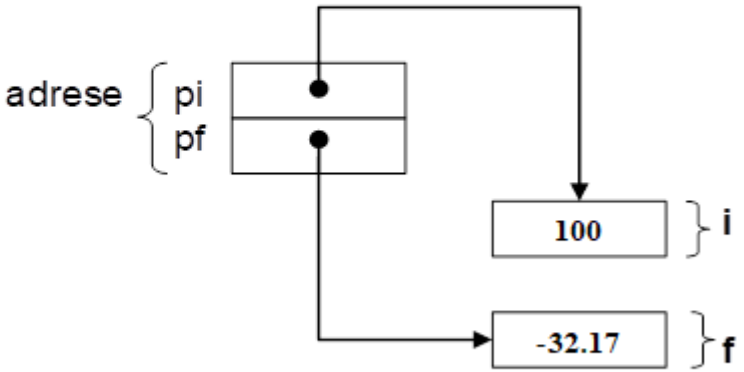
8.6 Relaţia „Pointeri – structuri”

- Structuri care conţin pointeri
- Liste înlănţuite

8.6.1 Structuri care conţin pointeri

```
void main (void )
{ struct ex
    { int * pi;
      float * pf;
    };                    /*definire tip struct ex*/
  struct ex adrese; /*declarare variabilă de tip struct ex*/
  int i = 100;
  float f;
  adrese.pi = &i;
  adrese.pf = &f;
  *adrese.pf = -32.17;
   printf("i = %d,   Tot i = %d", i, *adrese.pi);
   printf("f = %f,    Tot f = %f", f, *adrese.pf);
}
```

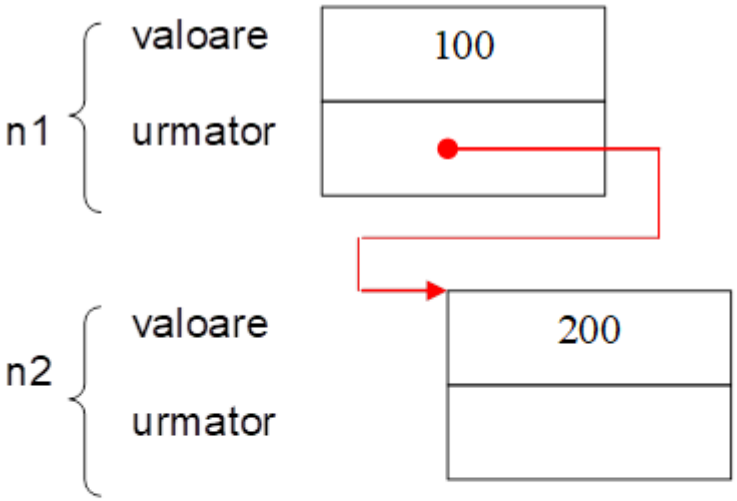


Pe ecran:

i = 100, Tot i = 100
f = -32.17, Tot f = -32.17

8.6.2 Liste înlănţuite

```
struct element
    {  int valoare;
       struct element * urmator;
    };                          /*definire tip structură*/

struct element n1, n2;          /*2 variabile de tip struct element*/
n1.urmator = &n2;               /*creăm o „legătură” între n1 şi n2*/
```

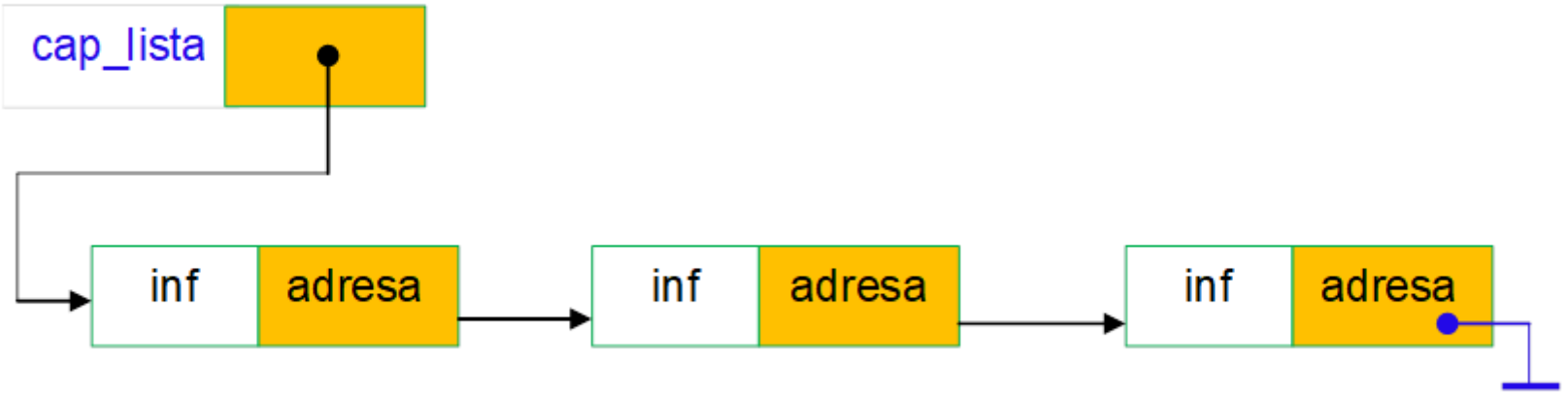


Obs: accesarea câmpului valoare al structurii n2 se poate realiza acum în trei moduri:

n2.valoare sau **n1.urmator->valoare** sau **(*n1.urmator).valoare**

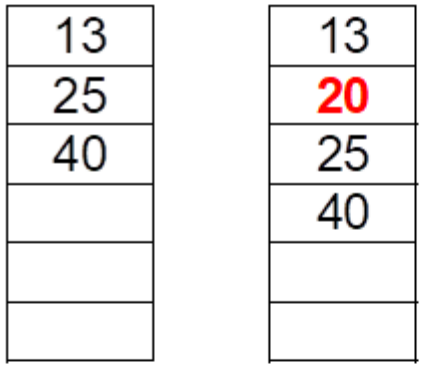
Listă înlănţuită:

- cea mai simplă formă „dinamică” de organizare a informaţiei
- colecţie de elemente de tip structură „legate” între ele prin pointeri (adresa elementului următor)
- Pentru fiecare element se alocă dinamic memoria necesară în momentul adăugării în listă şi se eliberează dinamic memoria în momentul „ştergerii” din listă
- Operaţiile de „inserare/ştergere” din listă se realizează foarte uşor
- Ordinea în parcurgerea logică a listei este dată de adresa din câmpul „adresa”
- Pentru ca lista să poată fi parcursă, inspectată, modificată, este necesar să se utilizeze un pointer la primul element al listei (*cap_lista*).
- Pentru marcarea sfârşitului listei se memorează **NULL** în câmpul adresă al ultimului element al listei.

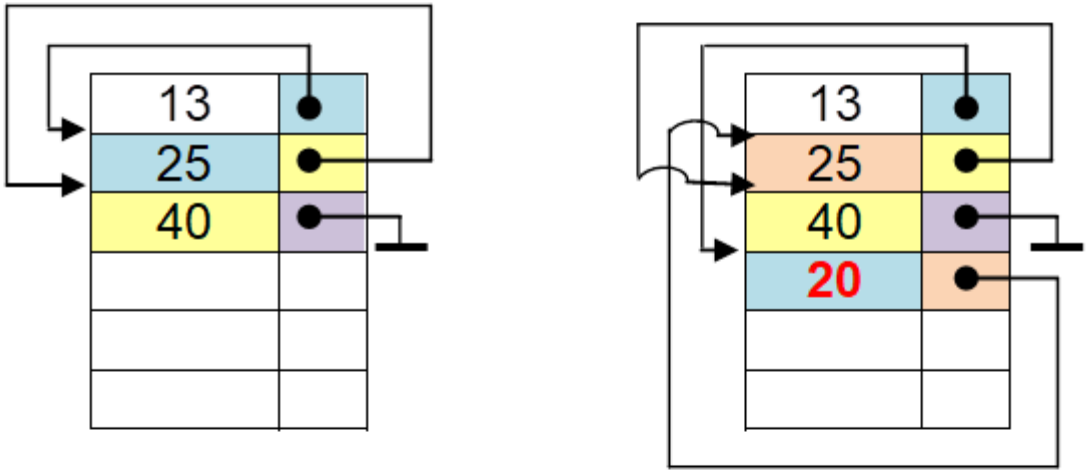


Comparaţie cu tipul tablou, pentru cazul inserării unui element in cadrul unui tablou ordonat crescător

Tablou



Listă simplu înlănţuită



Exemplu de utilizare practică a listelor înlănţuite:

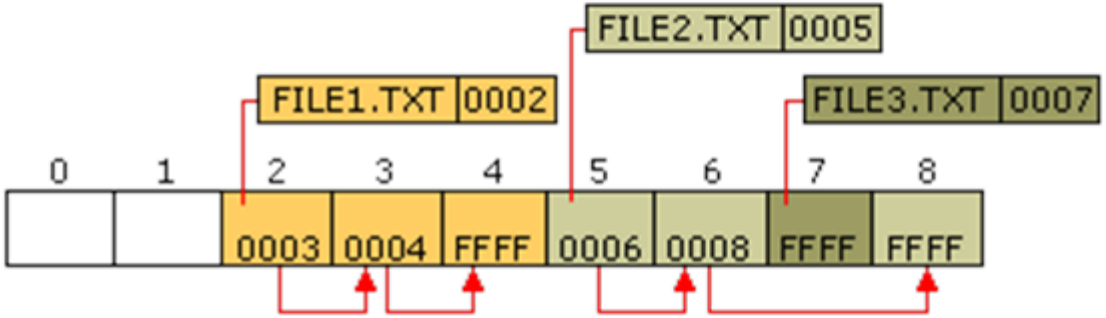
FAT - File Allocation Table

http://en.wikipedia.org/wiki/File_Allocation_Table

http://www.pctechguide.com/31HardDisk_File_systems.htm

The purpose of the File Allocation Table is to provide the mapping between **clusters** - the basic unit of **logical storage** on a disk at the operating system level - and the **physical location** of data in terms of **cylinders**, **tracks** and **sectors** - the form of addressing used by the drive's hardware controller.

The FAT contains **an entry** for every file stored on the volume **that contains the address of the file's starting cluster**.**Each cluster** contains **a pointer** to the **next cluster** in the file, **or an end-of-file indicator** at (0xFFFF), which indicates that this cluster is the end of the file.



The diagram shows three files: File1.txt uses three clusters, File2.txt is a fragmented file that requires three clusters and File3.txt fits in one cluster. In each case, the file allocation table entry points to the first cluster of the file.

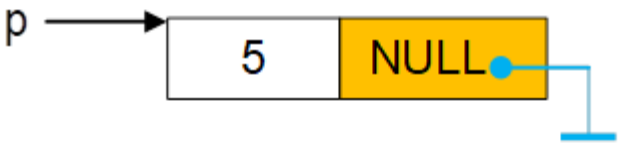
http://www.pctechguide.com/31HardDisk_File_systems.htm

Exemplu: program C simplu pentru creare, parcurgere și desființare a unei liste înlănţuite

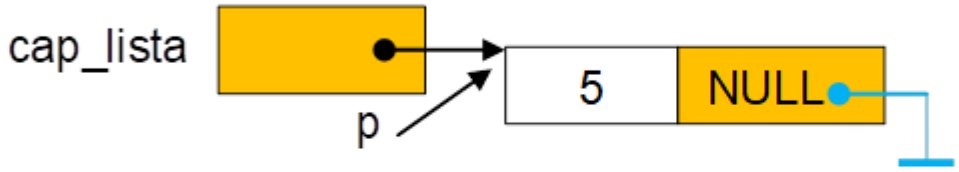
```
#include <stdio.h>
#include <stdlib.h>

struct element
  {int val;
   struct element * urm;
  };

typedef struct element ELEM;

void main (void)
  {   ELEM * cap_lista, *p, *q;
      int i, nr;            /*nr = numărul de elemente ce vor fi înscrise in listă*/
      printf("Câte elemente veţi înscrie in listă? ");
      scanf("%d", &nr);
```

```
/*se crează dinamic spaţiu de memorie pentru primul element al listei*/
p=(ELEM *)malloc(sizeof(ELEM));
```


```
 /*se testează dacă alocarea a reuşit sau nu*/
if (p==NULL)
  { printf("Alocare dinamică eşuată!");
    exit(1);          /*se încheie execuţia programului*/
  }
printf("Tastaţi valoarea primului element:");
scanf("%d", &p->val);
p->urm = NULL;        /*deocamdată este primul şi ultimul element al listei*/
```



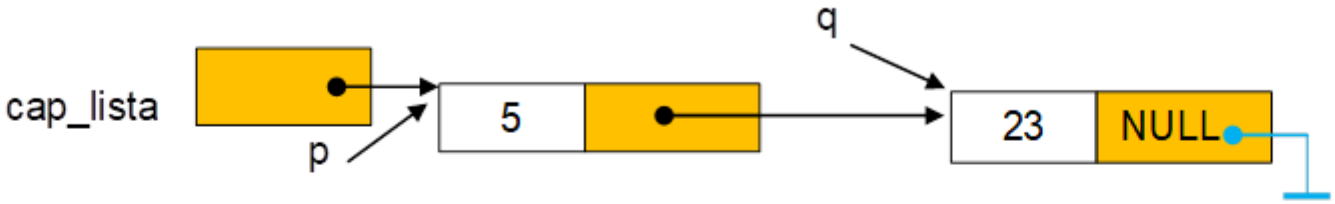

```
cap_lista = p;        /*memorează adresa primului element din listă*/
```



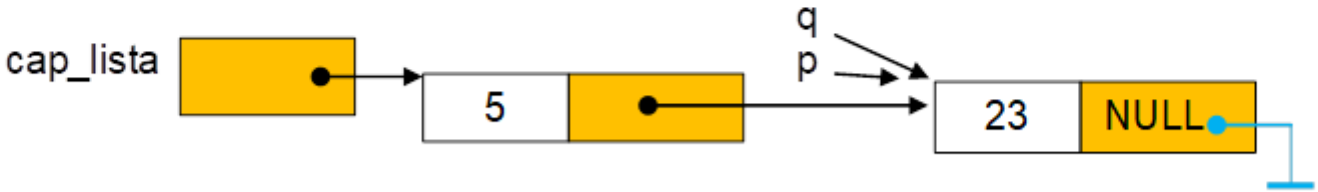

```
/*urmează crearea celorlalte nr-1 elemente ale listei*/
```

```
for (i=2; i<=nr; i++)
 { /*se crează dinamic spaţiu de memorie pentru noul element al listei*/
   q=(ELEM *)malloc(sizeof(ELEM));
  /*se testează dacă alocarea a reuşit sau nu*/
   if (q==NULL)
    { printf("Alocare dinamică eşuată!");
      exit(1);                    /*se încheie execuţia programului*/
    }
   printf("Tastaţi valoarea celui de-al %d element: ", i);
   scanf("%d", &q->val);
   q->urm = NULL;    /*noul element devine acum ultimul element al listei*/
```


```
   p->urm = q;   /*stabileşte legătura dintre elementul anterior şi cel nou*/
```

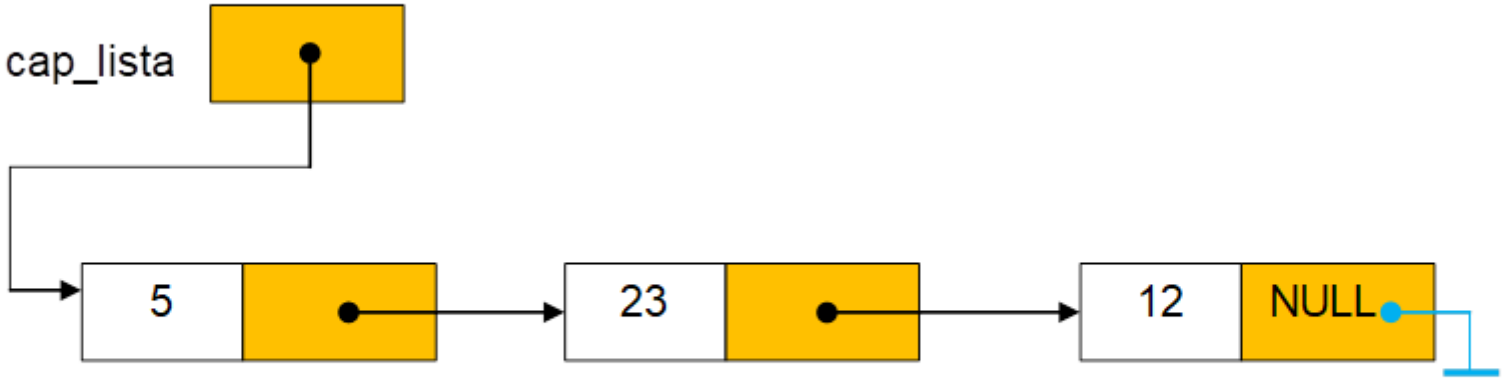



```
   p=q;          /*în p se memorează adresa noului element pentru ca variabila
                 q să fie folosită la următoarea trecere prin ciclul for pentru o
                 nouă alocare dinamică de memorie*/
```




```
}
```

```
/*parcurgerea listei, de la primul până la ultimul element, şi afişarea valorilor
  pe care le conţine*/

for (p=cap_lista, i=1; p!=NULL; p=p->urm, i++)
   printf("Valoarea elementului cu nr. %d este %d\n", i, p->val);

/*deoarece programul se încheie, se va elibera automat spaţiul de memorie alocat
anterior. Totuşi, un stil riguros de programare, impune eliberarea explicită a
spaţiului de memorie alocat dinamic, folosind funcţia free(), ca în instrucţiunea
for următoare*/

for (p=cap_lista ; p!=NULL; p=q)
     {  q=p->urm;  /*se salvează adresa următorului element*/
        free (p);     /*se eliberează spaţiul de memorie alocat elementului curent*/
     }
}
```



Operaţii cu liste înlănţuite

```
typedef struct element
    {  int val;
       struct element *urm;
    } ELEM;
```

ATENTIE!!!

Vor fi prezentate doar schite orientative (fragmente de program) care NU acopera TOATE situatiile posibile!!!

Scopul este prevenirea invatarii pe de rost si stimularea efortului de a INTELEGE notiunile inainte de a incerca sa le folositi in scrierea programelor.

Alocare dinamică de memorie pentru un element

```
ELEM* creeaza_elem( )
   { ELEM *p;
     p = (ELEM*) malloc(sizeof(ELEM));
     p->urm = NULL;
     return p;
}
```

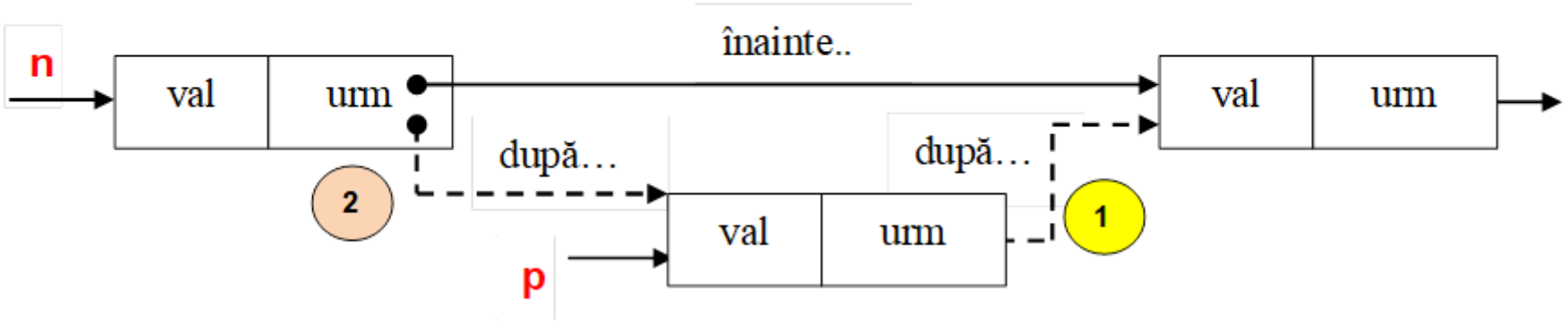
Obs. – Tema – De inserat in codul functiei secventa de testare a valorii returnate de malloc().

Adăugarea unui nou element la sfârşitul listei

```
ELEM *  adauga (ELEM * s, ELEM * primul)
  { ELEM *p;  /*variabila pointer locala*/
    if(primul == NULL)     /*creeaza primul element, daca nu exista*/
      { primul = s;
        s->urm = NULL;
        return primul;
       }
    /*cauta si gaseste ultimul element din lista*/
    for( p=primul; p->urm != NULL; p=p->urm)
                ;     /*sau: continue; */
    p->urm = s;
    s->urm = NULL;
    return primul;
   }

/*Obs: elementul care se adaugă trebuie să fie alocat dinamic
înainte de fi trimis ca parametru la funcţie*/
```

Obs: parcurgerea listei pentru găsirea ultimului element este secvenţială şi are o durată foarte mare. Se poate scrie o altă variantă în care să se folosească o variabilă suplimentară pentru memorarea adresei ultimului element al listei --- **TEMA**

Inserarea unui element între alte două elemente aflate deja în listă




```
/*insereaza elementul cu adresa p dupa elementul cu adresa n*/

void  inserez_dupa  (ELEM * n, ELEM * p)
  { p->urm = n->urm;
    n->urm = p;
    return;
  }
```

ATENTIE la cazurile speciale! Detalii la curs...

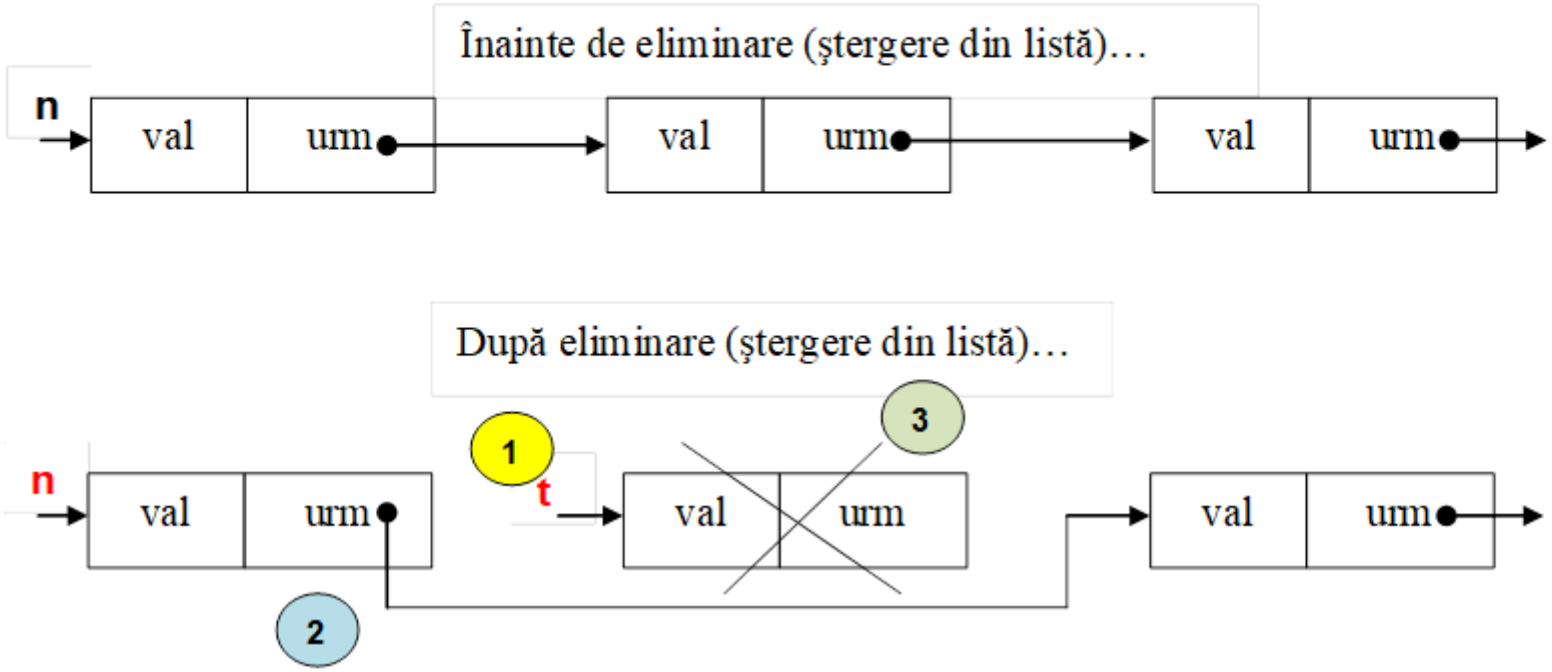
Obs:

- dacă se doreşte inserarea elementului cu adresa p înaintea celui cu adresa n, trebuie să utilizăm un alt mod de abordare a problemei, deoarece în cazul unei liste simplu înlănţuite nu se poate determina determina direct adresa elementului anterior, ci numai cea a elementului următor.
- O variantă corectă de rezolvare este interschimbarea valorilor elementelor de la adresele p şi n, urmată de inserarea elementului cu adresa p după cel cu adresa n.

```
/*insereaza elementul cu adresa p inaintea elementului cu adresa n*/

void inserez_inainte (ELEM * n, ELEM * p)
  { ELEM t;
    t = *p;
    *p = *n;
    *n = t;
    n->urm = p;
    return;
  }
```

ATENTIE la cazurile speciale! Detalii la curs...

Eliminarea („ştergerea”) unui element din listă




```
/*elimina (sterge) succesorul elementului cu adresa n*/

void elimina_succesor (ELEM * n)
  {ELEM * t;
    t = n->urm;
    n->urm = t->urm;
    free(t);
    return;
  }
```

1 2 3

ATENTIE la cazurile speciale! Detalii la curs...

Eliminarea unui anumit element, precizat prin adresa:

```
/*elimina (sterge) elementul precizat prin adresa acela*/

ELEM * sterge_elem (ELEM * acela, ELEM * primul)
  {ELEM * p;
   if(acela == primul)
        primul = acela->urm;
   else
      { for(p=primul; ((p!=NULL) && (p->urm!=acela)); p=p->urm)
                                        ;     /*instructiune vida*/
        if(p==NULL)
          { printf("elem. nu exista in lista!");
            return primul;
          }
        p->urm = p->urm->urm;
      }
   free(acela);
   return primul;
  }
```

ATENTIE la cazurile speciale! Detalii la curs...

Parcurgerea unei liste

Se realizează operaţiile dorite asupra fiecărui element al listei. De exemplu, se poate afişa pe ecran valoarea din câmpul ***val*** al fiecărei componente:

```
void afisare_val (ELEM * primul)
  { ELEM *p;
    int i=1;
    for( p=primul; p!=NULL; p=p->urm)
        printf("Elementul nr. %d din lista are valoarea %d\n", i++, p->val);
    return
  }
```

Căutarea unui element în cadrul listei

Căutarea unui element în cadrul listei se face, de obicei, în raport cu o aşa-numită „cheie" de identificare. Acest rol poate fi atribuit oricărui câmp de informaţii din cadrul elementului respectiv sau unei combinaţii de valori din câmpuri diferite.

Căutarea unui element în cadrul unei lista simplu înlănţuite este de natură **secvenţială**: elementele se parcurg în ordine, începând cu primul înscris în listă.

Căutarea se încheie în momentul găsirii elementului dorit sau dacă s-au parcurs toate elementele listei fără a găsi acel element.

```
ELEM* cauta_elem (int nr, ELEM * primul)
   { ELEM *p;
     for( p=primul; p!=NULL; p=p->urm)
        if(p->val == nr) return p;
     return NULL;
   }
```

TEMA:

Scrieţi mici programe C (sau functii C in cadrul unor programe) care să realizeze operaţiile prezentate intr-un mod cat mai corect si mai complet (sa ia in considerare toate situatiile posibile).