



# LISTE

Șl. Dr. Ing. Șerban Radu

Departamentul de Calculatoare

Facultatea de Automatică și Calculatoare



# Cuprins

- Liste simplu înlănțuite
- Operații cu liste simplu înlănțuite
- Variante de liste simplu înlănțuite

# Liste înlănțuite

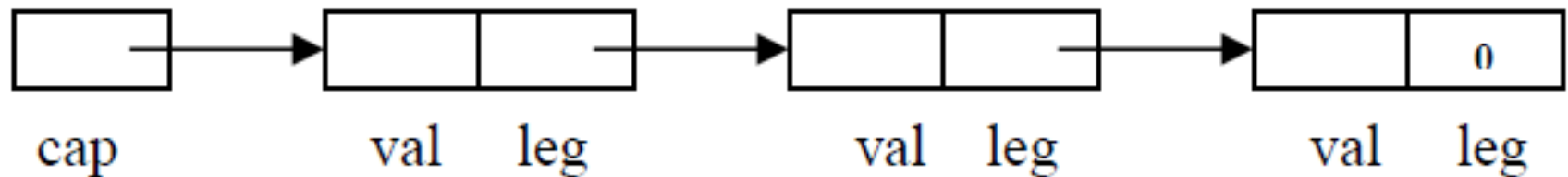
- O **listă înlănțuită (Linked List)** este o colecție de elemente, alocate dinamic, dispersate în memorie, dar legate între ele prin pointeri, ca într-un lanț
- O listă înlănțuită este o structură dinamică, flexibilă, care se poate extinde continuu, fără ca utilizatorul să fie preocupat de posibilitatea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

# Liste simplu înlănțuite

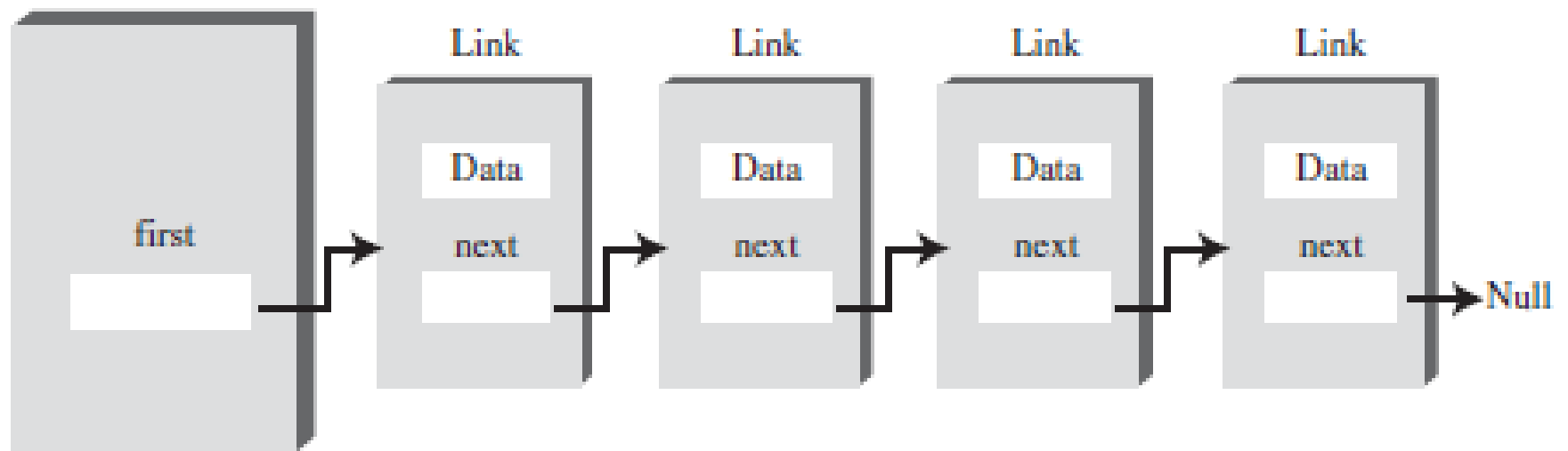
- Într-o **listă simplu înlănțuită** fiecare element al listei conține **adresa elementului următor din listă**
- Ultimul element poate conține ca adresă de legătură fie constanta NULL (un pointer către nicăieri), fie adresa primului element din listă (dacă este o listă circulară), fie adresa unui element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

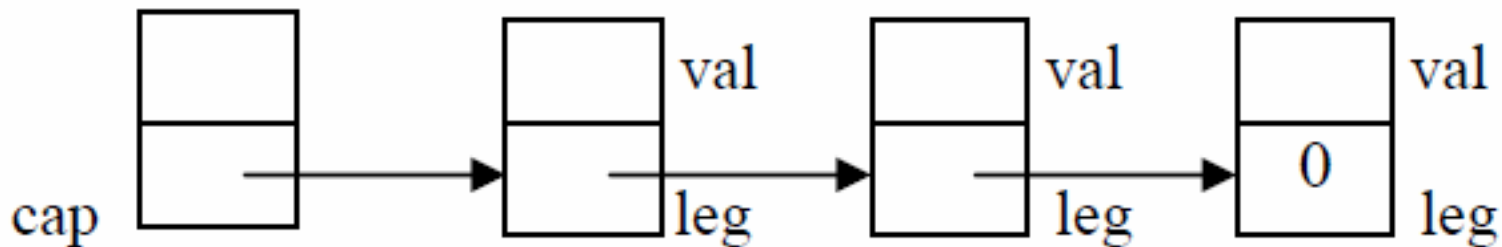


## Linked List



# 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 (traversarea listei)



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

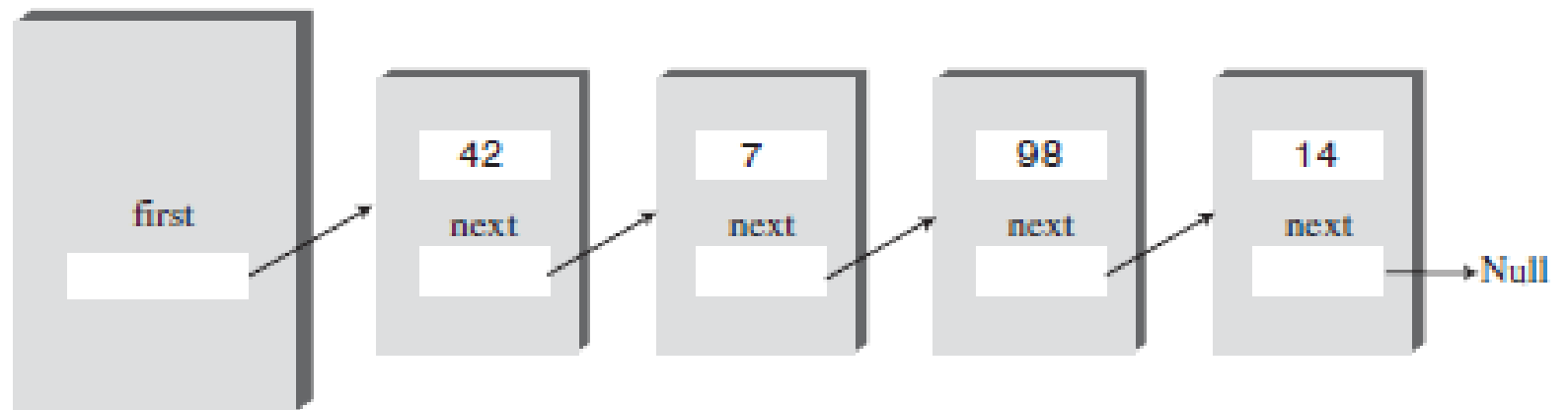
Crearea unui nou element din listă necesită alocarea de memorie, prin funcția **malloc**

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

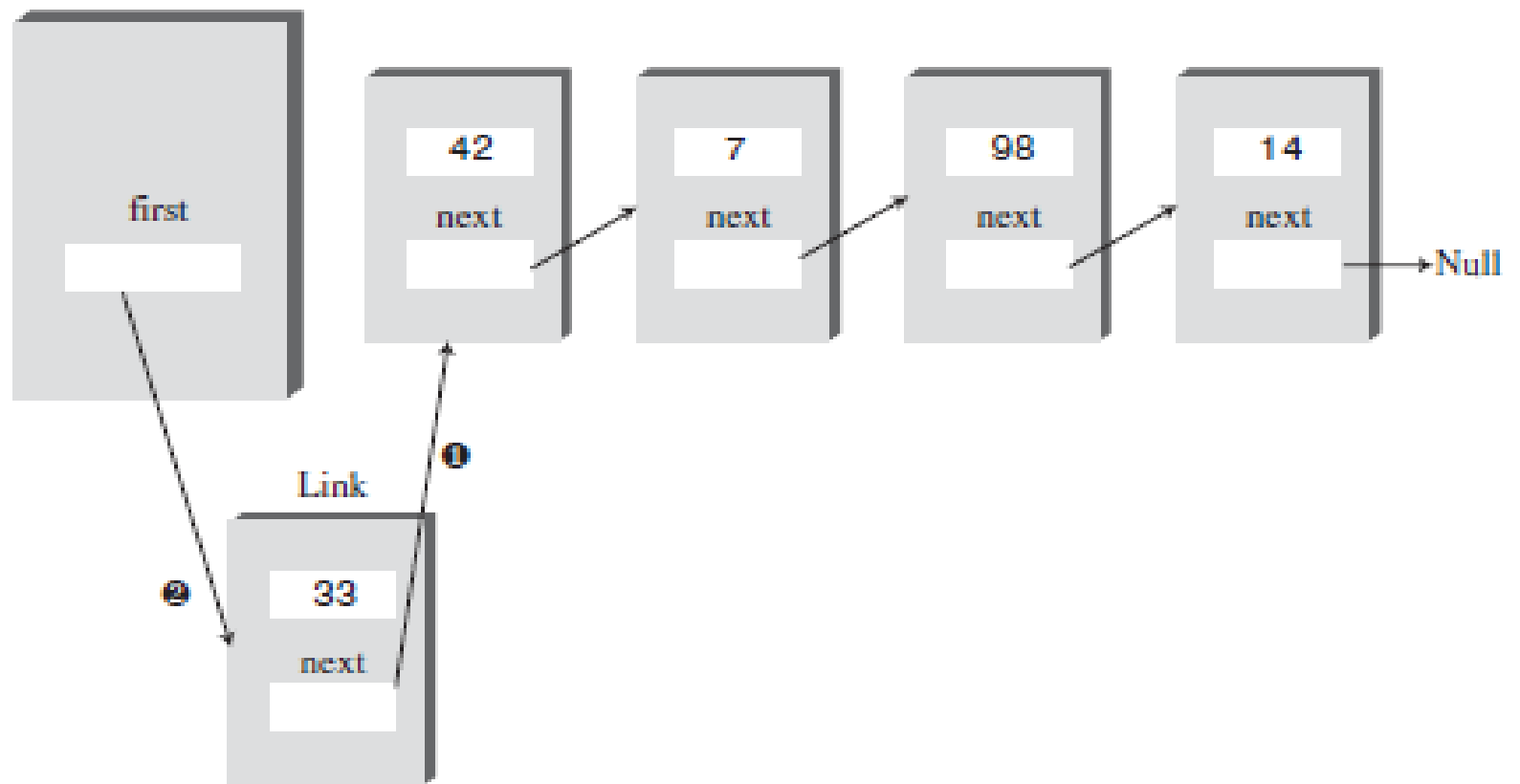
```
nou = (Nod*) malloc( sizeof(Nod));
```

# Adăugarea unui element la o listă înlănțuită

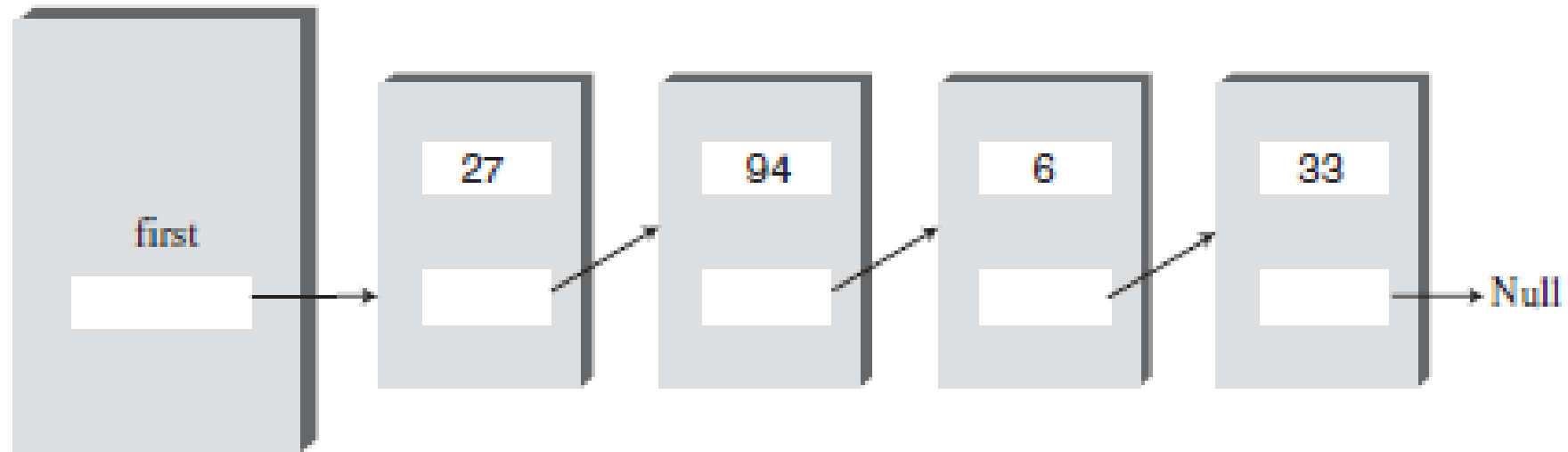
- Mereu la începutul listei
- Mereu la sfârșitul listei
- Într-o poziție determinată de valoarea noului element
- Dacă ordinea datelor din listă este indiferentă pentru aplicație, atunci cel mai simplu este ca adăugarea să se facă numai la începutul listei
- Afișarea valorilor din listă se face în ordine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



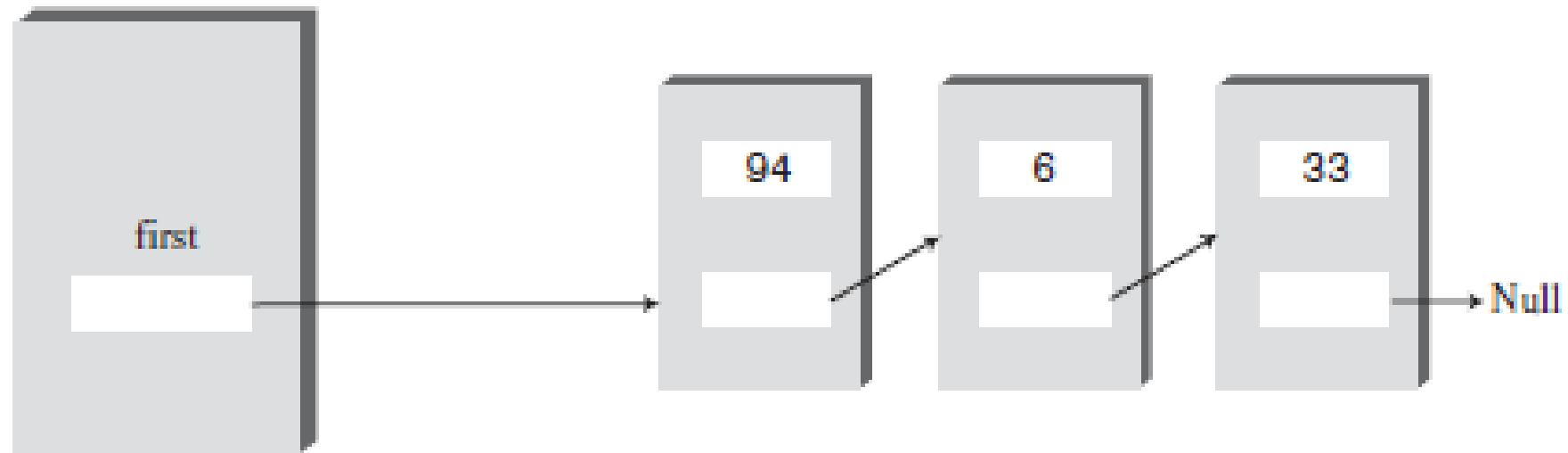
a) Before Insertion



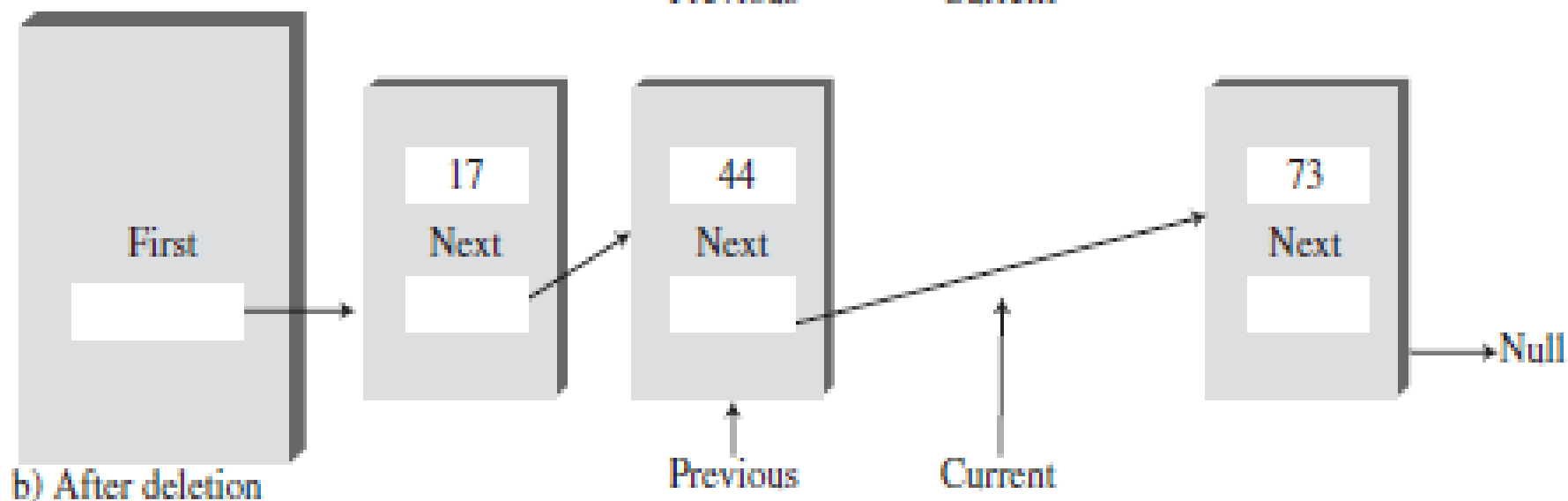
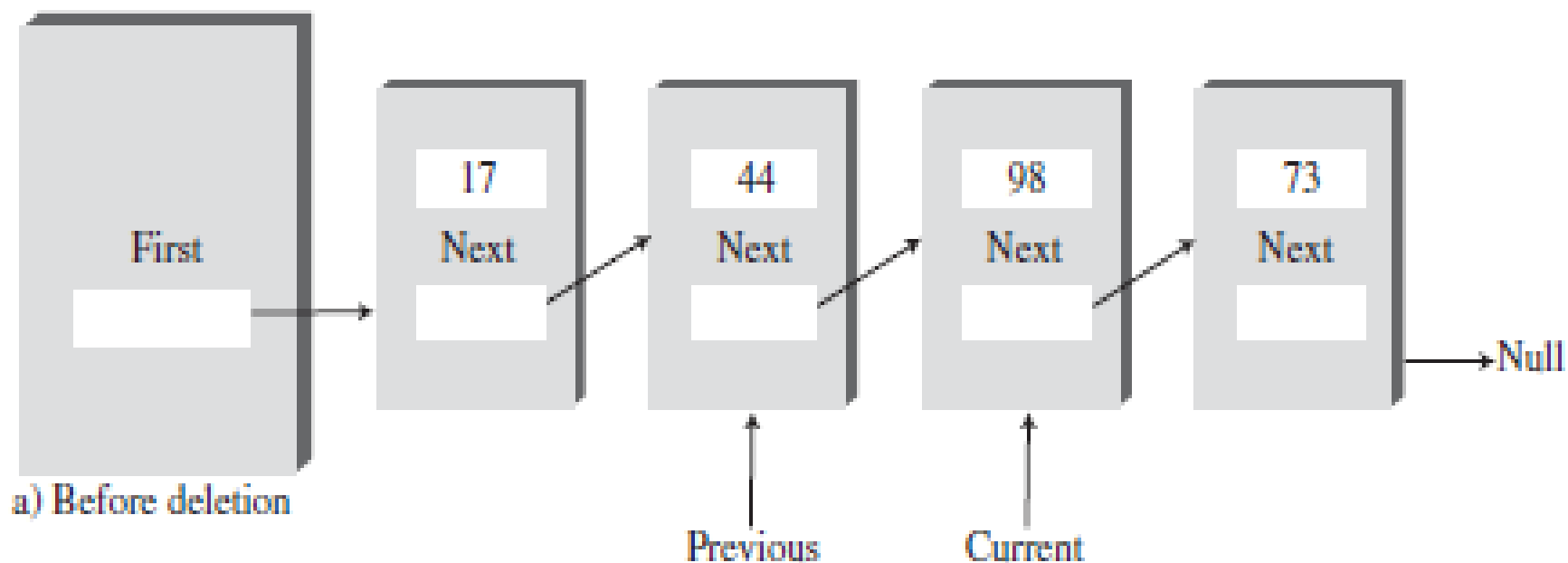
b) After Insertion



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)); // se alocă memorie  
    nou->val = x;  
    nou->leg = lst;  
    lst = nou;           // noul element este primul  
    printf("Introduceti valoarea elementului din lista = ");  
}  
while (lst != NULL) {           // se afișează lista  
    printf("%d\n", lst->val);  
    // în ordine inversă celei de adăugare  
    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 listă 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 listă 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 dacă 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 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;    // valoarea din primul nod
    p = s->leg;      // adresa nodului următor
    free (s) ;
    s = p;           // adresa vârfului stivei
    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  
    }  
}
```





# 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

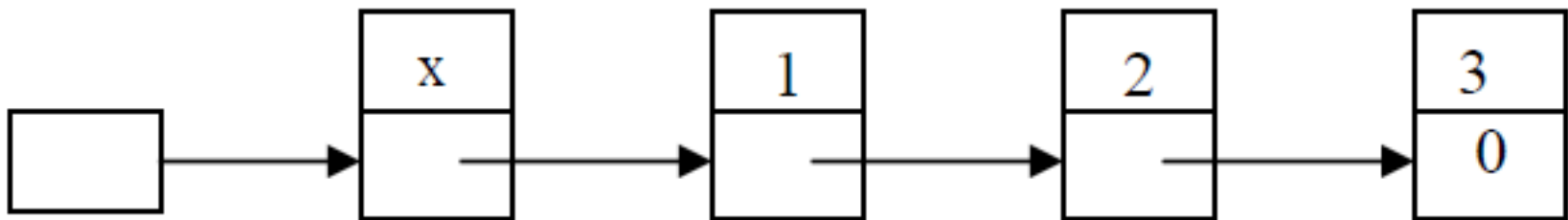



# Variante de liste înlănțuite

- 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  
    }  
}
```