat
p->leg = p->leg->leg;
free(q);
}
```



Inserarea și ștergerea dintr-o listă ordonată  
se pot exprima și recursiv

// inserare recursivă în listă ordonată

```
void insL (List & lst, T x) {
```

```
    Nod * aux;
```

```
    if ( lst !=NULL && x > lst->val)
```


```
        // dacă x mai mare ca primul element
```

```
        insL ( lst->leg, x);
```

```
    // se va introduce în sublista de după primul
```

```
    else {
```

```
        // lista vidă sau x mai mic decât primul element
```



```
aux = lst;  
// adresa primului element din lista veche  
lst = (Nod*)malloc(sizeof(Nod));  
lst->val = x;  
lst->leg = aux;  
// noul element devine primul element  
}  
}
```



```
// eliminare x din lista lst (recursiv)
```

```
void delL (List & lst, T x) {
```

```
    Nod* q;                // adresă nod de eliminat
```

```
    if (lst != NULL)        // dacă lista nu e vidă
```

```
        if (lst->val != x)
```

```
            // dacă x nu este primul element
```

```
            delL (lst->leg, x);
```

```
            // elimină x din sublista care urmează
```

```
    else {                  // dacă x este primul element
```

```
        q = lst;
```

```
        lst = lst->leg;
```

```
        // modifică adresa de inceput a listei
```

```
        free(q);
```

```
    } }
```



# Variante de liste înlănțuite

- Liste cu structura diferită față de o listă simplu înlănțuită:
  - ☐ liste circulare
  - ☐ liste cu element santinelă
  - ☐ liste dublu înlănțuite



# Variante de liste înlănțuite

- Liste cu elemente comune - un același element aparține la două sau mai multe liste, având câte un pointer pentru fiecare din liste
- În felul acesta, elementele pot fi parcurse și folosite în ordinea din fiecare listă



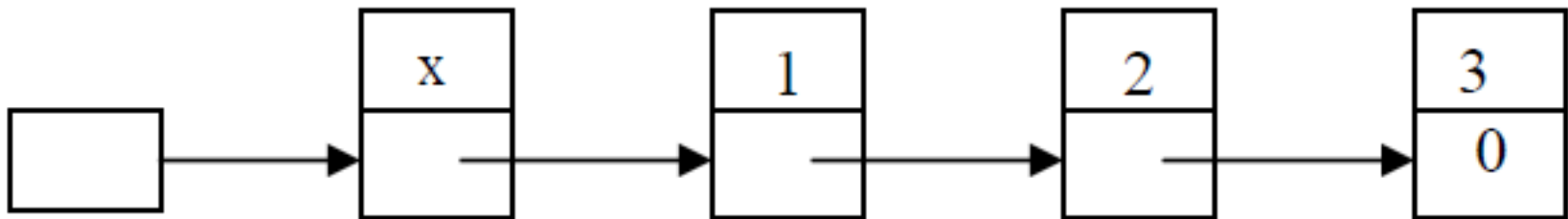



# Variante de liste înlănțuite

- Liste cu auto-organizare, în care fiecare element accesat este mutat la începutul listei (**Splay List**)
- În felul acesta, elementele folosite cel mai frecvent se vor afla la începutul listei și vor avea un timp de regăsire mai mic

# Liste cu santinelă

- O listă cu santinelă conține cel puțin un element (numit **santinelă**), creat la inițializarea listei și care rămâne la începutul listei, indiferent de operațiile efectuate





Deoarece lista nu este niciodată vidă și adresa de început nu se mai modifică la adăugarea sau la ștergerea de elemente, operațiile sunt mai simple (nu mai trebuie tratat separat cazul modificării primului element din listă)

## Exemple de funcții

// inițializare listă cu santinelă

```
void initL (List & lst) {  
    lst = (Nod*)malloc(sizeof(Nod));  
    lst->leg = NULL;           // nimic în lst->val  
}
```



```
// afișare listă cu santinelă
```

```
void printL (List & lst) {  
    lst = lst->leg;          // primul element cu date  
    while (lst != NULL) {  
        printf("%d ", lst->val);  
        // afișare element curent  
        lst = lst->leg;  
        // avans la următorul element  
    }  
}
```



// inserare în lista ordonată cu santinelă

```
void insL (List & lst, int x) {  
    Nod *p = lst, *nou ;  
    nou = (Nod*)malloc(sizeof(Nod));  
    nou->val = x;  
    while ( p->leg != NULL && x > p->leg->val )  
        p=p->leg;  
    nou->leg = p->leg;  
    p->leg = nou;  // nou după p  
}
```

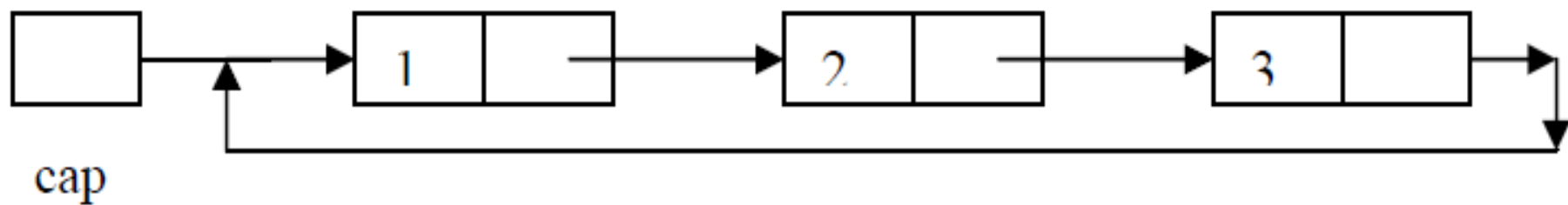
// eliminare din listă ordonată cu santinelă

```
void delL (List & lst, int x) {  
    Nod *p = lst;  
    Nod *q;  
    while ( p->leg != NULL && x > p->leg->val)  
        // caută pe x în listă  
        p = p->leg;  
    if (p->leg == NULL || x != p->leg->val) return;  
    // dacă x nu există în listă  
    q = p->leg;           // adresa nod de eliminat  
    p->leg = p->leg->leg;  
    free(q);  
}
```



# Liste circulare

- Listele circulare permit accesul la orice element din listă, pornind din poziția curentă, fără a fi necesară o parcurgere de la începutul listei
- Într-o listă circulară definită prin adresa elementului curent, nu este important care este primul sau ultimul element din listă






# Liste circulare

- Definiția unui nod de listă circulară este aceeași ca la o listă simplu înlănțuită
- Apar modificări la inițializarea listei și la condiția de terminare a listei - se compară adresa curentă cu adresa primului element, în loc de comparație cu constanta NULL
- Exemple de operații cu o listă circulară cu element santinelă



// inițializare lista circulară cu santinelă

```
void initL (List & lst) {  
    lst = (Nod*) malloc (sizeof(Nod));  
    // creare element santinelă  
    lst->leg = lst;  
    // legat la el însuși  
}
```



// adăugare la sfârșit de listă

```
void addL (List & lst, int x) {  
    Nod* p = lst;           // un cursor în listă  
    Nod* nou = (Nod*) malloc(sizeof(Nod));  
    nou->val = x;  
    nou->leg = lst;          // noul element va fi și ultimul  
    while (p->leg != lst)  
        // caută adresa p a ultimului element  
        p = p->leg;  
    p->leg = nou;  
}
```



// afișare lista

```
void printL (List & lst) {           // afișare conținut lista
    Nod* p = lst->leg;
    // primul element cu date este la adresa p
    while ( p != lst) {
        // repetă până când p ajunge la santinelă
        printf ("%d ", p->val);
        // afișare obiect din poziția curentă
        p = p->leg;
        // avans la următorul element
    }
}
```