Curs SDA (PC2)

Curs 4
Structuri de date
(continuare)

Iulian Năstac

10. Funcţii pentru alocarea dinamică a memoriei (Recapitulare)

- Alocarea dinamică este caracteristica prin care un program poate obţine memorie în timpul rulării.

- În limbajul C, variabilelor globale li se alocă memorie în timpul compilării, iar cele locale folosesc memoria stivă. Nici variabilele locale nici cele globale nu pot fi adăugate în timpul execuţiei programului.

- Există însă cazuri în care memoria necesară unui program nu poate fi cunoscută dinainte (de exemplu un procesor de texte sau o bază de date care pot necesita spaţii de memorie diferite pentru rulări diferite ale aceluiaşi program). Pentru acest tip de programe se utilizează alocarea dinamică a memoriei funcţie de numărul şi dimensiunea variabilelor noi create pe parcursul execuţiei.

(Recapitulare)

- Zona de memorie folosită pentru alocarea dinamică este obţinută din *heap* (care este o zonă de memorie liberă aflată între zona de memorie permanentă a programului care se desfăşoară şi cea stivă).
- Deşi mărimea zonei heap este necunoscută, ea conţine, în general, o cantitate destul de mare de memorie liberă.
- Principalele funcţii pentru alocarea dinamică în C sunt *malloc* şi *free* aflate în fişierele header *stdlib.h* şi *alloc.h (sau malloc.h - depinde de compilator)*.

11. Utilizarea incorectă a pointerilor
(Recapitulare)

- De multe ori erorile datorate pointerilor sunt foarte dificil de depistat.
- La folosirea greşită a unui pointer se citeşte sau se scrie într-o zonă necunoscută de memorie.
- Inconvenientul major este că efectele acestor greşeli sunt vizibile doar la execuţia programului, nu şi la compilare.
- Găsirea greşelilor este de multe ori o sarcină anevoioasă pentru programatori, mai ales că la unele rulări ale programelor, erorile nu sunt direct vizibile.

Cap. Structuri

(Recapitulare)

1. Definirea conceptului de structură

- Limbajul de programare C poate prelucra atât date şi variabile singulare, cât şi variabile grupate, care permit o prelucrare globală.
- Un exemplu elocvent, din cea de-a doua categorie, îl reprezintă matricile care sunt mulţimi ordonate de date de acelaşi tip, relaţia de ordine dintre elemente fiind dată cu ajutorul indicilor. Numărul indicilor indică dimensiunea unei matrice (tablou). Tipul comun al elementelor tabloului este şi tipul tabloului.
- De multe ori însă, este util să grupăm datele în alt mod decât cel utilizat pentru matrici. Este cazul datelor care nu sunt neapărat de acelaşi tip şi care necesită o prelucrare globală. Această formă de grupare poartă denumirea de ***structură***.

Ca o definiţie foarte generală, se poate spune că datele grupate conform unei ierarhii se numesc ***structuri***.

Observaţii:

- Datele elementare ale unei structuri pot fi izolate (singulare) sau tablouri;
- Fiecare structură reprezintă un tip nou de date, definit de utilizator.

2. Declaraţia de structură

Formatul general pentru declararea unei structuri este:

struct identificare
{
declaratii de variabile;
} identificare_1,identificare_2,...,identificare_n;

unde *identificare, identificare_1, identificare_2, ..., identificare_n* sunt nume care pot lipsi dar nu toate deodată.

Astfel:

- dacă *identificare_1, identificare_2, ... , identificare_n* sunt absenţi, atunci *identificare* trebuie să fie prezent.
- dacă *identificare* este absent, cel puţin *identificare_1* trebuie să fie prezent.

Este de precizat că:

- *identificare* defineşte un tip nou introdus prin declaraţia de structură dată.
- *identificare_1, identificare_2,... ,identificare_n* sunt structuri de tip *identificare.*

Observaţii:

- O structură de tip *identificare* poate fi declarată şi ulterior

  struct identificare numele_noii_structuri;

- O declaraţie oarecare de structură *identificare_t* (unde *t = 1…n*) poate fi înlocuită de un tablou *k*-dimensional de elemente de tip *identificare*:

  identificare_t[lim1][lim2]...[limk]

Exemple:

1) Următoarele trei exemple de cod au acelaşi rezultat:

```
struct data_calendaristica
{int zi;
char luna[11];
int an;
} data_nasterii,data_angajarii;
```

sau

```
struct
{int zi;
char luna[11];
int an;
} data_nasterii,data_angajarii;
```

sau

```
struct data_calendaristica
{int zi;
char luna[11];
int an;
};
...
  struct data_calendaristica data_nasterii,data_angajarii;
```

2) Structură care conţine datele personale:

```
struct date_personale
  { char nume_prenume[100];
    char adresa[1000];
    struct data_calendaristica data_nasterii,data_angajarii;
    char sex;
  };
    ..............
 struct date_personale director, angajati[1000];
```

Variabila *director* este o structură de tip *date_personale* iar *angajati[1000]* este un tablou de structuri.

3) Definirea unor numere complexe A, B şi C.

```
struct COMPLEX
        {double real;
         double imag;
        }A, B, C;
```

4) Poziţia unui punct pe ecran este dată de două coordonate:

```
struct punct
   { int x;
     int y;
   };
   ...
struct punct poz;
```

3. Accesul la elementele unei structuri

Accesul la elementele unei structuri se poate face sub una din cele două forme:

- *nume.nume_dată*
- *pointer -> nume_dată*

unde: *nume* este numele structurii,
nume_dată este numele componentei,
iar *pointer* este un pointer spre structură.

Exemple:

1)
```
struct data_calendaristica
        { int zi;
        char luna[10];
        int an;
        } dc,d[10];
    ...
 dc.zi=1;
dc.an=1995;
strcpy(dc.luna,”septembrie”);
...
d[3].zi=dc.zi;
d[3].an=dc.an;
strcpy(d[3].luna,dc.luna);
...
```

2) Funcţie ce calculează şi returnează modulul numărului complex z.

```
double modul(COMPLEX *z)
        {
        return sqrt(z->x * z->x + z->y * z->y);
        }
```

Trebuie precizat că structurile şi componentele acestora pot fi *iniţializate*.

4. Declaraţii de tip

Prin declararea unei structuri se introduce un tip nou de dată.

În general, se poate atribui nume unui tip, indiferent dacă el este un tip predefinit sau unul utilizator, utilizând o construcţie de forma:

typedef tip nume_tip;

unde

- *tip* este un tip predefinit sau un tip utilizator;
- *nume_tip* este numele care se atribuie tipului definit de *tip*.

Exemple:

1)Prin declaraţia

typedef double REAL;

datele

REAL x,y;

sunt de tip double.

2) Declararea tipului COMPLEX.

```
typedef struct
        { double real;
          double imag;
        } COMPLEX;
...
```

Putem declara numere complexe prin declaraţii de forma:

COMPLEX z,tz[10];

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<conio.h>
typedef struct {
                double x;
                double y;
               } COMPLEX;
void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c);

int main()
{
 COMPLEX a,b,c;
 printf("\n\nIntroduceti partea reala si partea imaginara ");
 printf("\n ale numarului complex a :\n");
 if(scanf("%lf %lf",&a.x,&a.y)!=2)
        {
          printf("\nEroare");
          exit(1);
         }
 ….
```

```
…
printf("a = %g + i*(%g)\n",a.x,a.y);
 printf("\nIntroduceti partea reala si partea imaginara ");
 printf("\n ale numarului complex b :\n");
 if(scanf("%lf %lf",&b.x,&b.y)!=2)
        {
          printf("\nEroare");
          exit(1);
         }
 printf("b = %g + i*(%g)\n",b.x,b.y);
 sum_c(&a,&b,&c);
 printf("\na+b = %g + i*(%g)",c.x, c.y);

 getch();
}


void sum_c(COMPLEX *a, COMPLEX *b, COMPLEX *c)
{
  c->x = a->x + b->x;
  c->y = a->y + b->y;
}
```

5. Reuniuni

În C unei date i se alocă o zonă de memorie potrivit tipului datei respective şi în zona alocată acesteia se pot păstra numai date de tipul menționat.

Exemplu:

double x;

Pentru x se alocă 8 octeţi (64 de biţi) şi în zona respectivă se păstrează numere reale reprezentate prin complement faţă de 2 (a se vedea capitolul aferent sistemelor de operare).

Observaţie:

Reprezentarea cu semn (în binar) se face standard în trei moduri:

- *în mărime şi semn (MS)*;
- *în complement faţă de 1 (C1)*;
- *în complement faţă de 2 (C2)*.

Exemplu:

	MS	C1	C2
+5	0101	0101	0101
-5	1101	1010	1011

Se pot ivi situaţii când în aceeaşi zonă de memorie am dori să păstrăm date de tipuri diferite. De exemplu dacă după un anumit timp nu mai este nevoie de variabila ***x***, zona ar putea fi utilizată pentru a păstra date de alt tip (*char, int, float,* etc.). Aceste reutilizări ale zonelor de memorie duc automat la economisirea de memorie.

Pentru a realiza aceast deziderat se grupează împreună datele ce se doresc a fi alocate în aceeaşi zonă de memorie şi se foloseşte o construcţie similară ca cea de la structuri pe care o vom denumi ***reuniune*** (şi foloseşte cuvântul cheie ***union*** la declarare).

Exemple:

1)
```
union a
        {int x;         /* 2 octeţi pentru x */
        long y;         /* 4 octeţi pentru y */
        double r;       /*8 octeţi pentru r*/
        char c;         /* 1 octet pentru c */
        } var;
```

În declaraţia de mai sus ***var*** este o reuniune de tipul **a**. Accesarea variabilelor se poate face: *var.x;* sau *var.y;* sau *var.r;* sau *var.c*; dar în locaţii diferite ale programului

Pentru ***var*** se alocă zonă de memorie suficientă pentru a păstra numărul maxim de octeţi (8 octeţi în acest exemplu). Dacă s-ar fi înlocuit *union* cu *struct* ar fi fost necesari 15 octeţi.

2)

```
struct data
        { int timp;
          union { int i;
                  float f;
                  double d;
                } zc;
        } util;
```

Putem accesa:

```
util.zc.i=123;
```

Ca observaţie, spre deosebire de structuri reuniunile nu pot fi iniţializate.

6. Câmpuri

- În limbajul C se pot defini şi prelucra date pe biţi. Această utilizare permite economisirea de memorie. De exemplu, presupunem că avem nevoie de o dată care ia numai valorile 0 sau 1. O astfel de dată s-ar putea păstra pe un singur bit, şi nu pe doi biţi cum ar fi o variabilă de tip întreg.

- **Prin definiţie, un şir de biţi formează un *câmp*.**

- Un câmp se păstrează într-un *cuvânt calculator* (care poate conţine mai multe câmpuri). Un cuvânt are 16 biţi (2 octeţi).

Practic, mai multe câmpuri se pot grupa formând o structură:

```
struct identificare
      { camp_1;
        camp_2;
       ...
        camp_n;
      } nume_1, nume_2, ..., nume_n;
```

Declaraţia de câmp este:

tip nume_camp: lungime_în_biţi;

sau:

tip: lungime_în_biţi;

unde ***tip*** poate fi: *unsigned*, *int*, sau *char*.

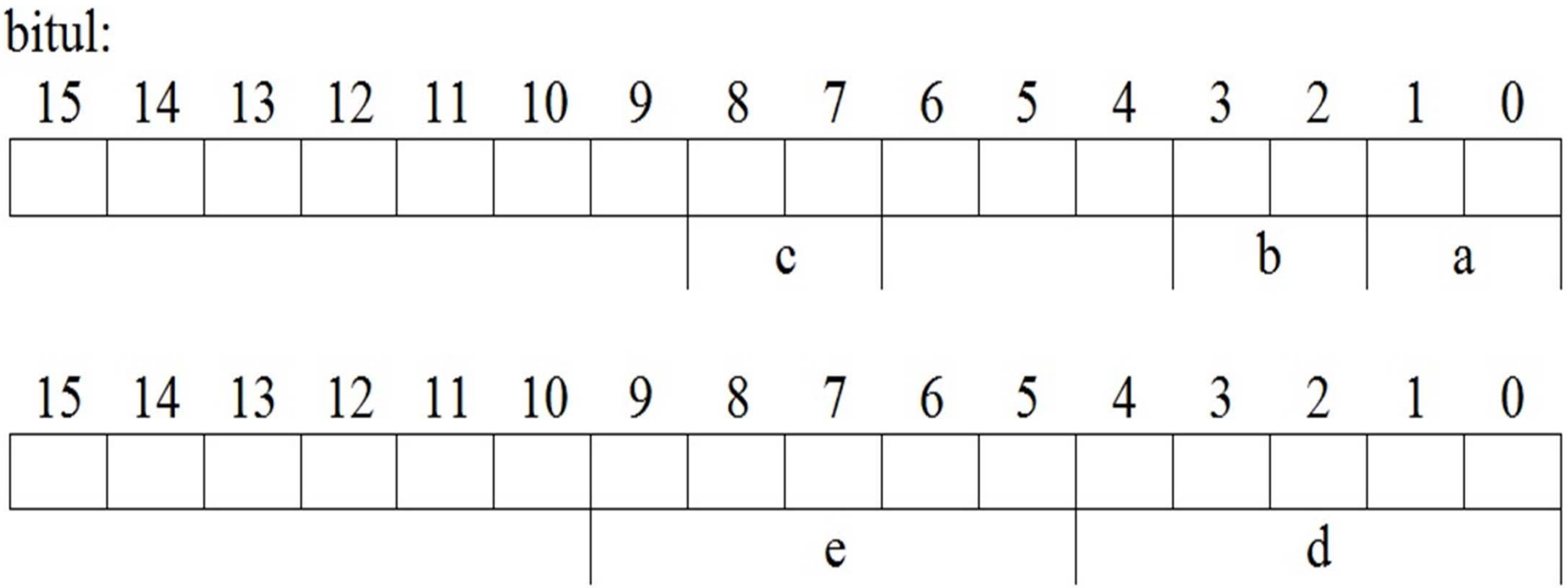
Observaţii:

- Dacă un câmp nu se poate aloca în cuvântul curent el se va aloca în cuvântul următor;
- Un câmp fără nume nu se poate referi. El defineşte o zonă neutilizată dintr-un cuvânt;
- Dacă lungimea în biţi a câmpului este 0, automat data următoare se alocă în cuvântul următor.

Exemplu:

```
struct
        { unsigned a:2;
          int b:2;
        unsigned :3;
        unsigned c:2;
        unsigned :0;
        int d:5;
        unsigned e:5;
        }x,y;
```

Pentru *x* se alocă două cuvinte astfel:



Observaţii:

- Nu se pot defini tablouri la câmpuri.
- Operatorul & nu se aplică la câmpuri.
- Programele ce utilizează câmpuri nu sunt portabile sau au o portabilitate redusă.
- Datele pe biţi necesită operaţiuni suplimentare (deplasări, setări, mascări de biţi, etc.). Utilizarea lor se justifică numai dacă se obţine o economie substanţială de memorie faţă de alocarea pe octeţi sau pe cuvinte de 16 biţi (de exemplu la sisteme programabile dedicate unde spaţiul de memorie este critic).

7. Tipul enumerare

- Tipul enumerare permite programatorului să folosească nume sugestive pentru valori numerice.
- Ca exemplu, pentru denumirea unei luni din an, ianuarie poate fie asociat cu valoarea 1, februarie asociat cu valoarea 2, etc.
- Tipul enumerare este folosit atunci când nu se vrea utilizarea unor numere întregi succesive, ci a unor simboluri asociate.

Formatul general al tipului enumerare este:

enum nume{nume_0,nume_1,nume_2,...,nume_k} v1,v2,...,vn;

unde:

- *nume* este numele tipului de enumerare introdus prin această declaraţie;
- *nume_0, ..., nume_k* sunt nume care se vor utiliza în locul valorilor numerice (*nume_i* are valoarea *i*);
- *v1, v2, ..., vn* sunt date care se declară de tipul *nume* (sunt similare cu datele de tip *int*).

Ca observaţie, se pot defini şi ulterior date de tip *nume*:

enum nume v1,v2,...,vn;

Exemple:

1)

enum {ilegal, ian, feb, mar, apr, mai, iun, iul, aug, sep, oct, nov, dec} luna;

În acest caz, atribuirea

luna=4;

este echivalentă cu formatul mai sugestiv

luna=apr;

iar *luna==8*

este echivalentă cu

luna==aug

2) *enum Boolean {false,true} a;*

Expresia

a==false

este echivalentă cu

a==0

iar

a==true

este echivalentă cu

a==1

3) *enum S{s1, s2, s3=100, s4, s5}*

Valorile asociate sunt: s1=0; s2=1; s3=100; s4=101; s5=102.

8. Date definite recursiv

- Datele definite recursiv vizează structuri de date declarate într-un format special, care conţine o formă de autoapelare.

- Pentru a clarifica lucrurile se consideră structura:

```
struct nume
        {
        declaraţii;
        };
```

Observaţii:

- declaraţiile pot defini date de diferite tipuri (predefinite sau utilizator) dar diferite de tipul ***nume***.

- declaraţiile pot defini pointeri spre date de tipul ***nume***.

În consecinţă, o declaraţie:

```
struct nume
        {declaratii;
        struct nume *nume1;
        declaratii;
        };
```

este corectă, în timp ce:

```
struct nume
        {declaratii;
        struct nume nume1;
        declaratii;
        };
```

este *incorectă*.

Definiţie

Dacă un tip utilizator are cel puţin o componentă care este pointer spre el însuşi, atunci acel tip este *direct recursiv*.

Exemplu:

```
typedef struct nod
                { char *cuvant;
                  int frecventa;
                  struct nod *urm;
                } NOD;
```

Cap. **Liste**

1. Introducere

Ca o primă definiţie, *lista* este o mulţime dinamică, înţelegând prin aceasta faptul că ea are un număr variabil de elemente.

La început lista este o mulţime vidă. În procesul execuţiei programului se pot adăuga elemente noi listei şi totodată pot fi eliminate diferite elemente din listă.

Observaţii:

- Elementele unei liste sunt date (sau structuri) de acelaşi tip. Tipul comun elementelor unei liste este un tip utilizator.

- De multe ori listele sunt organizate sub formă de tablou. Acest lucru nu este eficient deoarece listele sunt dinamice, iar tablourile, din limbajul C, nu.

- Există situaţii în care este dificil de a evalua numărul maxim al elementelor unei liste, precum şi cazuri când numărul lor diferă de la execuţie la execuţie. În consecinţă apare problema de a organiza o astfel de mulţime de tip listă.

- Ordonarea elementelor unei liste se face cu ajutorul pointerilor care intră în compunerea elementelor listei.

- Datorită acestor pointeri, elementele listei devin structuri recursive.

- Listele organizate în acest fel se numesc liste înlănţuite.

Prin definiţie, o mulţime dinamică de structuri recursive de acelaşi tip, pentru care sunt definite una sau mai multe relaţii de ordine cu ajutorul unor pointeri din compunerea structurilor respective, se numeşte **listă înlănţuită.**

- Elementele unei liste se numesc **noduri**.

- Dacă între nodurile unei liste există o singură relaţie de ordine, atunci lista se numeşte **simplu înlănţuită**. În mod analog, lista este **dublu înlănţuită** dacă între nodurile ei sunt definite două relaţii de ordine.

- O listă este **n-înlănţuită** dacă între nodurile ei sunt definite ***n*** relaţii de ordine.

Operaţii ce ţin de o listă înlănţuită:

a) crearea listei înlănţuite;

b) accesul la un nod oarecare al listei;

c) inserarea unui nod într-o listă înlănţuită;

d) ştergerea unui nod dintr-o listă înlănţuită;

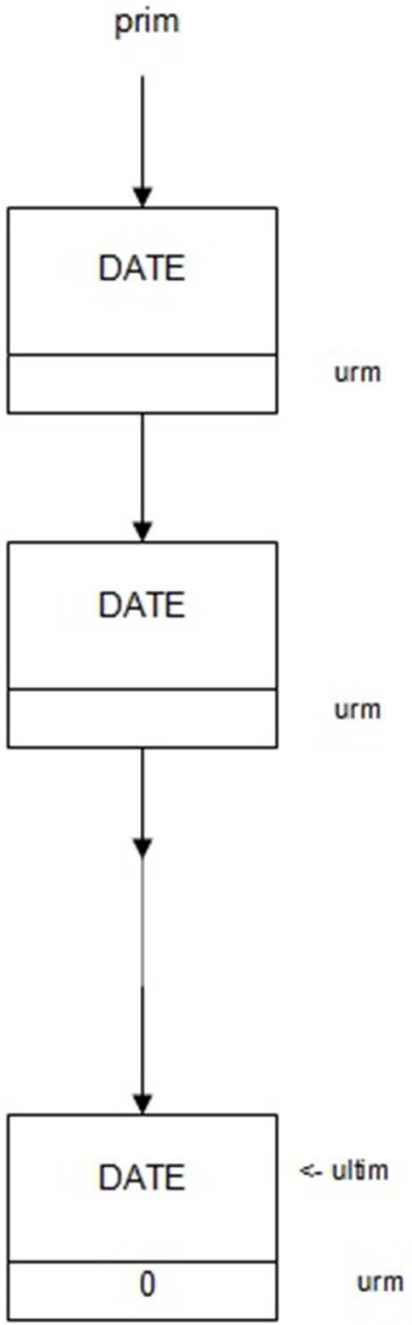
e) ştergerea unei liste înlănţuite.

2. Lista simplu înlănţuită

Între nodurile listei simplu înlănţuite avem o singură relaţie de ordonare. Există un singur nod care nu mai are succesor şi un singur nod care nu mai are predecesor. Aceste noduri formează capetele listei.

Vom utiliza doi pointeri spre cele două capete pe care îi notăm cu:

- **prim** - pointerul spre nodul care nu are predecesor
- **ultim** - pointerul spre nodul care nu are succesor.



Pointerul **urm** defineşte relaţia de succesor pentru nodurile listei. Pentru fiecare nod, el are ca valoare adresa nodului următor din listă. Excepţie face nodul spre care pointează variabila **ultim** (pentru care **urm** ia valoarea zero).

2.1. Crearea unei liste simplu înlănţuite

La crearea unei liste simplu înlănţuite se realizează următoarele operaţii:

1) Se iniţializează pointerii **prim** şi **ultim** cu valoarea zero, deoarece la început lista este vidă.

2) Se rezervă zona de memorie în memoria *heap* pentru nodul curent.

3) Se încarcă nodul curent **p** cu datele curente, dacă există şi apoi se trece la pasul ***4)***. Altfel, dacă nu există date, se șterge nodul curent **p**. Lista se consideră creată din noduri anterior introduse (daca există) şi se revine din funcţie.

4) Se atribuie adresa din memoria heap a nodului curent pointerului:

ultim -> urm = p, dacă lista nu este vidă;

prim = ultim = p, dacă lista este vidă.

5) Se atribuie lui **ultim** adresa nodului curent (**ultim=p**).

6) **ultim -> urm** = 0

7) Procesul se reia de la punctul ***2)*** de mai sus pentru a adăuga un nod nou la listă.

Observaţii:

- Se consideră tipul utilizator:

```
typedef struct nod
        { declarații;
          struct nod *urm;
        } NOD;
```

- La punctul **3)** se încarcă datele printr-o funcție pe care o vom denumi **incnod** care încarcă datele curente într-un nod de tip NOD.

- Funcţia incnod returnează:
  - 1 la încărcarea corectă a datelor în nodul curent
  - -1 când nu mai sunt de încărcat date
  - 0 la apariția unei erori (ex: memorie insuficientă)

- Funcţia incnod are prototipul:

  int incnod(NOD *p);

- O altă funcție necesară este cea care eliberează zona de memorie rezervată unui nod:

  void elibnod(NOD *p);

2.2. Accesul la un nod într-o listă simplu înlănţuită

- Putem avea acces la nodurile unei liste simplu înlănţuite începând cu nodul spre care pointează variabila globală **prim** şi trecând apoi pe rând de la un nod la altul, folosind pointerul **urm**.
- O altă metodă, mai bună, este aceea de a avea o dată, componentă a nodurilor, care să aibă valori diferite, pentru noduri diferite. În acest caz se poate defini accesul la nodul din listă pentru care data respectivă are o valoare fixată. O astfel de dată se numeşte **cheie** şi este de un tip oarecare (uzual se foloseşte char şi int). Funcţia returnează pointerul spre nodul căutat sau zero în cazul în care lista nu conţine un nod a cărui cheie să aibă valoarea indicată de parametrul ei.