Structuri şi liste înlănțuite

Tipul *structură* permite programatorului să imbine mai multe componente într-o singură variabilă. Componentele structurii au nume distincte şi se numesc membrii. Membrii unei structuri pot avea tipuri diferite. Deci, ca şi pointerii şi sirurile, structurile sunt considerate un tip derivat. Accesarea membrilor unei structuri se face cu "." sau cu "->" şi au cea mai înalta prioritate (ca şi () şi []).

Declararea structurilor
Se face folosind cuvântul rezervat "struct".
Exemplu: Declarația de mai jos creeaza tipul de dată "carte_de_joc":

```
struct carte_de_joc
        {
         int Număr ;
         char culoare;
        };
```

Astfel cartea 3 de treflă va avea "Număr =3" şi "culoare='t'". Celelalte caractere pentru culorile cărților sunt (frunza - 'f', caro - 'c', inima - 'i'). Numele structurii poate fi folosit acum pentru declararea variabilelor de acest tip. Abia în acest moment se rezervă loc în memorie pentru aceste variabile:

```
struct carte_de_joc c1, c2;
```

Pentru accesarea membrilor lui c1 şi c2, folosim operatorul ".".
Exemplu:

```
c1.Număr   = 3;
c1.culoare = 't';
c2.Număr   = 12;
c2.culoare = 'c';
```

O construcție de forma

variabila_structura . nume_membru

este folosită ca o variabilă în acelasi mod ca o simplă variabilă sau ca un element al unui şir. Numele unui membru trebuie să fie unic într-o structură specificată. Din moment ce membrii trebuie întotdeauna prefixați de un identificator de variabilă de structură unic, nu vor fi confuzii (ambiguități) între doi membri cu acelaşi nume, dar din structuri diferite.
Exemplu:

```
struct fruct
 {
  char nume[15];
  int calorii;
 }
struct leguma
 {
  char nume[15];
  int calorii;
```

```
}
struct fruct  a;
struct leguma b;
```

Putem accesa "a.calorii", respectiv "b.calorii" fără ambiguitate. Putem declara variabile de un tip structurat în timpul declarării acestuia.
Exemplu:

```
struct carte_de_joc
{
 int Număr ;
 char culoare;
} c1, c2, c3[52];
```

Identificatorul "carte_de_joc" este numele structurii. Identificatorii "c1" şi "c2" se declară ca fiind variabile de tip "struct carte_de_joc", iar identificatorul "c3" ca fiind un şir de tip "struct carte_de_joc". Dacă însă lipseşte numele structurii, atunci singura dată când se pot declara variabile de tip structură este în momentul declararii acesteia.
Exemplu:

```
struct
{
 char  *nume;
 int   nr_student;
 float medie;
} s1, s2, s3;
```

In această structură se declară trei variabile de tip structură, însă lipsind numele structurii înseamnă că nu se mai pot declara ăi alte variabile de acest tip. Dacă, de exemplu, scriem

```
struct student
{
 char  *nume;
 int   nr_student;
 float medie;
};
```

atunci "student" este numele structurii şi nu sunt variabile declarate în acest moment. Acum putem scrie

```
struct student temp, clasa[100];
```

şi declarăm "temp" şi "clasa" de tip "struct student".

Accesarea unui membru
În cele ce urmează, prezentăm un exemplu de folosire a operatorului de membru ".".
Exemplu:
În fisierul "cl_info.h" scriem:

```
#define NR_STUDENTI 100
struct student
{
 char  *nume;
 int   nr_student;
```

```
 float medie;
};
```

În alt fişier, scriem

```
#include "cl_info.h"
void main()
{
 struct student temp, clasa[NR_STUDENTI];
  . . . . .
}
```

Putem avea instrucţiuni de asignare cum ar fi:

```
temp.medie      = 4.00;
temp.nume       = "Ionescu";
temp.nr_student = 1023;
```

În continuare, scriem o funcţie care numără studenţii cu media 4.00:

```
int esec(struct student clasa[])
{
 int i, contor = 0;
 for (i = 0; i < NR_STUDENTI; ++i)
        contor += clasa[i].medie == 4.00;
 return contor;
}
```

Limbajul C pune la dispoziţie operatorul pointer către structura -> pentru accesarea membrilor unei structuri relativ la un pointer. Dacă o variabilă pointer este asignată cu adresa unei structuri, atunci un membru al structurii poate fi accesat printr-o construcţie de forma:

```
pointer_catre_structura -> nume_membru
```

O construcţie echivalentă este:

```
(*pointer_catre_structura).nume_membru
```

Parantezele sunt necesare deoarece operatorii "." şi "->" au prioritate mare şi se asociază de la stânga la dreapta. Astfel, parantezele sunt obligatorii, deoarece dacă în expresia de mai sus nu ar fi fost paranteze, atunci aceasta ar fi echivalentă cu

```
*(pointer_catre_structura.nume_membru)
```

Exemplu: Fie următoarele declaraţii şi asignari:

```
struct student temp, *p = &temp;
temp.medie      = 10.00;
temp.nume       = "Ionescu";
temp.nr_student = 1204;
```

Atunci obţinem următorul tabel:

Expresie	Expresie echivalenta	Valoare conceptuala
temp.medie	p -> medie	10.00
temp.nume	p -> nume	Ionescu
temp.nr_student	p -> nr_student	1204
(*p).nr_student	p -> nr_student	1204

Obdervație Operatorul unar "sizeof" poate fi folosit pentru determinarea numărului de octeți necesar memorării sale. De exemplu, expresia sizeof(struct carte_de_joc) va intoarce numățul de octeti necesari sistemului pentru memorarea unei variabile de tip "struct carte_de_joc". Pe cele mai multe sisteme tipul returnat de expresie este "unsigned".

Structuri, functii şi asignari

Limbajul C permite unui pointer către un tip structură să fie transmis ca argument al unei funcții şi returnat ca o valoare. ANSI C permite chiar unei structuri să fie trimisă ca argument pentru funcții şi returnată ca valoare. De exemplu, dacă "a" şi "b" sunt structuri, atunci expresia de asignare "a = b" este validă. Aceasta înseamnă că fiecare valoare a unui membru din structura "a" devine egală cu valoarea membrului corespunzător din structura "b". în C, structurile, pointerii şi vectorii pot fi combinați pentru crearea unor structuri de date complicate.

Exemplu: Baza de date cu studenti
In fisierul "student.h":

```
#define NR_STUDENTI 50
#define NR_CURSURI  10
struct student
 {
  char *nume;
  int nr_student;
  float medie;
 };
 struct data
 {
  short zi;
  char luna[10];
  short an;
 };
struct persoana
 {
  char nume[20];
  struct data zi_nastere;
 };
struct date_student
 {
  struct persoana p;
  int nr_student;
  float medie[NR_CURSURI];
 };
```

Observăm că "struct date_student" este construită cu structuri imbricate. De exemplu, dacă avem declarația:

struct date_student temp;

atunci expresia:

temp.p.zi_nastere.luna[0]

are ca valoare prima litera a lunii datei de naştere a studentului asociat lui "temp".

In continuare, vom scrie o funcție "citeste_date()" pentru a introduce date în variabile de tip "struct date". La apelul funcției, trebuie trimisă adresa variabilei ca argument.

Exemplu:

```
#include "student.h"
void citeste_date(struct data *z)
 {
  printf("Dati ziua(int) luna(string) an(int): ");
  scanf("%hd%s%hd", &z -> zi, z -> luna, &z -> an);
 }
```

Formatul %hd este folosit pentru conversia caracterelor de la tastatura la o valoare de tip "short". Funcția "citeste_date()" poate fi folosită pentru citirea informației într-o variabilă de tip "struct date_student" astfel:

```
struct date_student temp;
citeste_date(&temp.p.zi_nastere);
```

Initializarea structurilor

Toate variabilele externe şi statice, inclusiv variabilele de structură, care nu sunt explicit inițializate, sunt automat inițializate de către sistem cu zero. în C traditional, structurile statice şi externe pot fi inţtializate de catre programator. în ANSI C, putem iniția liza şi structuri definite "auto". Sintaxa este similară celei folosite la şiruri. O variabilă structura poate fi urmată de semnul "=" şi o lista de constante cuprinse intre acolade. Dacănu sunt suficiente valori pentru asignarea lor, atunci membrii ramasi sunt asignati cu zero implicit.

Exemple:

```
struct carte_de_joc  c = {12, 't'};
struct complex
  {
   double real;
   double imaginar;
  } m[3][3] =
    {
     {{1.0, -0.5}, {2.5, 1.0}, {0.7, 0.7}},
     {{7.0, -6.5}, {-0.5,1.5},{45.7,8.0}},
    };
```

Se observă că linia "m[2][]" este inițializată cu 0.

Facilitatea "typedef" este deseori folosită pentru redenumirea unui tip structură.

Exemple:

```
typedef char *  string;
typedef int     lungime;
typedef float   vector[10];
typedef double  (*PFD)(double);
```

După aceste redenumiri, putem face declarațiile:

lungime l1, l2;
string s1 = "abc", s2 = "xyz";
vector x;

Aceste declarații sunt echivalente cu:

int l1, l2;
*char * s1 = "abc", s2 = "xyz";*
float   x[10]; / Atentie ! Se inlocuieste vector cu x */*

La fel, declarația

PFD f;

este echivalentă cu

*double (*f)(double);*

Este vorba de un pointer la o funcție ce returnează tipul "double".

Structuri recursive

Prin analogie cu algoritmii, prin *structură recursivă* se înțelege o structură care are cel puțin o componentă de acelasi tip cu structura însăşi.
Pentru evitarea structurilor infinite, în definiția recursivă trebuie să existe o condiție de care depinde prezența efectiva a componentei sau componentelor recursive.
Structurile recursive pot fi implementate în C numai în forma unor structuri dinamice, deoarece o structura nu poate avea un câmp de acelaşi tip structura, ci doar pointer la structură.
Deci definiția unei structurii recursive recursivă va deveni:
o structură care are o componentă de tip pointer la structura însăşi.
Limitarea structurilor recursive se realizează prin existența valorii NULL pentru pointeri.
La modul general, o structura recursivă se defineste:

```
struct recursiva{ //
   tip1 camp1; // m campuri de tipuri oarecare
   tip2 camp2;
   ...
   tipm campm;
   struct recursivă *p1, *p2, ..., *pn; // L
   // n campuri de tip pointer la structura
};
```

Cu extensiile aduse de C++, cum numele unei structuri este nume de tip, linia L devine:

recursivă *p1, *p2, ..., *pn;

In continuare ne vom referi la structurile recursive care au *o singură componentă pointer* la structură.

Structura de date lista

Lista este o *structură dinamică*, situată in memoria centrală, în care toate elementele sunt de acelasi tip; numărul de elemente este variabil, chiar nul.

De remarcat diferențele față de definiția *tabloului*: tabloul este o *structura statică*, situată in memoria centrală, în care toate elementele sunt de acelaşi tip; numărul de elemente este constant.
O alta definire a listei este:
O lista L este o secventă de zero sau mai multe elemente, numite *noduri*, toate fiind de acelaşi *tip de baza* T.
L=a1,a2,...,an (n>=0)
Dacă n>=1, a1 se spune că este *primul nod* al listei, iar an, *ultimul nod*. Dacă n=0, lista este *vidă*.
Numărul de noduri se numeşte *lungimea* listei.
Un *nod* al listei liniare care apare ca o structură recursivă, având o componentă de tip pointer la structură, reprezentând *legatura* (înlănțuirea) spre nodul următor.
Lista in care fiecare nod are o singură înlănțuire se numeşte *listă simplu înlănțuită*.

```
struct nod{
      TipCheie cheie;
      TipInfo info;
      struct Nod* urmator;
 };
 struct nod* inceput;
 //pentru o scriere mai compacta se pot defini tipurile:
 typedef struct nod Nod, *pNod;
 pNod inceput;
```

Caracteristica unei astfel de structuri constă în prezența unei singure înlănțuiri. Câmpul *cheie* serveşte la identificarea nodului, câmpul *urmator* e pointer de înlănțuire la nodul următor, iar *info* conține informația utilă.
Variabila *inceput* indică spre primul nod al listei. In unele situații în locul lui început se utilizează un *nod fictiv*, adică o variabilă de tip struct Nod cu câmpurile *cheie* şi *info* neprecizate, dar campul *urmator* indicând spre primul nod al listei.
De asemenea uneori este util a se pastra pointerul spre ultimul nod al listei.
O variantă este cea a *listelor circulare* la care dispare noțiunea de prim, ultim nod, lista fiind un pointer ce se plimbă pe listă.

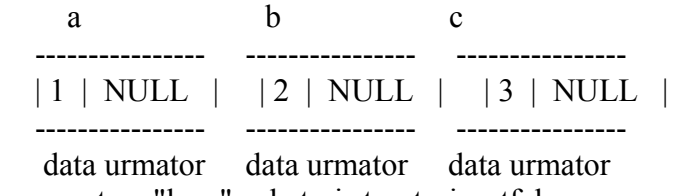
Presupunem că avem declarațiile

```
struct lista a, b, c;
```

Vrem să construim o listă înlănțuită formată din aceste trei variabile. Mai întâi, facem asignările:

```
a.data = 1;
b.data = 2;
c.data = 3;
a.urmator = b.urmator = c.urmator = NULL;
```

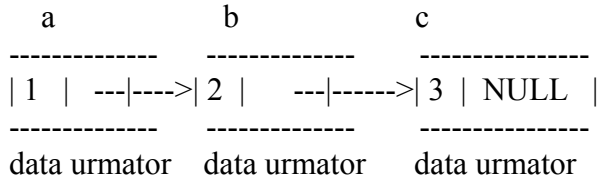
După aceste instrucțiuni, obținem în memorie:



Acum putem "lega" cele trei structuri, astfel:

```
a.urmator = &b;
b.urmator = &c;
```

Obținem:



Operații pentru liste

Operațiile de bază pentru liste liniar înlănțuite includ următoarele:

1. Crearea unei liste
2. Inserarea unui element
3. Ştergerea unui element
4. Traversarea listei
5. Numărarea elementelor unei liste
6. Căutarea unui element

- *Inserția unui nod la începutul listei*

Dacă *inceput* este variabila pointer ce indică spre primul nod al listei, iar *q* o variabilă auxiliară de tip pointer, secvența următoare realizează inserția la inceputul listei şi actualizează pointerul *inceput*:

```
pNod inceput, q;
q=(pNod)malloc(sizeof(Nod)); //creaza spatiu pentru un nou nod
q->urmator=inceput;
//asignarea campurilor cheie si info
inceput=q;
```

Secvența este corectă şi pentru inserția intr-o listă vida, caz in care valoarea lui *inceput* este NULL.

- *Insertia unui nod la sfarsitul listei*

Devine mai simplă dacă se pastrează o variabilă sfârşit indicând spre ultimul nod al listei:

```
pNod inceput, sfarsit, q;
q=(pNod)malloc(sizeof(Nod)); //creaza spatiu pentru nou nod ultim
sfarsit->urmator=q;
q->urmator=NULL;
//asignarea campurilor cheie si info
sfarsit=q;
```

Pentru insertia la sfârşitul listei este necesară existența a cel putin a unui nod, care se crează prin secventa de la paragraful anterior.

- *Insertia unui nod dupa unul indicat*

Este simpla pentru că se cunoaste pointerul spre nodul anterior şi spre cel următor celui care se inserează (pointerul spre nodul următor este valoarea câmpului următor al nodului indicat).

- *Inserţia unui nod in faţa unui nod indicat*

Printr-un artificiu, se reduce acest caz la cel anterior: se inserează un nod după cel indicat, cheia şi informatia din nodul indicat fiind atribuite noului nod inserat şi fiind inlocuite cu valorile nodului care trebuia inserat.

- *Tehnici de suprimare a nodurilor*

Pentru suprimarea nodului următor celui indicat de o variabilă pointer q, prin atribuirea:
q->urmator=q->urmator->urmator;
se exclud din listă legăturile la şi de la nodul de suprimat.

Pentru suprimarea nodului anterior unuia precizat, se aduce nodul precizat in cel de suprimat. Apoi se suprimă succesorul lui, deci cel indicat iniţial de variabila pointer q:
*q=*q->urmator; /* pentru expresia din dreapta, operatorul de selectie indirecta -> este mai prioritar decât cel de indirectare *, deci nu sunt necesare parantezele */

- *Traversarea unei liste*

Dacă nodul de inceput al listei este indicat de variabila inceput, o variabilă auxiliara q, care parcurge toate nodurile listei până când valoarea ei devine NULL, permite accesul la fiecare nod şi efectuarea operaţiei specifice traversării:

```
for(q=inceput;q!=NULL;q=q->urmator)
  //prelucrarea nodului indicat de q
```

Dacă lista are două noduri fictive, unul de inceput şi unul de sfârşit, secvenţa de traversare devine:

```
for(q=inceput->urmator;q!=sfarsit;q=q->urmator)
  //prelucrarea nodului indicat de q
```

- **Liste ordonate si reorganizarea listelor**

a) Căutarea intr-o listă neordonata; tehnica fanionului

Se consideră o listă simplu inlănţuită, cu nodurile de tip Nod. Dacă *inceput* indică spre primul nod al listei, iar ordinea cheilor in lista este aleatoare, căutarea unei chei implica traversarea listei.
Funcţia booleană gasit returnează valoarea pointerului spre nodul cu cheia egală cu cea căutată, dacă un astfel de nod există şi valoarea NULL in caz contrar:

```
pNod gasit(TipCheie val){
        pNod poz;
        poz=inceput;
        while (poz!=NULL)
            if (poz->cheie==val)
```

```
            return poz;
        else
            poz=poz->urmator;
    return poz; // cu valoarea NULL
```

Căutarea se poate perfecționa prin utilizarea *metodei fanionului*, lista prelungindu-se cu un nod fictiv numit *fanion*, la creare lista conținând acest unic nod. In funcția gasit, inainte de baleierea listei, informația căutată se introduce in cheia nodului fanion, astfel incât va exista cel puțin un nod cu cheia căutată:

```
pNod fanion;
pNod gasit(TipCheie val){
    pNod poz;
    for(poz=inceput,fanion->cheie=val;poz->cheie!=val;
       poz=poz->urmator);
    if(poz==fanion)
        return NULL;
    return poz;
```

b)Crearea unei liste ordonate; tehnica celor doi pointeri

In continuare se prezintă o metodă foarte simplă pentru crearea unei liste ordonate, tipurile pNod şi Nod fiind cele definite anterior. Lista se inițializează cu două noduri fictive pointate de două variabile pointer, *inceput* şi *fanion*:

```
pNod inceput, fanion;

void init(void);
   inceput=(pNod)malloc(sizeof(Nod));
   fanion=(pNod)malloc(sizeof(Nod));
   inceput->urmator=fanion;
}
```

Pentru introducerea unei noi chei in lista, păstrând ordonarea, se va scrie o funcție găsit, care dacă găseste cheia în listă returnează valoarea 1 şi pointerii p1 spre nodul găsit şi p2 spre cel anterior, respectiv în cazul negăsirii cheii, valoarea 0 şi pointerii p1 si p2 spre nodurile intre care trebuie făcută inserția:

```
int gasit(TipCheie val, pNod* p1, pNod* p2){
    for(*p2=inceput,*p1=(*p2)->urmator,fanion->cheie=val;
       (*p1)->cheie<=val;p2=p1,*p1=(*p1)->urmator);
    return *p1!=fanion && (*p1)->cheie==val;
```

Fragmentul de program care insereaza o noua cheie este:

```
  pNod p1,p2,p3;
   TipCheie val;
       ...
   if (!gasit(val,&p1,&p2)){
        p3=(pNod)malloc(sizeof(Nod)); //creare nod nou
        p2->urmator=p3; //legatura de la nodul anterior la cel nou
        p3->cheie=val;
```

```
                    //completare p3->info
        p3->urmator=p1; //legatura de la noul nod la cel urmator
   }
```

Pentru tipărirea cheilor dintr-o listă ordonată astfel creată, pointerul care parcurge nodurile trebuie să fie inițializat cu valoarea pointerului spre primul nod efectiv al listei, următor celui inceput, iar parcurgerea listei se face până la intâlnirea nodului fanion:

```
   pNod p;
   for(p=inceput->urmator;p!=fanion;p=p->urmator)
     //prelucrarea nodului indicat de p
```

Aplicație

Să se realizeze un program care răspunde la următoarele cerințe:
'a' - Citeşte o linie de forma
identificator, numar ;
unde numar este un număr intreg şi reține in evidență identificatorul impreună cu numărul asociat lui.
't' - Citeşte o linie care conține un identificator. Dacă acesta se găseste in evidență se tipăreşte valoarea asociată lui, in caz contrar se tipăreşte un mesaj de eroare.
's' - Citeşte o linie ce conține un identificator şi il şterge din evidență.
'l' - Tipăreşte in ordine alfabetică identificatorii din evidență impreună cu valorile asociate lor
'+' - Citeşte două linii, fiecare continând un identificator. In cazul in care ambii se află in evidență, se tipăreşte suma lor. In caz că unul sau ambii identificatori lipsesc din evidență se tipăreşte un mesaj de eroare.
'-' - Citeşte două linii, fiecare conținând un identificator. In cazul in care ambii se află in evidenta, se tipăreşte diferenta lor. In caz ca unul sau ambii identificatori lipsesc din evidenta se tipăreşte un mesaj de eroare.
'*' - Citeşte doua linii, fiecare conținând un identificator. In cazul in care ambii se află in evidenta, se tipăreşte produsul lor. In caz ca unul sau ambii identificatori lipsesc din evidenta se tipăreşte un mesaj de eroare.
'/' - Citeşte doua linii, fiecare conținând un identificator. In cazul in care ambii se află in evidenta, se tipăreşte rezultatul impărțirii lor. In caz că unul sau ambii identificatori lipsesc din evidenta se tipăreşte un mesaj de eroare.
'f' - Se termină programul
In listă identificatorii sunt pastrati in mod ordonat.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define Max 10
typedef struct  elem{
        char *id;
        int valoare;
        struct elem  *urm;
} Nod,*pNod;
void listare( void );
pNod cauta( char* );
```

```
void sterge( char* );
void introdu( char*, int );
nod *radacina = NULL;
pNod ccauta( pNod lista , char *s )
     {
       pNnod q1 ;
       for( q1 = lista ; q1 != NULL && strcmp( q1->id,s ) < 0 ;
            q1 = q1->urm );
       if ( q1 != NULL && strcmp( q1->id,s ) == 0 )
                return q1;
       return NULL;
     }
void clist( pNod lista )
     {
       pNod q;
       for( q = lista ; q != NULL ; q = q->urm )
          printf(" identificator:%s, valoare:%d\n",q->id,q->valoare);
     }
pNod csterge( pNod lista, char *s )
     {
       pNod q1 , q2;
       for( q1 = q2 = lista ; q1 != NULL && strcmp( q1->id,s ) < 0;
        q2 = q1, q1 = q1->urm);
       if( q1 != NULL && strcmp( q1->id,s ) == 0 )
          if( q1 != q2 )
            {
              q2->urm = q1->urm;
              return lista ;
            }
           else  return lista->urm;
        else
          {
          printf("Eroare:identificatorul %s nu apare in lista\n",s);
          return lista;
          }
     }
pNod cintrodu( pNod lista , char *s ,int v)
     {
       pNod q1 , q2 , aux;
       if( ( aux = (pNod)malloc(sizeof(Nod))) == NULL ||
          ( aux->id = (char *) malloc(strlen(s) +1)) == NULL )
           {
            printf(" Eroare: memorie insuficienta\n");
            exit( 1 );
           }
        else
        {
         strcpy( aux->id,s );
         aux->valoare = v;
        }
```

```
        for( q1 = q2 = lista ; q1 != NULL && strcmp( q1->id,s ) < 0;
          q2 = q1, q1 = q1->urm);
        if( q1 != NULL && strcmp( q1->id,s ) == 0 )
         {
              printf(" Eroare : %s apare in lista\n",s);
              return lista;
         }
         else
            if( q1 != q2 )
              {
                 q2->urm = aux ;
                 aux->urm = q1;
                 return lista ;
              }
              else
              {
                 aux->urm = lista;
                 return aux;
               }
      }
void listare(void)
      {
       clist( radacina );
      }
pNod cauta( char *s )
      {
       return ccauta( radacina , s );
      }
void sterge( char *s )
      {
       radacina = csterge ( radacina, s );
      }
void  introdu( char *s, int v)
      {
       radacina = cintrodu( radacina, s, v );
      }
void getlin( char *s , int *val )
      {
       int i = 0, j;
       char temp[ Max ];
       gets( temp );
       s[ i ] = '\0' ; *val = 0;
       while( temp[ i ] != '\0' )
         if( isalpha( temp[ i ] ))
          {
           j = 0;
           while( isalnum( temp[ i ])) s[ j++ ] = temp[ i++ ];
           s[ j ] = '\0';
          }
          else
```

```
            if( isdigit( temp[i] ))
              while( isdigit( temp[i] ))
              {
                *val = *val * 10 + temp[i] - '0' ;
                i++;
              }
              else i++;
        }
void ad( void )
       {
       int val;
       char s[ Max ];
       getlin( s , &val );
       if( strlen( s ) != 0 ) introdu( s , val );
        else printf(" Eroare : linie incorecta \n");
       }
void ti( void )
       {
        char s[ Max ];
        nod *p;
        int val;
        getlin( s , &val );
        if( strlen( s ) != 0 )
         {
          if( ( p = cauta( s )) != NULL )
            printf("identificator:%s,valoare :%d \n",p->id,p->valoare);
          else printf(" Eroare: identificator nedefinit \n");
        }
         else printf(" Eroare: linie incorecta \n");
       }
void st( void )
       {
        char s[ Max ];
        int val;
          getlin(s , &val );
          if( strlen( s ) != 0 )  sterge( s );
           else printf(" Eroare: linie incorecta \n");
       }
void oper( char c )
       {
        char s1[ Max ] , s2[ Max ];
        int val;
        nod *p1 , *p2;
        getlin(s1 , &val);
        getlin(s2, &val);
        if(( strlen( s1 ) != 0 ) && ( strlen( s2 ) != 0))
          if((( p1 = cauta( s1 )) != NULL) &&
             (( p2 = cauta( s2 )) != NULL ))
          {
           switch( c )
```

```
          {
           case '+' : val = p1->valoare + p2->valoare ; break;
           case '-' : val = p1->valoare - p2->valoare ; break;
           case '*' : val = p1->valoare * p2->valoare ; break;
           case '/' :  if( p2-> valoare != 0 )
                          val = p1->valoare / p2->valoare ;
                       else printf(" Eroare: Impartire la 0\n") ;
                       break;
          }
          printf(" Rezultatul operatiei e %d \n", val);
         }
          else printf(" Eroare: linie eronata \n ");
     }
void meniu(void)
      {
       char o;
       while( 1 )
       {
        clrscr();
        printf(" a .... adauga in evidenta un id si val asociata \n");
        printf(" t .... tipăreşte valoarea asociata unui id \n");
        printf(" s .... sterge un id \n");
        printf(" l .... listeaza  id-rii si valorile asociate \n");
        printf(" + .... calculeaza suma pt. 2 id \n");
        printf(" - .... calculeaza diferenta pt. 2 id \n");
        printf(" * .... calculeaza produsul pt. 2 id \n");
        printf(" / .... calculeaza impartirea pt. 2 id \n");
        printf(" f .... termina programul \n");
        printf("\n Introduceti optiune :"); o = getchar(); getchar();
        switch(tolower(o))
        {
         case 'a': ad();break;
         case 't': ti();break;
         case 's': st();break;
         case 'l': listare();break;
         case '+': case '-' : case '*' : case '/' : oper(o); break;
         case 'f' : return;
         default : printf(" Eroare : Comanda inexistenta \n");
        }
       getchar();
      }
     }
void main( void )
      {
       meniu();
      }
```

Funcția *ccauta* caută un identificator in listă returnând pointerul nodului in care s-a gasit identificatorul. Funcția *clist* afişează conținutul listei, *csterge* elimină un nod din listă, iar

cintrodu va introduce un nod in listă. Aceste funcții au ca şi parametru formal pointerul de inceput al listei prelucrate.
Folosind aceste funcții se definesc *cauta, sterge* şi *introdu* cu rolul de căutare , ştergere şi introducere a unui identificator, precum si funcția *listare*, care va lista identificatorii din lista.
Funcția meniu afişează meniul programului, preia un caracter , reprezentând o comandă , care se va prelucra prin apelul funcției asociate.
Funcția *oper* are ca şi parametru operatorul de executat. Funcția Citeşte doi identificatori găseşte in lista valorile asociate lor, după care execută şi afişează rezultatul operației.