```

# 1

# Noţiuni introductive referitoare la utilizarea calculatoarelor personale

## 1.1. Generalităţi privind limbajele de programare

Comunicarea dintre om şi calculator este facilitată de un limbaj. Prin intermediul limbajului, operatorul uman transmite calculatorului un program pe care acesta urmează să îl execute, motiv pentru care limbajul se numeşte limbaj de programare.

Primul pas în evoluţia limbajelor de programare l-a constituit limbajul de asamblare. Acesta păstrează o strânsă legătură cu limbajul calculatorului (limbaj cod-maşină, care este o succesiune de coduri binare corespunzătoare formei de memorare specifice calculatorului), fiecare instrucţiune fiind corespondente unei operaţii elementare pe care acesta o poate executa. Prin utilizarea reprezentării text (şir de caractere) a instrucţiunilor, limbajul de asamblare a uşurat mult scrierea programelor de calcul. Translatarea programelor din forma text în forma binară înţeleasă de calculator este realizată de un program specializat, numit asamblor.

Restricţia de a avea o corespondenţă directă între instrucţiunile limbajului şi codul maşină a fost înlăturată de limbajele de nivel înalt. Acestea folosesc instrucţiuni apropiate de raţionamentul uman care sunt transformate într-un grup de instrucţiuni specifice calculatorului de către un program specializat numit compilator. Compilatorul este dedicat unui anumit limbaj şi unui anumit calculator.

Indiferent de gradul de specializare a limbajului de nivel înalt, pentru elaborarea unui program de calcul se parcurg următoarele etape:

- Editarea fişierului sursă – constă în scrierea programului folosind regulile sintactice specifice limbajului utilizat;
- Compilarea fişierului sursă – transformarea instrucţiunilor programului sursă în instrucţiuni maşină cu ajutorul compilatorului. În cadrul acestei etape pot să apară erori sintactice generate de utilizarea incorectă a regulilor de scriere a programului sursă. Pentru
```

remedierea acestor erori se revine în etapa de editare a textului programului (a fişierului sursă). În cazul în care compilatorul nu detectează erori sintactice, el va genera un fişier numit fişier obiect care are numele fişierului sursă şi extensia obj;

- Editarea legăturilor – un program specializat, numit link-editor asamblează mai multe module obiect şi generează programul executabil sub forma unui fişier cu extensia exe. Şi în cadrul acestei etape pot să apară erori, generate, de regulă, de incompatibilitatea modulelor obiect asamblate. Şi de această dată programatorul trebuie să intervină pentru a elimina erorile, calculatorul nefiind capabil să le elimine singur;
- Lansarea în execuţie a programului executabil - prin tastarea numelui acestuia (numele fişierului fără extensia exe) la apariţia cursorului sistemului de operare. Şi în această etapă pot apare erori datorate fie datelor eronate transmise calculatorului, fie concepţiei greşite a programului. Programatorul trebuie să reanalizeze algoritmul de calcul care a stat la baza elaborării programului, precum şi datele pe care le-a introdus.

Din cele prezentate reiese clar că un calculator, oricât de performant ar fi, nu se poate substitui omului. El este şi va rămâne o unealtă a acestuia, destinată să-l ajute şi să-i uşureze munca.

1.2. Limbajul de programare C

Creat iniţial pentru realizarea şi dezvoltarea sistemului de operare UNIX, limbajul C a devenit ulterior cel mai utilizat limbaj de programare, atât pentru pentru crearea şi dezvoltarea sistemelor de operare, cât şi pentru realizarea programelor de aplicaţie.

Limbajul C este unicul limbaj de programare care utilizează forma limbajelor de nivel înalt, permiţând în acelaşi timp acţiuni asupra hardului calculatorului, similare celor realizate prin intermediul limbajului de asamblare. Numeroase aplicaţii efectuate pe un calculator personal cu ajutorul limbajului de asamblare pot fi realizate într-o manieră mult mai simplă cu ajutorul limbajului C. Cele mai bune compilatoare C generează programe executabile a căror viteză de execuţie doar în puţine cazuri poate fi îmbunătăţită prin rescrierea programului în limbajul de asamblare.

Fiind un limbaj bine structurat, prin sintaxa sa, limbajul C uşurează activitatea de scriere a programelor, care sunt modulare şi uşor de înţeles. El oferă numeroase facilităţi destinate să ajute programatorul în realizarea şi depanarea celor mai complexe programe de aplicaţie.

Un alt avantaj oferit de limbajul C îl constituie portabilitatea programelor realizate cu ajutorul lui. Un program scris în C, destinat rulării pe un anumit tip de calculator, poare fi modificat cu uşurinţă în vederea utilizării sale pe alte tipuri de calculatoare. Se spune că C-ul oferă cel mai ridicat grad de portabilitate.

1.3. Structura unui program C

În general, un program de calcul este o secvenţă de operaţii care se efectuează asupra unor date şi care implementează un alguritm sau o procedură de rezolvare a unei probleme. El constă dintr-o succesiune de caractere cu ajutorul cărora se alcătuiesc cuvintele şi propoziţiile (instrucţiunile). Acestea, asemănător limbajului natural, trebuie să respecte setul de reguli ce definesc sintaxa limbajului. Pentru scrierea textului ce constituie programul de calcul se pot folosi numai caractere ce alcătuiesc alfabetul acestuia. Alfabetul C cuprinde literele mari şi mici de la a la z, caracterul de subliniere "_" folosit ca element de legătură, cifrele zecimale de la 0 l 9 şi simbolurile speciale care reprezintă operatorii, delimitatorii şi semnele de punctuaţie.

În afara alfabetului, limbajul C este prevăzut cu un set de cuvinte, numite cuvinte cheie, fiecare având o specificaţie specială.

În numeroase situaţii practice, procedura de calcul este lungă şi complicată, astfel încât programul devine dificil de implementat, urmărit şi depanat. Tehnicile de programare procedurală oferă metode pentru împărţirea şi structurarea programului în module uşor de realizat, înţeles şi întreţinut. Aceste tehnici concentrează atenţia asupra organizării secvenţei de operaţii în cadrul programului şi de cele mai multe ori ignoră structurile de date asupra cărora se efectuează operaţiile.

Spre deosebire de programarea procedurală, programarea orientată pe obiect mai întâi concentrează atenţia asupra structurilor de date (care sunt mia bine organizate şi protejate) şi apoi asupra operaţiilor care se efectuează cu acestea.

Limbajul C este un limbaj de programare procedural, operaţiile fiind grupate în blocuri de instrucţiuni delimitate de o pereche de acolade "{}". La rândul lor, blocurile sunt grupate în două sau mai multe funcţii. O funcţie C este un modul care grupează în interiorul unei perechi de acolade un set de operaţii codificate sub formă de instrucţiuni.

Datele asupra cărora operează funcţia sunt fie intrările în funcţie, fie date locale ale acesteia. La rândul lor, intrările în funcţie pot fi date globale

ale programului de calcul şi/sau date transmise funcţiei prin intermediul parametrilor acesteia.

Rezultatele obţinute în urma efectuării operaţiilor funcţiei constituie ieşirile acesteia. Ele constau din modificările efectuate asupra datelor globale, modificările efectuate prin intermediul parametrilor de tip pointer, valoarea întoarsă de funcţie, sau alte acţiuni.

Fiecare funcţie C are un nume precedat de un cuvânt cheie care desemnează tipul funcţiei (tipul valorii întoarse de funcţie). Numele funcţiei este urmat de o pereche de paranteze rotunde între care se specifică tipul şi numele parametrilor funcţiei.

Există funcţii care nu au parametri sau nu furnizează nici un rezultat. O astfel de funcţie se specifică prin folosirea cuvântului cheie **void** în interiorul perechii de paranteze ce grupează argumentele, respectiv în faţa numelui ce desemnează funcţia.

Orice program scris în limbajul C, indiferent de gradul lui de complexitate, trebuie să conţină o funcţie al cărei nume este **main**. Această funcţie preia controlul de la sistemul de operare în momentul în care programul este lansat în execuţie.

Cel mai simplu program C este :

```
void main (void)
{
}
```

şi constă din funcţia main al cărei corp, delimitat de perechea de acolade, este gol. Fiind corect din punct de vedere sintactic, programul poate fi compilat, link-editat şi lansat în execuţie. Deoarece corpul funcţiei este vid în momentul lansării în execuţie acesta primeşte controlul de la sistemul de operare şi îl redă imediat, fără a întreprinde o altă acţiune.

Pentru ca programul să capete consistenţă, corpul funcţiei main trebuie completat cu instrucţiuni. De exemplu, pentru a transmite mesajul “Salut prieteni!” în corpul funcţiei main trebuie inserată cel puţin o instrucţiune, astfel că programul devine:

```
#include <stdio.h>
void main (void)
{
printf (“Salut prieteni !”);
}
```

După editarea fişierului sursă şi apăsarea simultană a tastelor <Ctrl> <F9>, care declanşează acţiunile de compilare, link-editare şi lansare în execuţie a programului, pe ecran nu va apare mesajul transmis. Deoarece, după terminarea execuţiei unui program se revine în fereastra de editare.

Pentru a ne convinge că programul a funcţionat, este necesară vizualizarea ecranului utilizator (se apasă simultan tastele <Alt><F5>). Revenirea în ecranul de editare se face prin apăsarea încă o dată a tastelor <Alt><F5>).

Dacă dorim ca mesajul transmis să fie singur pe ecran şi să se revină în fereastra de editare doar atunci când apăsăm o tastă oarecare, programul trebuie scris astfel:

```
#include <stdio.h>
#include <conio.h>
void main (void)
{
clrscr ();
printf ("Salut prieteni !");
getch ();
}
```

După cum se poate constata, programul conţine trei instrucţiuni care apelează trei funcţii (clrscr, printf şi getch) din bibliotecile de funcţii ale mediului Turbo C/Borland C.

Pentru a marca sfârşitul unei instrucţiuni se utilizează caracterul ";", similar limbajului Pascal.

Limbajul C ignoră spaţiile albe de tipul linie nouă, spaţiu obişnuit sau tabulatorul, cu excepţia cazului în care acestea sunt părţi ale unui şir de caractere încadrat între ghilimele.

Pentru ca programul sursă să poată fi uşor de citit şi urmărit se recomandă ca instrucţiunile să fie scrise pe linii separate, iar acoladele care delimitează un bloc să fie scrise fiecare pe câte o linie în aceeaşi coloană, aşa cum s-a procedat în exemplele prezentate.

1.4. Funcţii de bibliotecă, module antet

Limbajul C este dotat cu o mare varietate de funcţii, numite de bibliotecă, pentru prelucrarea tipurilor de date predefinite în limbaj: tipuri numerice, şiruri de caractere, tablouri, structuri, pointeri (adrese ale unor valori), fişiere. În marea majoritate, aceste funcţii au parametri şi întorc (calculează)un rezultat de un anumit tip. Cum ştie compilatorul limbajului să verifice dacă o astfel de funcţie este utilizată corect, într-un context corect şi dacă valoarea calculată este folosită corect? Soluţia adoptată de C are la bază modularizarea programului. În paralel cu module program, care, în esenţă conţin funcţii şi au un rol activ în cadrul programului, există module antet cu rol declarativ, ele fiind asociate modulelor program. Un

modul antet asociat unui modul program conţine toate informaţiile necesare compilatorului C pentru a putea verifica corectitudinea utilizării funcţiilor sau a altor entităţi de program definite în modulul program, atunci când acestea sunt importate în alte module (antet sau program). Dacă un modul program conţine funcţiile f1, f2, .., fn, funcţii apelabile din alte module, atunci modulul antet va cuprinde declaraţiile acestor funcţii, indicând numărul, tipul parametrilor şi tipul rezultatului. Rezultă că modulele antet au rolul de a oferi posibilitatea de verificare a exportului şi importului corect de entităţi de program între modulele unui program C.

Exemplul tipic în acest sens este cel al funcţiilor de bibliotecă. Acestea sunt grupate în raport cu tipurile de date prelucrate sau în raport cu efectele specifice realizate, fiecărui grup de funcţii (modul) corespunzându-i un modul antet. Exemple de grupuri standard de funcţii de bibliotecă şi numele modulelor antet asociate lor:

- conio.h declară funcţii pentru citirea de la tastatură şi afişarea rezultatelor în limitele unei ferestre
- float.h defineşte valori şi parametri caracteristici tipului float (real)
- limits.h defineşti limitele unor valori întregi
- math.h declară funcţii matematice obişnuite
- stdio.h declară funcţiile standard de intrare/ieşire
- string.h declară funcţii de manipulare a şirurilor de caractere
- time.h declară funcţii pentru manevrarea timpului .

Există un paralelism între funcţiile oferite de conio.h şi stdio.h, datorat faptului că funcţiile standard de intrare-ieşire sunt concepute să lucreze cu întregul ecran şi nu cu ferestre. Deosebirea fundamentală între cele două seturi de funcţii constă în faptul că, spre deosebire de funcţiile standard, funcţiile de citire-scriere definite de fişierul antet conio.h nu pot fi redirectate, ele lucrând în exclusivitate cu consola.

În limbajul C există o convenţie în ceea ce priveşte numele fişierelor sursă ce corespund modulelor program şi antet. Convenţia nu este obligatorie, dar este bine să fie respectată. Numele unui modul program va avea forma **nume.c**, în timp ce numele modulului antet asociat va respecta scrierea **nume.h** (h de la header =antet), unde nume este numele fişierului, iar c, respectiv h sunt extensiile.

Pentru a include textual conţinutul unui modul sursă C în alt modul sursă C (respectiv un modul antet într-un modul program), se utilizează directiva **#include**. Directiva are efect exclusiv pe parcursul compilării programului care le conţine, modificând textul prelucrat de compilator sau

influenţând procesul de compilare. Felul în care este specificat numele modulului de inclus ghidează modul de căutare al fişierului corespunzător modulului. Pentru varianta

#include<nume-fişier>

fişierul sursă nume-fişier este căutat printre fişierele mediului de programare C, în timp ce pentru varianta #include "nume-fişier", fişierul este căutat în zona de lucru a utilizatorului pe suportul de informaţie pe care se lucrează (hard disc sau disc flexibil).

```

# 2
# Tipuri de date

Orice program scris în orice limbaj de programare se compune din date şi instrucţiuni. Vom analiza în continuare principalele tipuri de date utilizate în limbajul C, precum şi aspecte privind modul de introducere, memorare şi afişare a acestora.

## 2.1. Constante şi variabile

Datele de diverse tipuri (întregi, reale, caractere sau şiruri de caractere) folosite în cadrul programelor de calcul sunt clasificate în constante şi variabile. O constantă este un spaţiu din memoria internă a calculatorului destinat memorării unui anumit tip de date al cărei conţinut nu poate fi modificat pe parcursul execuţiei programului de calcul. Spre deosebire de aceasta, o variabilă este tot un spaţiu din memoria internă, destinat memorării unui anumit tip de date al cărei conţinut poate fi modificat pe parcursul execuţiei programului de calcul. Cu alte cuvinte, o variabilă reprezintă un acelaşi spaţiu din memorie care stochează diferite valori de acelaşi tip la momente de timp diferite. Pentru a fi uşor identificate, variabilelor li se asociază un nume şi un tip, care se referă la tipul valorilor pe care le memorează.

Exemplul 2.1.

Tipăreşte constanta întreagă 10 şi constanta reală 3,14159.

```
/* Programul ex_2_1 */
#include <stdio.h>
void main(void)
{
        printf("\nAceasta este constanta întreagă %d", 10);
        printf("\nAceasta este constanta reală %f", 3.14159);
}
```
```

Acelaşi rezultat se obţine mai uşor cu programul din exemplul 2.2

Exemplul 2.2.

```
/* Programul ex_2_2 */
#include <stdio.h>
void main(void)
{
        printf("\nAceasta este constanta întreagă 10");
        printf("\nAceasta este constanta reală 3.14159");
}
```

Nici această variantă nu este avantajoasă, deoarece puterea funcţiei printf provine din capacitatea de a utiliza variabile în locul constantelor.

Rescriem programul, rezultând exemplul 2.3. Programul utilizează variabile în locul constantelor. În prima parte a programului se creează două variabile (una de tip întreg, alta de tip real) care capătă pentru început valorile 10 şi respectiv 3,14159. Acestea vor fi afişate cu ajutorul primelor două apeluri ale funcţiei printf. Apoi, valorile variabilelor sunt modificate, noile valori fiind afişate cu ajutorul celor două apeluri ale funcţiei printf din finalul programului.

Exemplul 2.3.

```
/* Programul ex_2_3 */
#include <stdio.h>
void main(void)
{
        int întreg;
        float real;
        întreg =10;
        real=3.14159;
        clrscr();
        printf("\nVariabila întreg are valoarea  %d", întreg);
        printf("\n Variabila real are valoarea %f", real);
        întreg = întreg +5;
        real = real – 3;
        printf("\n Noua valoare a variabilei întreg este  %d", întreg);
        printf("\n Noua valoare a variabilei real este %f", real);
        getch();
}
```

În primele două instrucţiuni se folosesc cuvintele cheie int şi float, pentru a crea variabila de tip întreg cu numele întreg şi variabila de tip real cu numele real. Efectul execuţiei acestor instrucţiuni este de a rezerva zone de memorie în care se vor stoca valorile atribuite în timpul execuţiei programului acestor variabile şi de a atribui acestor zone numele respective. Pentru definirea numelui variabilelor se poate folosi orice combinaţie de litere şi cifre, cu restricţia ca primul caracter să fie o literă. De obicei, variabilelor li se atribuie nume semnificative, putându-se utiliza unirea a două cuvinte prin utilizarea caracterului "_"(liniuţă de subliniere - underbar). În exemplul nostru, mai semnificativ ar fi fost utilizarea ca nume pentru variabila reală a numelui pi. Următoarele două instrucţiuni atribuie variabilei întreg valoarea 10, iar variabilei real valoarea 3,14159 folosind operatorul de atribuire "=". Valorile sunt apoi afişate cu ajutorul funcţiei printf în care în locul constantelor se utilizează numele variabilelor. În continuare, valorile celor două variabile sunt modificate cu ajutorul operatorilor aritmetici "+", respectiv "-". Noile valori sunt afişate prin apelarea funcţiei printf.

2.2. Tipuri de date

Principalele tipuri de date predefinite în linbajul C sunt tipurile aritmetice întregi şi reale. Aceasta se bazează pe faptul că entităţile de altă natură (caractere, valori de adevăr, culori, etc) pot fi codificate numeric, iar operaţiile specifice lor pot6 fi reprezentate prin prelucrări simple sau compuse efectuate asupra codurilor numerice adoptate. Astfel, pentru reprezentarea valorilor logice se recurge la o convenţie foarte simplă: fals se codifică prin valoarea întreagă 0, iar adevărat printr-o valoare întreagă nenulă (în cazul rezultatului evaluării unei expresii relaţionale sau logice, această valoare este 1).

Limbajul C are o deosebită putere de reprezentare, datorată mecanismelor pe care le oferă pentru declararea de tipuri derivate, pornind de la tipurile fundamentale. Astfel, pot fi declarate:

- Tipuri structurate- tablouri, structuri, uniuni;
- Tipuri funcţie- caracterizat atât prin tipul rezultatului furnizat, cât şi prin numărul şi tipul argumentelor necesare pentru obţinerea rezultatului;
- Tipuri pointer – oferă posibilitatea de adresare indirectă la entităţi de diferite tipuri (inclusiv pointer), deoarece o valoare de tip pointer reprezintă adresa unei entităţi definite în program.

Tipurile menţionate anterior se regăsesc, în marea majoritate, şi în alte limbaje de programare. Ca noutate, în limbajul C este predefinit tipul void, cu semnificaţia "nimic",sau "orice", în funcţie de contextul utilizării. Prezenţa acestui tip predefinit a fost impusă de necesităţi obiective:

- Pentru a face posibil acelaşi mod de declarare, atât pentru funcţiile care întorc un rezultat, cât şi pentru cele care au numai efecte laterale, fără a furniza o valoare rezultat; în cel de al doilea caz, tipul rezultatului se declară void, cu interpretarea "nimic".
- Pentru a declara pointeri către entităţi de tip neprecizat, caz în care tipul void se interpretează ca "orice".

2.2.1. Declaraţii de tip

Tratarea corectă a unei entităţi presupune cunoaşterea tipului său. Dacă, în cazul constantelor explicite, tipul poate fi dedus din modul în care sunt reprezentate acestea, în cazul entităţilor identificate prin nume simbolice (variabile, funcţii, constante simbolice), tipul lor trebuie precizat anterior primei utilizări printr-o declaraţie sau definiţie corespunzătoare.

Declaraţiile de variabile sunt, de cele mai multe ori, definiţii, în sensul că determină şi alocarea spaţiului de memorare necesar pentru păstrarea valorilor variabilelor declarate. Definiţia unei variabile poate fi completată prin specificarea unei valori iniţiale. Momentul efectuării iniţializării, precum şi eventualele iniţializări implicite sunt condiţionate de clasa de memorare în care este inclusă variabila respectivă. Astfel, variabilele declarate **extern** sau **static** se iniţializează o singură dată, înaintea începerii execuţiei programului, cu valoarea declarată, sau, în absenţa acesteia, cu valoarea implicită zero. Variabilele din clasa **automatic** (locale funcţiilor) sunt iniţializate numai dacă s-a cerut explicit, iniţializarea fiind reluată la fiecare apel al funcţiei în care au fost definite.

O declaraţie poate să se refere la una sau mai multe variabile, precizând următoarele elemente cu privire la acestea: clasa de memorare, tipurile, identificatorii şi eventualele valori iniţiale ale variabilelor declarate.

O deosebire esenţială între limbajul C şi alte limbaje de programare este aceea că o declaraţie în C se poate referi la variabile de tipuri diferite, însă derivate din acelaşi tip de bază. De exemplu, declaraţia

```
extern char c, *p, s[10];
```

se referă la trei variabile externe, de tipuri diferite: caracterul *c*, poinerul la caracter *p* şi vectorul de maxim 10 caractere *s*.

Deoarece clasa de memorare poate fi determinată şi implicit, declaraţiile de variabile pot avea una din următoarele forme:

clasa_de_memorare tip_de_bază listă_declaratori;

sau

tip_de_bază listă_declaratori;

La rândul său, lista declaratorilor poate fi formată din unul sau mai multe elemente, separate prin virgule. Un declarator poate să includă sau nu specificarea valorii iniţiale a variabilei definite, deci poate avea una din următoarele forme:

declarator = iniţializare

sau

declarator

unde structura de iniţializare este condiţionată de cea a declaratorului. Cel mai simplu declarator se rezumă la identificatorul entităţii al cărei tip este cel de bază. Declaratorii entităţilor de tipuri derivate se construiesc folosind operatorii de derivare [], () şi *.

Un caz particular este cel al declaraţiilor în care intervine cuvântul rezervat **const**. Valorile entităţilor declarate cu acest atribut pot fi utilizate, dar nu pot fi modificate pe parcursul execuţiei programului. Din acest motiv este necesar ca entităţile cu atributul const să fie iniţializate în momentul definirii. Pe această cale se definesc **constantele simbolice**, ca în exemplul următor:

```
const double pi = 3.14159;
const int maxim = 99;
```

În declararea variabilelor poate să intervină şi atributul volatile. Acesta se utilizează pentru a pune în evidenţă faptul că valoarea variabilei declarate cu acest atribut poate fi alterată de agenţi externi programului, care pot interveni pe parcursul execuţiei acestuia, modificând conţinutul zonei de memorie rezervate pentru variabila respectivă. Cele două atribute, numite şi **calificatori**, pot fi combinate pentru a preciza că programul poate utiliza, dar nu poate modifica valori furnizate de un agent extern (un alt program).

2.2.2. Tipuri aritmetice

Standardul limbajului C fixează o serie de elemente definitorii ale tipurilor aritmetice – identificatorii acestora, operaţiile ce pot fi efectuate asupra valorilor de aceste tipuri, modul de reprezentare a constantelor, descriptorii utilizabili pentru a specifica modul în care se realizează conversia între reprezentarea internă şi cea externă a valorilor în cazul operaţiilor de intrare – ieşire.

Există însă şi elemente care rămân la latitudinea implementatorilor, cum ar fi reprezentarea internă a valorilor de diferite tipuri, cu implicaţii directe asupra domeniului de valori reprezentate, modul de tratare a unor situaţii de eroare, ordinea de evaluare a unor operanzi, etc.

2.2.2.1. Tipuri întregi predefinite

La baza diferenţierii tipurilor întregi predefinite stau două criterii:

- Numărul de unităţi de memorare utilizate în fiecare caz;
- Modalitatea de tratare a valorii reprezentate – ca valoare cu sau fără semn.

Aplicând primul criteriu, întregii pot fi împărţiţi în patru clase:

- char
- short (sau short int)
- int
- long (sau long int).

divizate, la rândul lor, în clase de întregi reprezentaţi cu semn – signed – sau fără semn – unsigned.

Prin convenţie, în cazul întregilor din clasele short, int şi long, se consideră implicită reprezentarea cu semn (deci int este echivalent cu signed int).

Nu acelaşi lucru este valabil pentru tipul char, a cărui reprezentare este identică fie cu cea pentru signed char, fie cu cea pentru unsigned char.
De asemenea, trebuie menţionat faptul că informaţiile cu privire la un tip întreg pot fi specificate în orice ordine. Toate declaraţiile următoare sunt echivalente.

unsigned long int
int unsigned long
long int unsigned
unsigned int long

Chiar dacă numărul de unităţi de memorare utilizate pentru tipurile întregi este dependent de implementare, standardul C impune următoarele condiţii:

a. sizeof(char) <= sizeof(short) <= sizeof(int) <=sizeof(long);
b. char şi variantele sale se reprezintă pe cel puţin 8 biţi
c. short, int şi variantele lor se reprezintă pe cel puţin 16 biţi
d. long şi unsigned long se reprezintă pe cel puţin 32 biţi.

În tabelul următor sunt prezentate particularităţile tipurilor de bază.

Tip	Număr		Valoare	
	Octeţi	Biţi	Minimă	Maximă
char signed char	1	8	-128	127
unsigned char	1	8	0	255
int signed int short int signed short int	2	16	-32768	32767
unsigned int unsigned short int	2	16	0	65535
long int signed long int	4	32	-2147483648	2147483647
unsigned long int	4	32	0	4294967295

Reprezentarea constantelor întregi

În mod obişnuit, constantele întregi se reprezintă în baza 10, printr-o succesiune de una sau mai multe cifre. Există însă şi situaţii în care este preferabilă reprezentarea în octal sau hexazecimal (baza 8 sau 16).În aceste cazuri, reprezentarea constantei va începe printr-un prefix adecvat: 0 (cifra zero) pentru octal şi, respectiv, 0x sau 0X pentru hexazecimal.

Valoarea efectivă a constantei este determinată de prefix şi de cifrele utilizate (de la 0 la 7 în octal şi 0,..,9,A,..,F în hexazecimal.

Astfel, constantele

22 022 0x22

nu reprezintă aceeaşi valoare întreagă, deoarece 022 este o constantă octală a cărei valoare este 2*8 + 2 = 18, în timp ce 0x22 este constantă hexazecimală având valoarea 2*16 + 2 = 34.

Tipul unei constante întregi poate fi int, unsigned int, long sau unsigned long. De regulă, el este stabilit automat, în funcţie de numărul de biţi necesari pentru reprezentarea internă a valorii constantei respective. Programatorul are însă posibilitatea de a specifica, prin intermediul unui sufix, opţiunile proprii cu privire la tipul constantei. Sufixul poate fi format din una sau două litere, corespunzând următoarelor opţiuni:

- reprezentare fără semn (unsigned) – u sau U
- reprezentare de tip long – l sau L.

În cazurile în care există mai multe posibilităţi de stabilire a tipului unei constante, este selectată prima variantă care permite reprezentarea corectă a valorii constantei respective.

Baza	Sufix	Tipuri posibile
10, 8, 16	UL	unsigned long
	L	long, unsigned long
	U	unsigned int, unsigned long
10	fără	int, long, unsigned long
8, 16		int, unsigned int, long, unsigned long

2.2.2.2. Tipuri enumerare

Tipurile enumerare constituie cazuri particulare ale tipurilor întregi. Ele se utilizează pentru a realiza o reprezentare comodă şi sugestivă a tipurilor de obiecte ale căror valori sunt identificate printr-un număr finit de nume simbolice. De exemplu, unitatea de măsură pentru un obiect poate fi metrul, litrul, gramul, bucata; mijlocul de transport poate fi avionul, trenul, autocarul, autoturismul. Valorile constante identificate prin nume simbolice specifice problemei trebuie codificate prin entităţi specifice limbajului de programare, deci prin constante numerice. Prin utilizarea directă a codificării numerice se pierde însă din claritatea asigurată de folosirea numelor simbolice şi creşte riscul de utilizare incorectă a unor coduri. Soluţia acestei probleme este oferită de tipurile **enumerare** care declară **constante simbolice** cărora li se asociază coduri numerice de tip întreg.

Ca efect al unei declaraţii de forma

enum nume-tip {lista-nume-constante};

compilatorul asociază fiecărei constante enumerate un cod întreg, începând de la 0. Astfel, ca efect al declaraţiei

enum unitate {litru, metro, gram, bucată};

constantelor tipului enumerare **unitate** li se asociază valori între 0 (pentru litru) şi 3 (pentru bucată). Reprezentarea efectivă a acestor valori este dependentă de implementare (ea poate fi de tip int, sau un alt tip întreg mai "slab").

În cazurile în care programatorul doreşte să impună o anumită codificare numerică a constantelor de tip enumerare, diferită de cea stabilită implicit de către compilator, atunci în lista numelor constantelor pot fi folosite elemente de forma:

nume-constantă = valoare

ca în cazul următor:

enum transport {tren, avion =5, autocar, autoturism};

Ca efect al acestei declaraţii, constantelor autocar şi autoturism li se asociază codurile 6 şi respectiv 7, deoarece codificarea se continuă întotdeauna secvenţial, în ordinea enumerării.

2.2.2.3. Tipuri reale

Standardul ANSI (American National Standards Institute) prevede trei tipuri reale de bază: **float**, **double**, **long double**. Majoritatea implementărilor nu fac distincţie între **double** şi **long double**. În tabelul următor sunt prezentate caracteristicile acestor tipuri.

Tip	Număr		Valoare absolută	
	Octeţi	Biţi	Minimă	Maximă
Float	4	32	3.4E-8	3.4E+38
Double long double	8	64	1.7E-308	1.7E+308

În ceea ce priveşte precizia reprezentării, conform standardului, aceste trebuie să fie de cel puţin 6 cifre semnificative în cazul valorilor de tip **float** şi de cel puţin 10 cifre semnificative în cazul valorilor de tip **double** şi **long double**.

În cazul evaluării expresiilor, toţi operanzii de tip **float** sunt convertiţi automat la **double**.

Constantele de tip real se reprezintă de cele mai multe ori folosind notaţia uzuală

parte-întreagă.parte-subunitară

(din care poate lipsi fie partea întreagă, fie partea subunitară, dar în nici un caz ambele). În cazul valorilor reale foarte mari sau foarte mici, se preferă notaţia cu mantisă şi exponent. În această notaţie, mantisa este reprezentată de o constantă întreagă sau reală uzuală, fiind urmată de litera E (sau e) şi valoarea exponentului (cu sau fără semn). Valoarea unei astfel de constante este dată de formula:

$$\text{mantisa} * 10^{\text{exponent}}$$

De exemplu, numărul 54321 se scrie sub forma 5.4321e4, iar 0.00321 este reprezentat exponenţial sub forma 3.21e-3. Utilizând exprimarea exponenţială, în variabilele reale pot fi stocate valori mult mai mari sau mult mai mici decât în variabilele întregi. Totuşi, există dezavantajul că operaţiile cu variabile reale se desfăşoară mai încet decât

cele cu variabile întregi. Acesta este motivul pentru care algoritmii de calcul cu valori întregi sunt mai rapizi decât cei cu valori reale.

O declaraţie de tipul **float fval;** declară variabila fval de tipul real, rezervând patru octeţi de memorie pentru memorarea valorilor. Trei dintre aceştia sunt utilizaţi pentru memorarea mantisei, cu o precizie de 7 cifre, iar unul pentru memorarea exponentului.

O declaraţie de tipul double dval; declară variabila dval ca fiind de tipul double, rezervând opt octeţi de memorie pentru memorarea valorilor. Ca efect al creşterii numărului de octeţi rezervaţi, creşte domeniul valorilor memorate (de la 10^{-308} la 10^{308}), precum şi precizia de reprezentare exponenţială la 15 cifre exacte.

2.3. Adresele variabilelor

Memoria internă a calculatorului este organizată pe celule numite octeţi, care sunt numerotate de la 0 până la capacitatea maximă a memoriei. Aceste numere ataşate octeţilor poartă numele de adrese. La declararea variabilelor, pe lângă faptul că se precizează numele şi tipul, acestora li se rezervă şi un spaţiu de memorie (un număr de octeţi care variază funcţie de tip), deci li se ataşează o adresă. Pentru variabilele de tip caracter, adresa este chiar adresa octetului, în timp ce pentru celelalte tipuri, adresa este adresa primului octet din cei doi, patru sau opt octeţi alocaţi.

Din cele prezentate până acum putem concluziona că o variabilă are:

- o valoare care poate fi modificată în timpul execuţiei programului,
- un tip care determină mărimea zonei de memorare rezervată pentru memorarea valorilor,
- o adresă care reprezintă locul în memoria internă unde urmează să fie memorate valorile.

În limbajul C toate cele trei elemente asociate unei variabile sunt accesibile programatorului. Astfel, valoarea variabilei este furnizată de numele acesteia, adresa se obţine folosind în faţa numelui caracterul "**&**", numit operator de adresă, iar numărul de octeţi se obţine prin apelul funcţiei **sizeof**.

```

# 3
# Citirea şi scrierea datelor

În acest capitol vom analiza operaţiile de afişare a rezultatelor pe monitorul calculatorului şi cele de introducere a datelor de la tastatură.

## 3.1. Afişarea datelor – funcţia printf

În limbajul C funcţia **printf** este funcţia standard prin intermediul căreia rezultatele obţinute în urma procesului de calcul sunt afişate pe ecranul calculatorului.

Forma generală de apel:

printf(constantă şir de caractere, listă de valori);

O constantă şir de caractere este o succesiune de litere, cifre, spaţii albe şi caractere speciale delimitate de o pereche de ghilimele. Primul parametru al funcţiei printf este o construcţie de acest tip şi conţine textul ce va fi afişat pe ecran, precum şi descriptorii de format. Fiecare descriptor de format poate fi plasat oriunde în şirul de caractere şi constă din caracterul % urmat de una sau două litere. Rolul descriptorilor de format este de a specifica conversia de efectuat asupra valorilor preluate din memoria internă (unde se găsesc memorate sub formă binară) înainte ca acestea să fie inserate în textul afişat pe ecran, precum şi de a marca locul în care acestea vor fi inserate în text. Se spune că, după conversie valorile se substituie descriptorilor de format în şirul de caractere care va fi afişat pe ecran. Acest mecanism impune ca între descriptorii de format şi lista de valori să existe o corespondenţă biunivocă, adică fiecărei valori să-i corespundă un descriptor de format.

Tabelul următor prezintă descriptorii de format care pot fi utilizaţi în apelul funcţiei **printf** pentru afişarea diferitelor tipuri de valori.
```

Descriptor de format	Tipul de date pentru care se foloseşte
%c	Caracter
%s	Şir de caractere
%d	Întreg zecimal cu semn
%f	Real (notaţie zecimală)
%e	Real (notaţie exponenţială)
%g	Real (selectează notaţia zecimală sau exponenţială care este mai scurtă)
%u	Întreg zecimal fără semn
%x	Întreg hexazecimal fără semn
%o	Întreg octal fără semn
%p	Adresa hexazecimală

Pentru a explicita modul de lucru al funcţiei printf, analizăm instrucţiunea următoare:

printf("Aceasta este constanta %d",10);

Mai întâi, valoarea constantei 10 este convertită conform descriptorului de format specific valorilor de tip întreg, valoarea obţinută este inserată în text în locul descriptorului de format şi apoi textul este afişat pe ecran.

Textul afişat pe ecran: Aceasta este constanta 10

Funcţia printf oferă programatorului posibilitatea de a controla afişarea pe ecran. Pentru valorile de tip întreg sau real, în descriptorul de format, după caracterul % se poate specifica mărimea câmpului (numărul de celule de pe ecran rezervate pentru afişarea valorii), precum şi numărul de zecimale în cazul valorilor reale.

Exemplul 3.1.

```
/* Programul 3.1.*/
#include <stdio.h>
void main (void);
{
        int i;
        float fval=5.025;
        for (i = 1; i <= 10; i++)
                printf("\n%6d\t%12.3f", i, fval + (i – 1) * 1.5);
}
```

Programul afişează pe ecran un tabel format din două coloane. În prima afişează numărul curent având o lăţime de şase spaţii, iar în a doua valori reale care vor fi afişate pe o lăţime de 12 spaţii, din care trei sunt rezervate zecimalelor.

Despre instrucţiunea de ciclare **for** vom discuta mai târziu. Analizăm funcţia printf. Pentru a construi textul care se va afişa pe ecran, funcţia printf substituie valoarea variabilei i descriptorului %6d, apoi evaluează valoarea expresiei fval + (i – 1) * 1.5 şi o substituie descriptorului %12.3f .

În urma execuţiei acestui program, rezultă următorul tabel:

```
1       5.025
2       6.525
3       8.025
.
.
10      17.025
```

În şirul de caractere care constituie primul parametru al funcţiei printf, pe lângă descriptorii de format se pot utiliza comenzi de control, numite **secvenţe escape**. Pentru acestea se utilizează simbolul \ numit backslash, care determină o semnificaţie diferită de cea obişnuită a caracterelor care îi urmează.

Secvenţele escape utilizate în limbajul C sunt prezentate în tabelul următor:

Secvenţa	Semnificaţia
\n	Salt la o linie nouă (new line)
\t	Deplasare la dreapta (tab)
\b	Deplasare la stânga cu o poziţie (backspace)
\r	Întoarcere la cap de rând (carriage return)
\f	Trecere pe linia următoare (formfeed)
\’	Apostrof
\”	Ghilimele
\\	Backslash
\xdd	Codul ASCII în hexazecimal (fiecare d reprezintă o cifră hexazecimală)

Utilizarea secvenţei escape \xdd în cadrul funcţiei printf este o modalitate de a crea imagini grafice folosind codurile hexazecimale din extensia IBM. În programul următor este prezentat modul cum poate fi trasat un dreptunghi pe ecran.

Exemplul 3.2.

```
/* programul 3.2. */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        clrscr ();
        gotoxy (20,5);
        printf("\xC9\xCD\xCD\xCD\xCD\xCD\xBB");
        gotoxy(20,6);
        printf("\xC8\xCD\xCD\xCD\xCD\xCD\xBC");
}
```

Prin execuţia programului se obţine pe ecran o imagine de forma:

3.2. Citirea datelor

Funcţia scanf este perechea fucţiei printf în cadrul operaţiilor de intrare – ieşire, permiţând introducerea datelor de la tastatură. Pentru exemplificare, considerăm programul următor în care se citeşte de la tastatură o valoare reală care exprimă temperatura în grade Celsius şi afişează apoi acea temperatură exprimată în Kelvin.

Exemplul 3.3.

```
/*Program ex_3_3*/
#include <stdio.h>
void main (void)
{
        float temp;
        printf("\n Introduceti temperatura în grade Celsius");
        scanf("%f", &temp);
        printf("\n Temperatura în Kelvin este  %f", temp+273.15);
}
```

Se observă că în primul apel al funcţiei printf nu s-au folosit descriptorii de format şi nici lista de variabile. Efectul unui astfel de apel constă în afişarea şirului de caractere pe ecran şi constituie modalitatea prin care programul transmite mesaje utilizatorului.

Analizând funcţia scanf, constatăm că este foarte asemănătoare cu printf. Primul parametru este tot o constantă şir de caractere care reprezintă descriptorul de format. Deosebirea esenţială între cele două funcţii constă în aceea că, spre deosebire de printf care utilizează numele variabilelor (conţinutul acestora), scanf foloseşte adresa acestora. Acest lucru este normal dacă înţelegem că scanf primeşte de la tastatură valoarea introdusă conform descriptorului de format şi o depune în zona de memorie rezervată variabilei.

Funcţia scanf poate citi într-un singur apel, a cărui formă generală este

scanf (constantă şir de caractere, lista adreselor variabilelor);

mai multe valori pentru variabile de diverse tipuri. Exemplificăm cu următorul program, care citeşte trei valori (un caracter, un întreg şi un real) printr-un singur apel al funcţiei scanf.

Exemplul 3.4.

```
/*Program 3.4. */
#include <stdio.h>
void main (void)
{
        char car;
        int ival;
        float fval;
        printf("\n Introduceţi un caracter, un întreg şi un real");
        scanf ("%c %d %e", &car, &ival, &fval);
        printf("\n Valorile introduse sunt: %c\t%d\t%f", car, ival, fval);
}
```

La introducerea datelor de la tastatură, separarea acestora se va face plasând un spaţiu între ele. Spaţiile constituie modalitatea prin care scanf detectează sfârşitul unei valori şi respectiv începutul celei următoare. Ca spaţiu de separare la introducerea datelor se poate folosi spaţiul obişnuit (blank), tasta <TAB>, sau tasta <ENTER>. Astfel, dacă la rularea programului 3.4. se doreşte introducerea valorilor A, 15 şi 13.14, acestea pot fi tastate astfel:

A 15 13.14 <ENTER> sau
A <TAB> 15 <TAB> 13.14 <ENTER> sau
A <ENTER> 15 <ENTER> 13.14 <ENTER>

Se observă că, indiferent de varianta aleasă, ultima acţiune constă în apăsarea tastei <ENTER>.

Descriptorii de format ai funcţiei scanf sunt practic aceiaşi cu cei folosiţi de funcţia printf.

Există două funcţii specializate în citirea caracterelor.

La tastarea variabilelor, funcţia scanf le preia şi în acelaşi timp le afişează pe ecran. Se spune că acţionarea tastelor se face cu ecou. Există situaţii în care se doreşte citirea unui caracter de la tastatură fără a se face ecoul tastei apăsate. Această operaţie se realizează prin apelarea funcţiei getch. De exemplu, într-o instrucţiune de forma car = getch(); variabila de tip caracter car primeşte codul tastei apăsate fără ca pe ecran să apară caracterul asociat acesteia.

Dacă se doreşte citirea unui caracter cu ecou, atunci se va utiliza funcţia getche. Aceasta este similară funcţiei getch, cu deosebirea că afişează caracterul asociat tastei apăsate.

```

# 4
# Operatori

Scopul oricărui program de calcul este de a prelucra datele stocate în variabile, în conformitate cu un algoritm de calcul, pentru a obţine rezultate. Variabilele, deşi constituie o parte importantă a uni limbaj, fără un set de operatori şi un set de instrucţiuni care să stabilească ce operaţii se efectuează asupra lor şi cum se fac aceste operaţii, nu au nici o importanţă.

Operatorii, ca şi în algebră, reprezintă simboluri sau cuvinte care determină într-un program efectuarea unor operaţii cu variabile.

## 4.1. Operatorul de conversie explicită (cast)

Dacă operanzii unei atribuiri nu sunt de acelaşi tip, atunci valoarea atribuită trebuie convertită la tipul variabilei căreia I se atribuie. Astfel de conversii pot conduce la depăşiri, sau la pierderi de precizie, în multe cazuri rezultatul conversiei fiind dependent de implementare. De exemplu, atribuirea unei valori reale unei variabile de tip întreg conduce la trunchierea părţii subunitare şi chiar la depăşire dacă partea întreagă a valorii reale nu este reprezentabilă ca valoare de tipul întreg respectiv.

Conversia poate fi impusă de către programator prin utilizarea operatorului de conversie explicită de tip, denumit cast. Acest operator prefixat, care se utilizează sub forma

(declaraţie-tip) expresie

determină conversia valorii expresiei la tipul declaraţie-tip.

Conversiile explicite sunt utile în cazurile în care se doreşte ca rezultatul unei operaţii să fie de alt tip decât cel determinat implicit, pe baza tipurilor operanzilor. De exemplu, dacă se doreşte obţinerea rezultatului real, netrunchiat, al împărţirii a două valori de tip **int**, atunci cel puţin unul dintre operanzi trebuie convertit explicit la tipul **double**, ca în expresia

(double) a / b.
```

4.2. Operatorul sizeof

Rezultatul aplicării acestui operator asupra operandului său este reprezentat de un întreg specificând dimensiunea spaţiului de memorie (de regulă în octeţi) necesar pentru stocarea oricărei valori de acelaşi tip cu cel al operandului. Prin urmare, operatorul **sizeof** prelucrează tipuri, spre deosebire de ceilalţi operatori care prelucrează valori. Operatorul **sizeof** poate fi aplicat direct asupra unui tip, sub forma

sizeof (declaraţie-tip)

sau asupra tipului unei expresii, sub forma

sizeof expresie

Deoarece tipul operanzilor este determinat încă din etapa de compilare, **sizeof** este un operator cu efect la compilare. Aceasta înseamnă că operandul asupra căruia se aplică **sizeof** nu este evaluat, chiar dacă este reprezentat de o expresie.

Programul din exemplul 4.1. poate fi utilizat pentru a afla particularităţile de reprezentare pentru câteva dintre tipurile predefinite în C.

Exemplul 4.1.

```
/*Program 4.1.*/
#include <stdio.h>
void main ()
{
     printf(" char short int long float double\n%3d%6d%6d%6d%6d%6d\n",
          sizeof(char), sizeof(short), sizeof(int), sizeof(long), sizeof(float),
          sizeof(double));
}
```

4.3. Operatori aritmetici

Operaţiile aritmetice obişnuite sunt codificate în limbajul C prin operatorii:

+ adunare,
- scădere,
* înmulţire,
/ împărţire,
% restul împărţirii a doi întregi.

Considerăm programul de calcul din exemplul 3.3. care realizează transformarea în grade Celsius şi Kelvin a unei temperaturi exprimate în grade Fahrenheit.

Exemplul 4.2.

```
/*Program ex_4_2*/
#include <stdio.h>
void main (void)
{
        float ftemp, ctemp, ktemp;
        printf("\n Introduceti temperatura în grade Fahrenheit >");
        scanf("%f", &ftemp);
        ctemp=(ftemp-32)*5/9;
        ktemp=ctemp+273.15;
        printf("\n Temperatura în grade Celsius %f\nTemperatura în Kelvin
        %f", ctemp,ktemp);
}
```

Referitor la modul în care limbajul C evaluează expresiile în care intervin operatorii aritmetici, sunt necesare următoarele precizări:

- ca şi în algebră, în expresiile în care nu intervin paranteze mai întâi se efectuează operaţiile de înmulţire şi împărţire, apoi cele de adunare şi scădere;
- în expresiile în care intervin paranteze, acestea vor fi evaluate începând cu perechea cea mai interioară şi terminând cu perechea exterioară.

În exemplul prezentat, în calculul temperaturii în grade Celsius

```
ctemp=(ftemp-32)*5/9;
```

întâi se calculează expresia din paranteză, apoi rezultatul este înmulţit cu 5 şi apoi împărţit la 9.

4.4. Operatori de atribuire şi de atribuire aritmetică

Dacă se face o comparaţie între un program scris în C şi un altul (rezolvând aceeaşi problemă) scris în alt limbaj, se va constata că programul în C este mai scurt. O explicaţie este faptul că limbajul C utilizează o serie de operatori care comprimă instrucţiunile.

Considerând următoarea instrucţiune:

```
a=a+b;
```

în care valoarea lui *b* este adunată cu valoarea lui *a* şi rezultatul este atribuit variabilei *a* prin intermediul operatorului de atribuire "=", această instrucţiune poate fi scrisă

a+=b;

efectul fiind acelaşi, dar forma fiind mai compactă.

Toţi operatorii aritmetici pot fi combinaţi cu semnul egal în acelaşi mod, obţinându-se operatorii de atribuire aritmetică prezentaţi în tabelul 4.1.

Tabel 4.1. Operatori de atribuire aritmetică

Operator	Scop	Exemplu	Forma echivalentă în alte limbaje
=	Atribuire simplă	a=15	a : = 15
+ =	Adunare cu atribuire	a + = b	a : =a + b
- =	Scădere cu atribuire	a - = b	a : =a - b
* =	Înmulţire cu atribuire	a * = b	a : =a * b
/ =	Împărţire cu atribuire	a / = b	a : =a / b
% =	Atribuire modulo n	a % = b	a : =a mod b

O altă modalitate de a compacta codul este aceea de a utiliza expresiile în locul variabilelor. Acest lucru se poate constata din exemplul următor, în care s-a rescris, într-o formă compactă, programul de conversie a temperaturii.

Exemplul 4.3.

```
/* Program ex_4_3 */
#include <stdio.h>
void main (void)
{
        float ftemp, ktemp=273.15;
        printf("\n Introduceti temperatura în grade Fahrenheit >");
        scanf("%e", &ftemp);
        ktemp+=(ftemp-32)*5/9;
        printf("\n Temperatura în grade Celsius %g \nTemperatura în
                Kelvin %g", (ftemp-32)*5/9 ,ktemp);
}
```

Se observă că în noua versiune de program sunt utilizate doar două variabile, folosind în locul variabilei *ctemp* din apelul funcţiei printf expresia (ftemp-32)*5/9, iar pentru evaluarea temperaturii în Kelvin operatorul de atribuire aritmetică + =.

4.5. Operatori de incrementare şi decrementare

Operatorul de incrementare, simbolizat ++ şi operatorul de decrementare, simbolizat -- , sunt specifici limbajului C şi au ca efect creşterea cu 1, respectiv micşorarea cu 1, a valorii variabilei căreia i se aplică.

Pentru exemplificare, fie programul următor:

Exemplul 4.4.

```
/* Program ex_4_4 */
#include <stdio.h>
void main (void)
{
        int i = 0, j =0;
        printf("\ni=%4d\tj=%4d",i,j);
        printf("\ni=%4d\tj=%4d",i++,--j);
        printf("\ni=%4d\tj=%4d",i,j);
}
```

Dacă se rulează acest program, se vor obţine pe ecran următoarele rezultate:

```
i = 0    j =0
i = 0    j =-1
i = 1    j =-1
```

Primul apel al funcţiei printf afişează valorile iniţiale ale lui i şi j. În al doilea apel al funcţiei sunt folosiţi operatorii de incrementare, respectiv decrementare. Din rezultatul obţinut observăm că lui *i* nu i s-a modificat (încă) valoarea, în timp ce lui *j* i s-a modificat, ajungând la –1. Rezultatul explică modul în care acţionează operatorii ++ şi --:

- dacă operatorul este utilizat în faţa numelui variabilei, atunci mai întâi se modifică valoarea variabilei şi apoi este folosită noua valoare. Acest mod de utilizare poartă numele de **autoincrementare**, respectiv **autodecrementare**.
- Dacă operatorul este folosit după numele variabilei, atunci mai întâi este folosită vechea valoare, după care valoarea este modificată. Acest mod de utilizare se numeşte **postincrementar**e, respectiv **postdecrementare**.

Operatorii de incrementare şi de decrementare contribuie la compactarea programului. Dacă nu am folosi aceşti operatori, programul din exemplul 4.5. ar fi:

Exemplul 4.5.

```
/* Program ex_4_5 */
#include <stdio.h>
void main (void)
{
        int i = 0, j =0;
        printf("\ni=%4d\tj=%4d",i,j);
        j-=1;
        printf("\ni=%4d\tj=%4d",i,j);
        i+=1;
        printf("\ni=%4d\tj=%4d",i,j);
}
```

4.6. Operatori relaţionali

Pentru a putea lua decizii într-un program, apare necesitatea de a pune întrebări despre relaţia care există la u moment dat între variabile.

Operatorii relaţionali sunt simboluri cu ajutorul cărora construim întrebări privind relaţiile existente între variabile. Acestea formează **expresii relaţionale**, care pot fi **adevărate** sau **false**.

Limbajul C foloseşte şase operatori relaţionali, prezentaţi în tabelul 4.2.

Tabelul 4.2. Operatori relaţionali

Operator	Semnificaţie
<	Strict mai mic
<=	Mai mic sau egal
>	Strict mai mare
>=	Mai mare sau egal
= =	Egal cu
!=	Diferit de

Într-o expresie relaţională pot fi folosiţi, pe lângă operatorii relaţionali şi operatori aritmetici, ca în exemplul următor:

Exemplul 4.6.

```
/* Program ex_4_6 */
#include <stdio.h>
void main (void)
{
        int val;
        printf("\n Introduceti o valoare întreagă >");
        scanf("%d", &val);
        printf("\n Este %d egal cu 5+7 ? %d", val, val= =5+7);
}
```

Dacă rulaţi acest program şi tastaţi pentru *val* numărul 12, pe ecran apare textul **Este 12 egal cu 5+7 ? 1**, iar dacă introduceţi o valoare diferită de 12, rezultatul pe ecran va fi: **Este … egal cu 5+7 ? 0**. În exemplul nostru mai întâi este evaluată expresia 5+7, apoi valoarea obţinută este comparată cu valoarea variabilei *val*, rezultând un răspuns 1 sau 0, funcţie de valoarea de adevăr a propoziţiei (adevărat, respectiv fals).

Atenţie: Una din greşelile cele mai frecvente şi nu întotdeauna uşor observabile constă în folosirea operatorului de atribuire = în locul celui de egalitate = =. Această greşeală poate avea consecinţe negative datorită efectelor laterale pe care le cauzează. Astfel, expresia *val= =5+7* va avea valoarea 0 sau 1, total diferită de situaţia *val = 5+7*, în care variabilei val i se atribuie valoarea 12.

4.7. Operatori logici

Pentru luarea unei decizii este necesar să combinăm mai multe expresii relaţionale, adică să efectuăm operaţii cu valorile adevărat sau fals, asociate cu valorile întregi 1 şi respectiv 0.

Aceste operaţii sunt operaţiile din logica matematică: sau logic (OR), şi logic (AND), respectiv negaţia logică (NOT). Aceşti operatori logici sunt utilizaţi, împreună cu operatorii relaţionali, pentru construirea expresiilor logice necesare în luarea deciziilor. În tabelul 4.3. sunt prezentaţi operatorii logici.

Tabelul 4.3. Operatorii logici

Operator	Semnificaţie
||	Sau logic (OR)
&&	Şi logic (AND)
!	Negaţie logică (NOT)

Pentru a înţelege modul în care se utilizează operatorii logici împreună cu operatorii relaţionali, considerăm exemplul următor în care se solicită apăsarea unei taste şi se răspunde la întrebarea dacă tasta apăsată este o cifră, sau un alt caracter.

Exemplul 4.7.

```
/* Program ex_4_7 */
#include <stdio.h>
void main (void)
{
        char ch;
        printf("\n Apăsaţi o tastă >");
        ch = getche();
        if(ch>=48 && ch<=57)
                        printf(" Este o cifră");
                else
                        printf(" Nu este o cifră");

}
```

```

# 5
# Instrucţiuni de ciclare (iterative, bucle)

În rezolvarea unor probleme concrete apare necesitatea ca o instrucţiune sau un grup de instrucţiuni să fie executate în mod repetat. Adică în mod ciclic sau iterativ, până când este îndeplinită o anumită condiţie. Mecanismul care permite acest lucru se numeşte ciclu, iteraţie, buclă.

În funcţie de locul în care se efectuează testul de părăsire a unui ciclu, există trei tipuri de cicluri:

1. ciclul cu contor, care este utilizabil atunci când se cunoaşte numărul de iteraţii care trebuie efectuat. În acest caz, pentru a controla bucla, se utilizează o variabilă de tipul întreg, numită contor, a cărei valoare este comparată în permanenţă cu numărul de iteraţii ce trebuie efectuate pentru a decide dacă este necesară repetarea secvenţei de instrucţiuni ce defineşte corpul buclei sau părăsirea acesteia.
2. ciclu cu test iniţial, care este utilizat atunci când nu se cunoaşte numărul de iteraţii ce trebuie efectuate, iar verificarea condiţiei de parcurgere a buclei este necesar a se efectua înaintea efectuării setului de instrucţiuni ce definesc corpul său. În cazul în care condiţia de părăsire a buclei este îndeplinită încă de la ănceput, nu se va efectua nici o iteraţie.
3. ciclu cu test final, atunci când nu se cunoaşte numărul de iteraţii ce trebuie efectuate, iar verificarea condiţiei de buclare se realizează după efectuarea setului de instrucţiuni care formează corpul buclei. În acest caz, dacă condiţia de părăsire a buclei este îndeplinită de la început, se efectuează totuşi o iteraţie.
```

5.1. Bucla for

Bucla for este un mecanism prin care se pot programa ciclurile cu test iniţial cu sau fără contor.

Formatul general al instrucţiunii este:

```
for(exp1; exp2; exp3)
    instrucţiune;
```

unde:

exp1 – reprezintă iniţializarea buclei, respectiv atribuirea unei valori de pornire variabilei contor

exp2 – reprezintă testul de sfârşit al buclei

exp3 – reprezintă actualizarea variabilei contor

Acest format este echivalent cu:

```
exp1;
while(exp2)
{
    instrucţiune;
    exp3;
}
```

Pentru exemplificare, considerăm exemplul următor, care calculează factorialul unui număr întreg.

Exemplul 5.1.

```
/* Program ex_5_1 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
    int n, con;
    long fact=1;
    printf("\n Introduceţi un număr natural >");
    scanf("%d",&n);
    for(con=1; con<=n; con++)
        fact*=con;      // similar cu fact=fact*con;
    printf("\n%d!=%ld",n,fact);
    getch();
}
```

Oricare din cele trei expresii poate lipsi, dar caracterele ; trebuie să apară. Dacă lipseşte **exp1** este necesară iniţializarea separată a variabilei contor. Exemplul anterior devine:

Exemplul 5.2.

```
/* Program ex_5_2 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int n, con=1;
        long fact=1;
        printf("\n Introduceţi un număr natural >");
        scanf("%d",&n);
        for(; con<=n; con++)
                fact*=con;      // similar cu fact=fact*con;
        printf("\n%d!=%ld",n,fact);
        getch();
}
```

Utilizând operatorul virgulă, în fiecare expresie pot apare expresii multiple. Pentru a exemplifica modul de utilizare a mai multor expresii în cadrul expresiei de iniţializare, considerăm programul din exemplul 5.3. în care se calculează suma primelor n numere naturale

Exemplul 5.3.

```
/* Program ex_5_3 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int n, i, suma;
        printf("\n Introduceţi un număr natural >");
        scanf("%d",&n);
        for(i=1,suma=0; i<=n; i++)
                suma+=i;        // similar cu suma=suma+i;
        printf("\Suma primelor %d numere naturale este %d",n,suma);
        getch();
}
```

Dacă lipseşte **exp2**, se presupune implicit că valoarea ei este 1, astfel încât bucla nu se va termina niciodată (buclă infinită).

Dacă rescriem programul 5.3. din care eliminăm toate cele trei expresii ale instrucţiunii for, se obţine o buclă infinită care încearcă să calculeze suma tuturor numerelor naturale.

Exemplul 5.4.

```
/* Program ex_5_4 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int  i=1, suma=0;
        clrscr();
        for( ; ; )
                suma+=i++;
        printf("\Suma primelor %d numere naturale este %d",i,suma);
}
```

Întrucât lipseşte şi expresia de modificare a valorii contorului, aceasta trebuie realizată printr-o instrucţiune plasată în interiorul buclei. În exemplul prezentat, modificarea variabilei *i* s-a realizat prin operaţia de postincrementare din instrucţiunea

```
suma +=i++;
```

echivalentă cu următorul set de instrucţiuni:

```
suma=suma+i;
i=i+1;
```

Corpul buclei poate fi format dintr-o instrucţiune, sau dintr-un set de instrucţiuni. Dacă este format dintr-un set de instrucţiuni, pentru delimitarea acestuia se folosesc acoladele. Setul de instrucţiuni inclus între acolade poartă denumirea de **instrucţiune compusă** sau **bloc de instrucţiuni**. Deoarece compilatorul C tratează ca fiind o instrucţiune ansamblul format din cuvântul cheie for, cele trei expresii cuprinse între parantezele rotunde care îl urmează şi corpul buclei, este posibilă înlănţuirea buclelor for, adică utilizarea în corpul unei bucle for a altei bucle for. Pentru a exemplifica utilizarea instrucţiunii compuse şi a buclelor for înlănţuite, considerăm programul din exemplul 5.5. care generează tabla înmulţirii.

Exemplul 5.5.

```
/* Program ex_5_5 */
#include <stdio.h>
```

```
#include <conio.h>
void main (void)
{
        int  i, j;
        clrscr();
        for( i=1;i<=9 ;i++ )
        {
                for( j=1;j<=9 ;j++ )
                printf("%4d",i*j);
                printf("\n");
        }
        getch();
}
```

Pentru fiecare valoare a lui *i* de la 1 la 9 se execută bucla interioară care, prin apelul repetat al funcţiei printf afişează produsul lui i cu 1, 2, 3, .., 9, adică tabla înmulţirii cu *i*. După terminarea buclei interioare, care conţine în interiorul ei o singură instrucţiune, se pregăteşte afişarea unui nou rând prin instrucţiunea **printf("\n");** care face parte din corpul primei bucle.

Aşa cum am precizat la începutul acestui paragraf, instrucţiunea for poate fi utilizată şi pentru programarea ciclurilor cu un număr necunoscut de iteraţii. Pentru a exemplifica acest mod de utilizare, considerăm programul din exemplul 5.6.

Exemplul 5.6.

```
/* Program ex_5_6 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int  ncar = 0;
        clrscr();
        printf(„Tastaţi un şir de caractere>");
        for( ; getche()!=13 ; )
                ncar++;
        printf("\nNumărul caracterelor tastate este %d", ncar);
        getch();
}
```

Programul îi solicită utilizatorului să tasteze un şir de caractere. Caracterele tastate, în număr necunoscut, sunt citite cu ajutorul funcţiei

getche, în cadrul buclei for, până când s-a apăsat tasta <ENTER> (care are codul 13). Caracterele introduse sunt contorizate cu ajutorul variabilei ncar.

Programul poate fi rescris astfel încât iniţializarea şi modificarea variabilei *ncar* să se realizeze în cadrul expresiilor buclei for. Variabila *ncar* nu va fi contor al ciclului, deoarece valoarea sa nu este folosită pentru părăsirea buclei.

Exemplul 5.7.

```
/* Program ex_5_7 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int  ncar;
        clrscr();
        printf(„Tastaţi un şir de caractere>”);
        for( ncar=0; getche()!=13 ;ncar++ )
                ;
        printf(“\nNumărul caracterelor tastate este %d”, ncar);
        getch();
}
```

5.2. Bucla while

Bucla while este similară ca structură şi mecanism de funcţionare cu bucla for.

Formatul general al instrucţiunii este:

```
while(expresie)
        instrucţiune;
```

Se evaluează expresia şi atâta timp cât valoarea ei este diferită de zero se execută instrucţiunea. Instrucţiunea poate fi simplă sau compusă. Dacă expresia este evaluată la valoarea zero, execuţia instrucţiunii while se încheie. Ca atare, este posibil ca instrucţiunea să nu se execute niciodată. Blocul poate lipsi dacă întreaga acţiune se realizează la evaluarea expresiei.

Reluăm exemplul de calcul al factorialului, realizându-l cu ajutorul buclei while.

Exemplul 5.8.

```
/* Program ex_5_8 */
#include <stdio.h>
```

```
#include <conio.h>
void main (void)
{
        int n, con=1;
        long fact=1;
        printf("\n Introduceţi un număr natural >");
        scanf("%d",&n);
        while(con<=n)
                {
                        fact*=con;      // similar cu fact=fact*con;
                        con++;
                }
        printf("\n%d factorial este %ld",n,fact);
        getch();
}
```

Comparând acest program cu programul din exemplul 5.2. se observă că deosebirile constau în iniţializarea variabilei de control *con* în momentul declarării şi înlocuirea buclei for cu setul de instrucţiuni

```
while(con<=n)
        {
                fact*=con;
                con++;
        }
```

Dacă nu se modifică variabila de control astfel încât expresia logică să devină falsă, bucla nu se va termina niciodată, devenind infinită. Pentru exemplificare, considerăm programul din exemplul 5.9. care calculează, pentru diferite valori ale lui *n* introduse de la tastatură, valoarea sumei $S_n=1^2+2^2+3^2+..+n^2$.

Exemplul 5.9.

```
/* Program ex_5_9 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int i, n, suma ;
        while(1)
                {
                        printf("\n Introduceţi un număr natural >");
```

```
                scanf("%d",&n);
                for(i=1,suma=0; i<=n; i++)
                        suma+=i*i;
                printf("\nSuma pătratelor primelor %d numere
                naturale este %d",n,suma);
        }
}
```

Observaţii:

1. În corpul buclei while se află o buclă for, lucru permis de compilatorul C. Este posibilă utilizarea în corpul unei bucle a altor bucle de acelaşi tip sau de tipuri diferite.
2. Deoarece expresia logică a buclei while este 1, adică tot timpul adevărată. Bucla este infinită şi, după tastarea unui număr, calculul sumei şi afişarea rezultatului, se revine de fiecare dată la solicitarea de a introduce un nou număr. Pentru a termina execuţia programului, se foloseşte facilitatea oferită de sistemul de operare MS-DOS care opreşte execuţia oricărui program ce execută operaţii de citire sau scriere la apăsarea simultană a tastelor <Ctrl> şi <C>.

În exemplele prezentate, bucla while a fost utilizată pentru a realiza un ciclu cu contor (exemplul 5.8.) şi o buclă infinită (exemplul 5.9.). Bucla while poate fi utilizată şi pentru a realiza cicluri cu test iniţial şi cu un număr necunoscut de iteraţii.

Exemplul 5.10.

```
/* Program ex_5_10 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        int ncar ;
        clrscr();
        printf("\n Tastaţi o propoziţie : \n");
        while(getche() != '\r')
                ncar++;
        printf("\nPropoziţia conţine %3d caractere",ncar);
        getch();
}
```

Programul solicită introducerea unei propoziţii de la tastatură. Caracterele care alcătuiesc propoziţia, în număr necunoscut, sunt citite în

corpul buclei while cu ajutorul funcţiei getche şi sunt contorizate de variabila *ncar*. Bucla se termină în momentul în care se apasă tasta <Enter>, acţiune care marchează sfârşitul propoziţiei. Execuţia programului continuă cu afişarea numărului de caractere tastate. Se remarcă faptul că bucla while nu conţine o variabilă de control, iar expresia ei logică conţine apelul unei funcţii. Acest lucru este posibil deoarece funcţia getche este interpretată ca o variabilă având valoarea codului asociat caracterului tastat. Această valoare este comparată cu codul tastei <Enter>, simbolizat în program prin secvenţa escape '\r'.

5.3. Bucla do while

Ca structură, aceasta este similară cu buclele for şi while, dar modul de funcţionare este diferit.

Formatul general al instrucţiunii este:

```
do
        instrucţiune;
while(expresie);
```

Structura buclei do while cuprinde:

- cuvântul cheie *do* care marchează începutul buclei;
- o instrucţiune (simplă sau compusă) care formează corpul buclei;
- o instrucţiune de forma *while(expresie logică);* care constituie testul buclei. Se remarcă faptul că această instrucţiune se termină cu caracterul ;.

Testul se efectuează la sfârşit. Chiar dacă expresia este falsă de la început, corpul buclei se execută o dată.

Exemplul 5.11.

```
/* Program ex_5_11 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        float an ;
        int i=1;
        clrscr();
        do
```

```
            {
            an=2*i/(2*i+1);
            printf("\na%2d=%7.4f",i,an);
            i++;
            }
        while(i<=10);
        getch();
}
```

Programul calculează şi afişează primii zece termeni ai şirului cu termenul general $a_n=2n/(2n+1)$.

În cadrul programului se utilizează variabila reală a_n în care se memorează valoarea termenului curent al şirului, şi variabila întreagă *i*, iniţializată cu 1, ca variabilă de control a buclei.

Iniţial, este executat setul de instrucţiuni cuprins între cuvintele cheie do şi while (corpul buclei) pentru calculul şi afişarea valorii termenului curent al şirului şi pentru modificarea variabilei de control.

În final, este evaluată expresia relaţională *i*<=10. Dacă aceasta este adevărată se reia ciclul de instrucţiuni, în caz contrar, bucla se termină.

5.4. Comenzi pentru modificarea fluenţei controlului

5.4.1. Instrucţiunea return

Este utilizată pentru ieşirea rapidă dintr-o funcţie. Dacă o funcţie returnează o valoare, instrucţiunea return trebuie folosită pentru transmiterea acestei valori rutinei apelante.

Funcţia din exemplul următor returnează cea mai mare valoare dintre două valori transmise ca parametri.

Exemplul 5.12.

```
            int max( int x, int y)
        {
            if(x > y)
                return(x);
            else
                return(y);
        }
```

A doua utilizare constă în ieşirea dintr-o funcţie înainte de sfârşitul codului, în momentul îndeplinirii unei anumite condiţii. În acest caz, în loc de a introduce restul codului într-o instrucţiune if, se poate folosi instrucţiunea return. Dacă funcţia este de tip void, instrucţiunea poate să nu returneze nici o valoare.

5.4.2. Instrucţiunea break

Este utilizată pentru ieşirea rapidă dintr-o buclă, înainte de sfârşitul acesteia.

În cazurile în care singura ieşire dintr-o prelucrare ciclică este cea forţată (se spune că ciclul este cu "test la mijloc"), există mai multe soluţii de codificare a prelucrării respective, dintre care cea mai simplă este următoarea:

Exemplul 5.13.

```
for (suma = 0; ; )
{
        printf("Introduceţi numărul");
        scanf("%d",&val);
        if (val = = 0)
                break ;
        suma += val ;
}
```

Dacă este necesar, corpul unei instrucţiuni de ciclare poate conţine mai multe instrucţiuni break, corespunzătoare mai multor condiţii de ieşire forţată. Posibilitatea ieşirilor forţate nu o exclude pe cea a ieşirii normale, condiţionată de testul efectuat la începutul sau la sfârşitul ciclului. Astfel, ciclul for din exemplul 5.14. are 3 ieşiri – una normală, corespunzătoare condiţiei "s-au citit n valori" şi două forţate, corespunzătoare condiţiilor "s-a citit o valoare nulă" şi, respectiv, "adăugarea la sumă a valorii ar produce depăşire".

Exemplul 5.14.

```
#include <stdio.h>
void main ()
{
        int n, i, suma, val, temp;
        printf("Număr maxim de valori:");
```

```
scanf(“%d”,&n);
for (i=suma=0 ; i<n ; i++)
        {
                printf(“Introduceţi numărul”);
                scanf(“%d”,&val);
                if (val < = 0)
                        {
                                printf(“S-a citit o valoare nulă!\n”);
                                break ;
                        }
                temp = suma + val;
                if(temp<0)
                        {
                                printf(“Valoarea prea mare!\n”);
                                break
                        }
                suma = temp;
        }
        printf(“Suma celor %d valori este “%d\n, i, suma);
}
```

5.4.3. Instrucţiunea continue

Instrucţiunea continue se utilizează numai în interiorul ciclurilor, pentru a determina saltul la începutul secvenţei de instrucţiuni care formează corpul ciclului respectiv. În cazul utilizării instrucţiunii continue nu se părăseşte bucla.

Exemplul 5.15.

```
 Citire(int tab[50])
{
        int i, temp;
        i = 0;
        while (i<50)
        {
                printf(“Introduceţi elementul %d:”, i);
                scanf(“%d”,&temp);
                if (temp<0)
                        continue ;
                tab[i++]=temp ;
```

```
        }
}
```

Exemplul 5.15 prezintă o funcţie care doreşte citirea a 50 de numere pozitive. Dacă se introduce o valoare negativă, se presupune că a fost o eroare de tastare şi se revine la începutul buclei. Întrucât contorul *i* nu a fost incrementat, se va relua citirea aceluiaşi element.

5.4.4. Instrucţiunea de salt goto

Efectul instrucţiunii break este limitat la o singură instrucţiune de ciclare sau selecţie. Există însă situaţii în care se doreşte întreruperea unei succesiuni de prelucrări incluse unele în altele. Un exemplu de acest tip este cel al prelucrărilor pe bază de meniu, în care este reluată ciclic secvenţa de prelucrări ”citeşte, selectează şi tratează opţiunea”, ieşirea din ciclu fiind realizată după tratarea opţiunii de oprire sau dacă la tratarea unei opţiuni se detectează o condiţie de eroare. În aceste situaţii trebuie părăsit atât corpul selecţiei, cât şi cel al ciclului.

Folosirea instrucţiunii break nu rezolvă problema decât parţial, deoarece nu are efect decât la un singur nivel. Pentru rezolvarea completă a problemei se poate recurge fie la utilizarea unei variabile de control şi efectuarea unor teste asupra acesteia, fie la instrucţiunea de salt goto.

Instrucţiunea de salt are structura:

goto etichetă;

unde etichetă este un nume care identifică, sub forma

etichetă: instrucţiune;

instrucţiunea cu care se continuă execuţia programului.

Instrucţiunile goto şi etichetele la care se referă acestea pot fi utilizate doar în corpul funcţiilor. Domeniul de valabilitate al numelui cu rol de etichetă este întregul bloc reprezentând corpul funcţiei în care apare. Bineînţeles că aceeaşi etichetă poate fi referită de mai multe instrucţiuni goto, dar trebuie să identifice o singură instrucţiune.

```

# 6
# Instrucţiuni de decizie (condiţionale)

În funcţie de context, la un moment dat, un program trebuie să aibă posibilitatea să selecteze un anumit set de instrucţiuni pe care să îl execute.

## 6.1. Instrucţiunea if

Ca şi în alte limbaje de programare şi în C instrucţiunea condiţională de bază este instrucţiunea if.

Formatul general al instrucţiunii este :

```
if (expresie)
    instrucţiune;
```

Structura instrucţiunii cuprinde:

- O instrucţiune de forma: if (expresie logică) care are rolul de a decide dacă instrucţiunea (sau setul de instrucţiuni) care urmează se va executa.
- Corpul instrucţiunii de decizie, format dintr-o instrucţiune sau un set de instrucţiuni (care formează o instrucţiune compusă).

Exemplul 6.1.

```
/* Program ex_6_1 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
    float a, b, x;
    clrscr();
    printf("\nIntroduceţi coeficienţii a şi b ai ecuaţiei de gradul I :");
    scanf("%f %f",&a,&b);
```
```

```
        if (a != 0)
                {
                ax=-b/a;
                printf("\nRădăcina ecuaţiei %6.2f*x+%6.2f=0 este x=%f", a,
        b, x);
                }
        getch();
}
```

Programul rezolvă o ecuaţie de gradul I de forma ax+b=0. El solicită mai întâi introducerea coeficienţilor *a* şi *b*, iar dacă *a* este diferit de 0 calculează rădăcina x=-b/a şi afişează rezultatul.

6.2. Instrucţiunea if..else

Structura generală a acestei instrucţiuni este:

if(expresie logică)
setul de instrucţiuni 1
else
setul de instrucţiuni 2

Se completează astfel structura instrucţiunii if, după setul de instrucţiuni care alcătuiesc corpul acesteia, cu o instrucţiune formată din cuvântul cheie else (fără caracterul ;), urmată de o instrucţiune sau un set de instrucţiuni care se va executa în cazul în care expresia logică este falsă.

Reluăm exemplul 6.1., dezvoltându-l, astfel încât să afişeze masajul "Ecuaţie degenerată" dacă a=0.

Exemplul 6.2.

```
/* Program ex_6_2 */
#include <stdio.h>
#include <conio.h>
void main (void)
{
        float a, b, x;
        clrscr();
        printf("\nIntroduceţi coeficienţii a şi b ai ecuaţiei de gradul I :");
        scanf("%f  %f",&a,&b);
        if (a != 0)
                {
```

```
                ax=-b/a;
                printf("\nRădăcina ecuaţiei %6.2f*x+%6.2f=0 este x=%f", a,
        b, x);
                }
        else
                printf("\nEcuaţie degenerată");
        getch();
}
```

Exemplul 6.3. prezintă un program care calculează rădăcina patrată a unui număr introdus de la tastatură. Dacă numărul introdus este negativ, programul va afişa mesajul "Valoare eronată pentru rădăcina pătrată". Linia **#include <math.h>** este necesară deoarece funcţia sqrt pentru calculul rădăcinii pătrate are prototipul în fişierul math.h.

Exemplul 6.3.

```
/* Program ex_6_3 */
#include <stdio.h>
#include <math.h>
void main (void)
{
        float i, rez;
        clrscr();
        printf("\nIntroduceţi valoarea:");
        scanf("%f,&i);
        if (i < 0)
                printf("\nValoare eronată pentru rădăcina pătrată");
        else
                {
                rez=sqrt(i);
                printf("\nRădăcina pătrată = %f,rez");
                }
        getch();
}
```

Deoarece limbajul C tratează ansamblul instrucţiunilor de decizie ca instrucţiuni compuse, este posibilă utilizarea acestora în cadrul corpurilor buclelor sau invers, utilizarea buclelor în corpul instrucţiunilor de decizie.

Efecte laterale

Orice instrucţiune de atribuire închisă între paranteze este de fapt o expresie, a cărei valoare este egală cu valoarea atribuită.

Astfel, expresia:

(suma = 6+4)

are valoarea 10, astfel încât expresia

((suma = 6+4)<25)

are întotdeauna valoarea adevărat (1), întrucât 10<25.

Fie următoarea secvenţă de program:

```
if((c=getch())=='a')
        sort();
else if(c=='b')
        test();
else
        printf("nici o operaţie");
```

Când programul ajunge la expresia ((c=getch())=='a'), se opreşte şi aşteaptă tastarea unui caracter pe care îl atribuie lui *c*, apoi compară acel caracter cu 'a'. Dacă s-a tastat 'a', se execută funcţia sort(). Dacă nu, se continuă cu clauza else. Dacă s-a tastat 'b', se execută funcţia test(). Dacă nici una din condiţii nu este adevărată, se va afişa mesajul „nici o operaţie". Din acest exemplu se poate observa posibilul efect lateral al unei instrucţiuni de atribuire: asignarea valorii atribuite unei expresii.

Operatorul virgulă

Acest operator se poate utiliza pentru introducerea mai multor expresii într-un set de paranteze. Expresiile sunt evaluate de la stânga la dreapta, întreaga expresie luând valoarea ultimei evaluări.

Exemplul 6.4.

```
/* Program ex_6_4 */
#include <stdio.h>
void main (void)
{
        char c,c1;
        c = getch();
```

```
        if ((c1=c, c=getch())= = 'a')
                printf("a");
        else
                printf("c1");
}
```

Primul caracter tastat este atribuit lui *c*. Al doilea caracter este atribuit tot lui *c* (după ce valoarea acestuia a fost atribuită lui *c1*) şi automat întregii expresii, astfel încât, dacă al doilea caracter tastat a fost 'a', se execută clauza if, iar în caz contrar clauza else.

6.3. Instrucţiunea switch

Construcţiile de genul if..else if..else if.. sunt uneori lungi şi greoaie. Instrucţiunea switch permite selectarea unei variante dintre mai multe cazuri posibile. Pentru exemplificare, vom compara două programe care simulează un calculator de buzunar.

Exemplul 6.5.

```
/* Program ex_6_5 */
#include <stdio.h>
void main (void)
{
        int oper, x, z;
        printf("Operand 1:");
        scanf("%d", &x);
        printf("Operand 2:");
        scanf("%d", &y);
        printf("Operator:");
        if ((oper=getche())= = '+')
                printf("\nRezultat adunare: %d\n",x+y);
        else if (oper = = '-')
                printf("\nRezultat scădere: %d\n",x-y);
        else if (oper = = '*')
                printf("\nRezultat înmulţire: %d\n",x*y);
        else if (oper = = '/')
                if (y = = 0)
                        printf("\nOperand 2 eronat pentru împărţire\n")
                else
```

```
                printf("\nRezultat împărţire: %d\n",x/y);
        else if (oper = = '%' || oper = = '&' || oper = = ':')
                printf("\nNu s-a efectuat operaţia\n");
        else
                printf("\nEroare\n");
}
```

Programul efectuează anumite calcule aritmetice. Se observă tratarea cazului special la operaţia de împărţire. De data aceasta, nu a fost necesară includerea între acolade a instrucţiunilor din clauza else respectivă, întrucât nu este vorba de o instrucţiune compusă, ci de o construcţie if..else simplă, chiar dacă ea ocupă mai multe rânduri.

Programul rescris cu ajutorul instrucţiunii switch arată astfel:

Exemplul 6.6.

```
/* Program ex_6_6 */
#include <stdio.h>
void main (void)
{
        int oper, x, z;
        printf("Operand 1:");
        scanf("%d", &x);
        printf("Operand 2:");
        scanf("%d", &y);
        printf("Operator:");
        oper=getche();
        switch(oper)
        {
                case'+': printf("\nRezultat adunare: %d\n",x+y);
                break;
                case '-': printf("\nRezultat scădere: %d\n",x-y);
                break;
                case '*': printf("\nRezultat înmulţire: %d\n",x*y);
                break;
                case '/': if (y = = 0)
                                printf("\nOperand 2 eronat pentru
        împărţire\n")
                          else
                                printf("\nRezultat împărţire: %d\n",x/y);
                break;
```

```
            case'%':
            case'&':
            case ':': printf("\nNu s-a efectuat operaţia\n");
            break;
            default: printf("\nEroare\n");
        }
}
```

Instrucţiunea switch preia valoarea lui *oper* şi o compară cu fiecare caz corespunzător unui prefix case. Dacă se găseşte o identitate, execuţia începe din acel punct şi se termină la întâlnirea unei instrucţiuni break sau la sfârşitul instrucţiunii switch. Dacă nu există nici o identitate şi în instrucţiunea switch apare prefixul default, execuţia începe din acel loc. Dacă nu există nici acest prefix, întreaga instrucţiune este ignorată.

Valoarea testată (în acest caz *oper*) trebuie să fie de un tip compatibil cu int (char, enum, orice variantă de int). Nu se pot utiliza numere reale, şiruri, pointeri sau structuri (se pot însă folosi elemente de tip compatibil cu int din cadrul structurilor).

Pentru fiecare prefix case se poate introduce o singură valoare, care trebuie să fie o constantă.

Instrucţiunea break se se va utiliza de fiecare dată când se termină acţiunile unui anumit prefix case. În caz contrar, se vor executa şi restul instrucţiunilor, până la primul break întâlnit (sau până la sfârşitul instrucţiunii). Astfel, pentru cazurile în care *oper* este '%', '&'sau ':', se observă că se execută aceeaşi acţiune, afişarea mesajului: nu s-a efectuat operaţia.

Dacă nu s-a întâlnit nici un caz de operator specificat, se afişează mesajul corespunzător prefixului default.

6.4. Operatorul condiţional ?:

În general, selecţia între două alternative, pe baza unei condiţii, se face prin instrucţiunea if..else. Utilizarea acestei instrucţiuni fiind foarte frecventă, a fost introdus un operator special, având formatul:

exp1 ? exp2 : exp3

Semnificaţia sa este următoarea: dacă exp1 este adevărată, se evaluează exp2 şi întreaga expresie ia această valoare; în caz contrar, se evaluează exp3 şi întreaga expresie ia valoarea ei.

Fie o funcţie care returnează minimul a două valori:

```
int min( int x, int y)
{
        if(x < y)
                return(x);
        else
                return(y);
}
```

Acest cod poate fi transcris astfel:

```
int min( int x, int y)
{
        return((x < y) ? x : y);
}
```

sau sub formă de macrou:

```
#define min(x,y) ((x<y) ?x:y)
```

De fiecare dată când este întâlnită în program expresia min(e1,e2), ea va fi înlocuită cu ((e2<e2) ?e1:e2) şi se continuă compilarea. Această soluţie este mai generală, *x* şi *y* nemaifiind limitate la tipul int, putând fi de orice tip care acceptă relaţia ”<”.

```

# 7
# Funcţii

## 7.1. Rolul şi modul de funcţionare al unei funcţii

Funcţia este un concept foarte important în matematică. Acest lucru este valabil în multe limbaje de programare, unde întâlnim noţiunea de funcţie în accepţiunea sa generală, şi este esenţial şi în limbajul C, unde toate prelucrările sunt organizate ca o ierarhie de funcţii de mici dimensiuni.

Un program C poate conţine o singură funcţie, **main**, sau mai multe. Limbajul C este astfel conceput încât să încurajeze utilizarea funcţiilor. Ca urmare, întâlnim adesea în programele scrise în C funcţii având o descriere de câteva rânduri, dar care încapsulează prelucrări bine precizate. Pentru a le utiliza, trebuie să cunoaştem numai ce prelucrări realizează, nu şi cum sunt implementate prelucrările. De altfel, aceasta este cheia funcţiilor de bibliotecă, prin utilizarea cărora un programator foloseşte rezultatele muncii altor programatori, nefiind silit să pornească întotdeauna de la zero în rezolvarea problemelor noi.

Funcţiile unui program nu sunt grupate într-un singur fişier. Cel puţin funcţiile bibliotecilor standard sunt compilate separat şi încorporate în programul executabil la editarea de legături. Programatorul poate profita de faptul că unitatea de compilare este fişierul, repartizând funcţiile programului său în mai multe fişiere. În acest fel, modificarea unei funcţii va fi urmată de recompilarea doar a fişierului care o conţine, nu a întregului program, aceasta traducându-se adesea printr-un câştig de timp important.

Diferitele părţi ale programului trebuie să folosească consistent entităţile sale. Aceasta înseamnă că un nume care nu este local funcţiei trebuie să se refere la aceeaşi entitate (valoare, funcţie, tip), indiferent dacă apare într-un singur fişier sau în mai multe fişiere ale programului. Consistenţa este verificată de editorul de legături, care pune cap la cap părţile compilate separat, dar terenul poate fi pregătit de compilator, dacă
```

acestuia i se furnizează informaţii (declaraţii) suplimentare. Acestea constau din fişierele antet.

Funcţia constă dintr-un set de operaţii specifice care se efectuează asupra datelor de intrare (date globale ale programului şi/sau date transmise funcţiei prin intermediul parametrilor acesteia) şi asupra datelor proprii şi furnizează un set de rezultate (modificări efectuate asupra datelor globale, valoarea întoarsă de funcţie, sau alte acţiuni). În momentul apelului, funcţia preia controlul programului, execută propriul set de instrucţiuni şi revine în funcţia apelantă.

Un alt aspect important în crearea funcţiilor îl constituie modularizarea programelor. În general, programele de calcul sunt realizate astfel încât să îndeplinească mai multe activităţi. Separarea şi plasarea acestora în cadrul unor funcţii face ca munca de elaborare şi depanare a programului să fie mult mai uşoară. Programele complexe se lucrează în echipă, fiecare programator realizând una sau mai multe funcţii specifice activităţii programului.

Al treilea aspect al folosirii funcţiilor îl constituie independenţa variabilelor utilizate de acestea. Variabilele declarate în cadrul unei funcţii, numite variabile locale, sunt private, adică nu pot fi accesate de funcţia principală sau de alte funcţii. În acest fel este posibilă utilizarea aceluiaşi nume de variabilă în funcţii diferite, ceea ce face mai uşoară elaborarea programelor complexe de mari dimensiuni.

7.2. Definirea şi declararea funcţiilor

Intrările într-o funcţie şi rezultatele oferite de aceasta constituie interfaţa funcţiei cu funcţia apelantă. Utilizatorul funcţiei trebuie să înţeleagă numai această interfaţă, fără a fi preocupat de detaliile realizării funcţiei. Programatorul care proiectează şi realizează o funcţie trebuie însă să fie preocupat atât de interfaţa funcţiei realizate, cât şi de modul de implementare efectivă a operaţiilor pe care funcţia le va efectua în momentul apelării.

7.2.1. Declararea funcţiilor

Prin declarare, o funcţie este făcută cunoscută programului, astfel încât ea să poată fi apelată de alte funcţii din program.

Declararea funcţiilor se poate realiza în două moduri.

Primul mod, putem să-i spunem clasic, specifică numai numele funcţiei şi tipul valorii returnate:

tip nume_funcţie();

Nu există nici o informaţie despre parametri, astfel încât nu se pot face verificări de eroare. De exemplu, declararea unei funcţii prelucrare ar fi:

void prelucrare();

Al doilea mod, modern, utilizează o construcţie, numită prototip de funcţie, care introduce şi informaţii despre parametri:

tip nume_funcţie($infp_1$, $infp_2$,etc);

unde inp_i poate avea una din formele: "tip" sau "tip nume".

Pentru fiecare parametru formal se poate specifica tipul de dată, la nevoie împreună cu numele său.

Este de preferat utilizarea celei de a doua metode, deoarece permite compilatorului să facă verificările privind numărul şi tipul parametrilor la apelul funcţiilor. De asemenea, la nevoie, se pot realiza conversiile necesare.

7.2.2. Definirea funcţiilor

Prin definire este specificat codul efectiv al funcţiei.

Definiţia unei funcţii trebuie să apară o singură dată în program şi să cuprindă antetul şi corpul funcţiei, în forma următoare:

```
tip_rezultat   nume_funcţie (listă_parametri)
declaraţii_parametri
{
instrucţiuni
}
```

Antetul precizează numele funcţiei, lista parametrilor şi tipul rezultatului. Dacă tipul rezultatului nu este specificat, el este implicit considerat int. Dacă lista parametrilor nu este vidă, fiecare parametru poate fi eventual specificat doar prin numele său, tipurile parametrilor fiind precizate prin declaraţii care preced corpul funcţiei.

Exemplul 7.1.
Se prezintă definirea unei funcţii care realizează ridicarea la puterea n a unei cantităţi întregi.

```
double putere(x,n)
double x;
```

```
int n;
{
        int i ;
        double p=1 ;
        for (i=1 ; i<=n ; ++I)
                p*=x;
        return (p);
}
```

Parametrii *x* şi *n* sunt locali funcţiei, ei nefiind cunoscuţi în afara acesteia. Dacă, de exemplu, în funcţia main ar exista o instrucţiune de forma x += n; aceasta ar fi identificată de către compilator drept o eroare.

Corpul funcţiei este un bloc ce cuprinde definiţii, declaraţii de entităţi locale funcţiei şi instrucţiunile necesare realizării acţiunilor propuse, încadrate de semnele { }. Se poate considera corpul funcţiei ca fiind o instrucţiune compusă care descrie prelucrările necesare pentru a ajunge la valoarea funcţiei pornind de la valorile parametrilor. Rezultatul acţiunii funcţiei este dat de expresia conţinută de instrucţiunea return aflată în corpul funcţiei şi având forma:

```
return;
return expresie;
return (expresie);
```

O funcţie poate conţine un număr oarecare de instrucţiuni return, eventual nici una. Execuţia oricăreia dintre ele provoacă oprirea execuţiei funcţiei şi întoarcerea rezultatului obţinut către apelantul acesteia. Execuţia unei funcţii se poate termina şi printr-o instrucţiune return care nu este urmată de o expresie, sau în absenţa instrucţiunii return, după execuţia ultimei instrucţiuni din corpul funcţiei, fără a se întoarce un rezultat.

Observaţie: o funcţie care uneori întoarce un rezultat, iar alteori nu, este dubioasă, fiind foarte posibilă prezenţa unei erori de programare.

Apelul funcţiei transferă controlul de la funcţia apelantă la funcţia apelată. Apelul se face prin numele funcţiei, urmat de lista parametrilor actuali, închisă între paranteze. Apelul, fiind o instrucţiune, se termină cu caracterul ;.

Apelul unei funcţii are forma generală:

```
nume_funcţie (listă_parametri_actuali);
```

Dacă într-o expresie numele funcţiei nu este urmat de paranteze, se generează un pointer la acea funcţie.

Argumentele actuale, dacă există, sunt transmise funcţiei apelate prin valoare.

La întâlnirea apelului de funcţie, se rezervă memoria necesară pentru parametrii formali şi se face iniţializarea acestora cu valorile corespunzătoare ale parametrilor actuali.

Exemplul 7.2.
Este prezentat un program care utilizează funcţia pentru calculul puterii întregi a unui număr real.

```
#include <stdio.h>
double putere(double x, int n)
{
        int i ;
        double p=1 ;
        for (i=1 ; i<=n ; ++I)
                p*=x;
        return (p);
}
 void main ()
{
        int i;
        printf("Intoduceţi un număr întreg pozitiv:");
        scanf(„%d",&i);
        printf("%d%f%f\n", i, putere(3.14, i), putere (-3.0, i));
        getch();
}
```

Programul cere introducerea de la tastatură a unui număr natural *i* şi calculează puterea *i* a două valori reale. Funcţia getch() nu face decât să aştepte apăsarea unei taste. Efectul este util şi constă în menţinerea afişării ecranului utilizatorului până când este apăsată o tastă oarecare (în acel moment se va încheia execuţia programului şi se va reveni în fereastra de editare).

În acest exemplu funcţia putere este apelată de două ori. La fiecare apel se creează spaţiul necesar parametrilor *x* şi *n* şi (ca şi spaţiul necesar variabilei locale *i*) şi se copiază aici valorile parametrilor actuali corespunzători.

Regulile care guvernează transmiterea valorilor parametrilor sunt similare celor de la iniţializarea variabilelor. Tipul fiecărui parametru este comparat cu cel al parametrului formal corespunzător şi se realizează conversiile implicite de tip: float la double, char şi short la int. Alte conversii necesare trebuie realizate prin cast.

7.3. Utilizarea mai multor funcţii într-un program

Limbajul C permite utilizarea mai multor funcţii în interiorul unui program, existând două deosebiri fundamentale faţă de situaţia întâlnită în limbajul Pascal.

Prima deosebire constă în faptul că în limbajul C nu este permisă definirea unei funcţii în interiorul altei funcţii, lucru care este permis în limbajul Pascal. În limbajul C toate funcţiile sunt definite în mod independent.

A doua deosebire constă în faptul că, în limbajul Pascal, o funcţie sau o procedură definită în cadrul unei alte funcţii nu poate fi apelată de o funcţie din exterior, în timp ce în limbajul C o funcţie poate apela toate celelalte funcţii.

Pentru a exemplifica modul de comunicare între funcţiile limbajului C, considerăm programul din exemplul 7.3. care utilizează trei funcţii pentru a calcula suma pătratelor a două numere reale introduse de la tastatură.

Exemplul 7.3.

```
#include <stdio.h>
#include <conio.h>
/* declararea funcţiilor*/
float suma(float, float);
float(pătrat(float);
float suma_pătrate(float, float);

/* funcţia principală/*
void main (void)
{
        float a, b;
        clrscr();
        printf("\n Introduceţi două valori reale: ");
        scanf("%f %f",&a, &b) ;
        printf(« \n Suma pătratelor este %f", suma_pătrate (a, b));
        getch();
}
```

```
/* definirea funcţiilor*/

float pătrat(float x);
{
        return (x*x);
}

float suma(float x1, float x2)
{
        return(x1 +x2);
}

float suma_pătrate(float y1, float y2)
{
        return(suma(pătrat(y1), pătrat(y2)));
}
```

Toate cele trei funcţii furnizează o valoare reală şi deci au tipul float.

Prima funcţie, numită pătrat, are un singur parametru, *x*, de tip real, şi returnează funcţiei apelante valoarea pătratului acestuia.

A doua funcţie, numită suma, are doi parametrii de tip real, *x1* şi *x2*, şi returnează funcţiei care o apelează valoarea sumei acestora.

Ultima funcţie, numită suma_pătrate, are doi parametri de tip real *y1* şi *y2* şi returnează funcţiei care o apelează valoarea sumei pătratelor acestora.

Funcţia main poate apărea în orice punct din program. De obicei, ea este plasată la început, ceea ce facilitează citirea programului.

Pentru afişarea sumei pătratelor variabilelor *a* şi *b* introduse de la tastatură, funcţia main apelează funcţia printf, care, la rândul său, apelează funcţia suma_pătrate cu parametri actuali *a* şi *b*. Funcţia suma_pătrate, pentru a evalua expresia a^2+b^2, apelează funcţia suma, care are ca parametri actuali două apeluri la funcţia pătrat. Astfel, valorile variabilelor *a* şi *b* sunt transferate pe rând funcţiei pătrat, care furnizează valorile a^2 şi b^2. Acestea sunt transmise apoi funcţiei suma, care furnizează valoarea a^2+b^2, iar în final aceasta este transmisă de către funcţia suma_pătrate funcţiei printf care afişează rezultatul.

7.4. Variabile globale

În exemplele prezentate declararea variabilelor s-a făcut fie în interiorul funcţiei principale main, fie în interiorul celorlalte funcţii. Limbajul C oferă posibilitatea declarării variabilelor din cadrul unui program în afara oricărei funcţii. O variabilă declarată în acest mod poartă numele de variabilă globală şi are proprietatea că poate fi accesată de toate funcţiile care sunt definite după instrucţiunea de declarare a acesteia.

Pentru ilustrare, considerăm programul din exemplul 7.4.

Exemplul 7.4.

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

void citire (void);
void împărţire (void);
void rezultat (void)ş

/* declararea constantelor şi a variabilelor globale*/
const float infinit = 3.4e+38;
float a, b, cit;

/* definire funcţii */

void main (void)
{
        citire();
        împărţire();
        rezultat();
        getch();
}

void citire();
{
        clrscr();
        printf(”Introduceţi două valori reale:”);
        scanf(“%f %f “, &a, &b);
}
```

```
void împărţire ();
{
        if ( b = = 0)
                cit = infinit;
        else
                cit = a / b;
}

void rezultat ()
{
        if (cit = = infinit)
                printf("\nOpera'ie f[r[ sens");
        else
                printf("\na=%f b=%f a:b=%f",a, b, cit);
}
```

Programul citeşte de la tastatură două valori reale, efectuează împărţirea acestora şi afişează rezultatul. Pentru aceasta sunt folosite trei funcţii (citire, împărţire şi rezultat) definite ca fiind de tip void şi fără parametri. Transferarea valorilor de la o funcţie la alta se face prin intermediul variabilelor globale *a*, *b cit* şi *infinit*, declarate la începutul programului fiind accesibile tuturor funcţiilor din program.

Spre deosebire de variabilele locale, care se distrug în momentul părăsirii funcţiei, variabilele globale rămân memorate pe toată durata execuţiei programului.

Variabilele globale pot fi declarate oriunde în interiorul unui program, dar funcţiile definite înaintea lor nu le vor recunoaşte. Dacă în programul din exemplul 7.4. mutăm declaraţiile variabilelor după definirea funcţiei citire şi înainte de funcţia împărţire, compilatorul va semnala două erori generate de faptul că variabilele *a* şi *b* nu sunt definite (pentru funcţia citire).

7.5. Funcţii recursive

O funcţie se consideră a fi recursivă dacă se poate defini în funcţie de ea însăşi.

Puterea recursivităţii, ca instrument matematic, constă în posibilitatea de a defini un set infinit de obiecte printr-un număr finit de reguli.

În felul acesta se poate descrie un număr infinit de calcule printr-un program recursiv finit, chiar dacă acest program nu conţine repetiţii explicite.

Limbajul C permite apeluri recursive de funcţii, apeluri directe sau indirecte. Aceasta înseamnă că funcţia f conţine un apel al său (apel direct), sau că o funcţie g apelată de f conţine un apel al lui f (apel indirect).

La scrierea unei funcţii recursive sunt esenţiale două aspecte:

- condiţia de terminare şi
- numărul de apeluri recursive.

Vom analiza aceste două aspecte pe câteva exemple.

Un exemplu simplu de recursivitate îl constituie calculul factorialului. Relaţia *n!=n(n-1)!* permite o rezolvare recursivă, deoarece se poate calcula factorialul unui număr *n* dacă se cunoaşte factorialul numărului *n-1*. Condiţia de terminare este 0!=0 sau 1!=1.

Numărul de apeluri recursive este limitat doar de dimensiunea memoriei disponibile, deoarece mecanismul recursivităţii salvează mereu în stivă rezultatele parţiale.

Exemplul 7.5.

```
/* Calculul recursiv al factorialului*/

#include <stdio.h>
int factorial (int n)
{
        if (n<0)
        {
                printf("Nu se poate calcula factorialul\n");
                exit(0);
        }
        if (n = = 1)
                return 1;
        return n*factorial(n-1);
}

void main (void)
{
        int a;
        printf("Introduceţi numărul pentru care doriţi să determinaţi
                factorialul\n");
        scanf("%d", &a);
```

```
        printf(" Factorialul lui %d este ", factorial(a));
}
```

7.6. Directive de preprocesare

Directivele de preprocesare constituie comenzi adresate compilatorului, pe care acesta trebuie să le execute înainte de efectuarea compilării propriu-zise.

Orice directivă de preprocesare începe cu caracterul #. Caracterul \ la sfârşit de linie marchează continuarea directivei la linia următoare. O directivă este permisă oriunde în codul sursă.

Cele mai utilizate directive de preprocesare sunt directiva **#define** şi directiva **#include**.

7.6.1. Directiva define

Cea mai simplă utilizare a directivei #define este de a atribui un nume unei constante. Pentru exemplificare, considerăm programul din exemplul 7.6. în care se utilizează directiva #define pentru a defini constanta matematică π, necesară calculului ariei şi volumului unei sfere.

Exemplul 7.6.

```
#include <stdio.h>
#include<conio.h>
#define PI 3.14159

double arie(float);
double volum(float);

void main (void)
{
        float raza;
        clrscr();
        printf("Introduceţi raza sferei:");
        scanf("%f", &raza);
        printf("\nAria sferei de rază R= %g este S=%g, iar volumul V=%g",
                raza, arie(raza), volum(raza));
        getch();
}
```

```
double arie(float r)
{
        return (4*PI*r*r);
}



double volum (float r)
{
        return (4*PI*r*r*r/3);
}
```

Programul solicită introducerea de la tastatură a unei valori reale reprezentând raza sferei şi afişează aria şi volumul sferei de rază *r*. În acest scop, sunt utilizate funcţiile arie şi volum de tipul double pentru evaluarea expresiilor $4\pi r^2$ şi respectiv $4\pi r^3 /3$.

Directiva #define PI 3.14159 are rolul de a atribui constantei 3.14159 numele simbolic PI. Când compilatorul întâlneşte această directivă, mai întâi modifică textul programului substituind cuvântul PI în toate expresiile sau instrucţiunile în care apare cu valoarea 3.14159 şi apoi va efectua compilarea.

Directivele #define pot fi plasate oriunde în interiorul programului, dar este de preferat să fie plasate la început. Întotdeauna numele atribuit prin intermediul acestei directive trebuie scris cu litere mari. Avantajul utilizării directivei #define constă în aceea că, dacă dorim modificarea valorii constantei, acest lucru se va face numai în directivă şi apoi vom recompila programul.

O altă utilizare a directivei #define este aceea de a defini formule de calcul.

Exemplul 7.8.

```
#include <stdio.h>
#include<conio.h>
#define PI 3.14159
#define ARIE(x)  4*PI*x*x
#define VOLUM(y)  4*PI*y*y*y/3

void main (void)
{
```

```
        float raza, s, v;
        clrscr();
        printf("Introduceţi raza sferei:");
        scanf("%f", &raza);
        s=ARIE(raza);
        v=VOLUM(raza);
        printf("\nAria sferei de rază R= %g este S=%g, iar volumul V=%g",
        raza, s, v);
        getch();
}
```

În faza de precompilare instrucţiunile:

```
        s=ARIE(raza);
        v=VOLUM(raza);
```

vor fi transformate în instrucţiunile:

```
        s=4*3.14159*raza*raza;
        v=4*3.14158*raza*raza*raza/3;
```

Mai întâi este substituită valoarea lui PI ca efect al primei directive, după care, în momentul apelului, parametrii *x* şi *y* vor fi substituiţi în formulele respective cu cuvântul raza, rezultând instrucţiunile care vor fi folosite în faza de compilare.

Utilizarea directivei #define în definirea formulelor poartă numele "macro".

Macrourile se execută mai repede decât funcţiile echivalente lor, deoarece compilatorul nu necesită punerea parametrilor pe stivă şi returnarea valorilor. Macrourile, pe de altă parte, măresc dimensiunea codului şi au unele restricţii de utilizare. Dintre ele:

- un macrou nu se poate apela dintr-un alt limbaj de programare;
- nu există pointeri la macrouri (pe când la funcţii există);
- nu toate funcţiile pot fi convertite în macrouri.

Decizia alegerii între macrouri şi funcţii depinde de compromisul care trebuie realizat între viteza de calcul şi economia de memorie.

Un aspect esenţial care trebuie avut în vedere la utilizarea macrourilor îl constituie modul în care sunt construite expresiile din directiva #define, precum şi modul în care sunt apelate.

Se consideră, de exemplu, macroul care calculează suma a două numere:

```
#define SUMA (x, y)    x+y
```

care este utilizat într-o expresie de forma:

val = 10*SUMA (3, 2);

în scopul obţinerii rezultatului 50. Se va constata că, în urma execuţiei instrucţiunii, valoarea variabilei *val* va fi 32, deoarece prin substituirea macroului instrucţiunea devine:

val = 10*3+2;

Având în vedere ordinea în care se execută operatorii aritmetici (ordinea operaţiilor), valoarea afişată este corectă.

Pentru a obţine rezultatul dorit (50), va trebui să utilizăm parantezele pentru a schimba ordinea operaţiilor. Există două moduri posibile în care se poate acţiona:

- în definirea macroului:

  #define SUMA(x, y) (x+y)

- în apelul macroului :

  val = 10*(SUMA(3, 2));

7.6.2. Directiva include

Directiva #include are rolul de a solicita compilatorului ca, în faza de preprocesare, acesta să includă, în fişierul sursă în care este folosită, un alt fişier, cu scopul de a fi compilate împreună. În mod curent, directiva #include este folosită pentru includerea fişierelor antet.

Directiva are efect exclusiv pe parcursul compilării programului care le conţine, modificând textul prelucrat de compilator sau influenţând procesul de compilare. Felul în care este specificat numele modulului de inclus ghidează modul de căutare al fişierului corespunzător modulului. Pentru varianta

#include<nume-fişier>

fişierul sursă nume-fişier este căutat printre fişierele mediului de programare C, în timp ce pentru varianta

#include “nume-fişier”

fişierul este căutat în zona de lucru a utilizatorului pe suportul de informaţie pe care se lucrează (hard disc sau disc flexibil).

7.7. Directive de compilare condiţionată

Directivele :

#if expresie_constantă
….
#endif

determină compilarea condiţionată. Dacă expresie_constantă este evaluată la o valoare diferită de zero, se compilează liniile dintre #if şi #endif.

#if expresie_constantă
….
else
….
#endif

Dacă expresie_constantă este evaluată la o valoare diferită de zero, se compilează liniile dintre #if şi else, altfel se vor compila liniile dintre else şi #endif.

Directivele:

#ifdef NUME
….
#endif

#ifndef NUME
….
#endif

determină compilarea liniilor …., dacă NUME a fost definit, respectiv nu a fost definit.

```

# 8
# Pointeri

Programul şi datele sale sunt păstrate în memoria calculatorului (RAM – Random Acces Memory – memorie cu acces aleator). Memoria este împărţită în octeţi (opt biţi). Doi octeţi formează un cuvânt, patru octeţi formează un cuvânt lung, iar 16 octeţi formează un paragraf (pe IBM PC). Fiecare octet are în memorie o adresă unică, octeţi consecutivi având adrese consecutive.

Un pointer este o variabilă care păstrează adresa unei date, în loc de a memora data însăşi.

Prin utilizarea pointerilor, limbajul C oferă utilizatorilor multiple facilităţi, dintre care mai importante sunt:

- accesarea directă a diverselor zone de memorie prin modificarea adresei memorate în pointer; în acest fel pot fi create şi exploatate liste înlănţuite.
- crearea de noi variabile în timpul execuţiei programului, în cadrul mecanismului de alocare dinamică a memoriei;
- întoarcerea dintr-o funcţie a mai multor valori către funcţia apelantă; în mod uzual, o funcţia întoarce o singură valoare prin intermediul instrucţiunii return;
- accesarea directă unor elemente dintr-o structură de date cum ar fi şiruri sau tablouri.

## 8.1. Declararea şi utilizarea pointerilor

Ca orice tip de variabilă, înainte de a fi utilizată, variabila de tip pointer trebuie declarată. Declararea se face în cadrul unei instrucţiuni având forma generală:

tip *nume_pointer 1, .., *nume_pointer n;
```

unde *tip* poate fi oricare dintre tipurile fundamentale de date (int, float, double, char), iar *nume_pointer i* un şir de caractere cu condiţia ca primul caracter să fie o literă (aceeaşi condiţie ca la numele de variabile). Singura deosebire faţă de o instrucţiune de declarare de variabile este prezenţa caracterului *, care semnalează compilatorului că a fost declarată o variabilă pointer şi nu o variabilă obişnuită.

Pentru a explica modul de lucru cu pointeri, considerăm exemplele următoare:

Exemplul 8.1.

```
main()
{
        int varint, *ptrint;
        ptrint = &varint;
        varint = 200;
        printf("adresa varint: %p\n", &varint);
        printf("valoare varint: %d\n", varint);
        printf("valoare ptrint: %p\n", ptrint);
        printf("valoare adresată de ptrint: %d\n",*ptrint);
}
```

În exemplul 8.1. sunt declarate două variabile, *varint* şi *ptrint*. Prima este o variabilă de tip întreg, care păstrează o valoare de tip int. A doua este o variabilă de tip pointer la o variabilă întreagă, ea memorând adresa unei variabile de tip int. Operatorul * este numit în limbajul C operator de indirectare.

Adresa lui *varint* este atribuită lui *ptrint*, iar valoarea întreagă 200 este atribuită lui varint.

Rezultatul execuţiei programului este:

```
adresa varint: FFD4
valoare varint: 200
valoare ptrint: FFD4
valoare adresată de ptrint: 200
```

Primele două linii dau adresa şi conţinutul lui *varint*. A treia linie reprezintă conţinutul lui *ptrint* (adresa lui *varint*), respectiv adresa de memorie în care programul a creat *varint*. Această valoare poate diferi de la execuţie la execuţie. Ultima linie afişează valoarea memorată la adresa respectivă.

Instrucţiunea

```
varint = 200;
```

este echivalentă cu

*ptrint = 200;

întrucât *varint* şi **ptrint* se referă la aceeaşi locaţie de memorie.

Exemplul 8.2.

```
#include <stdio.h>
void main(void)
{
        float a = 5.2, *ptr_a;
        int *ptr_b, b = 10;
        clrscr();
        printf("\n Valorile iniţiale  a=%6.3f  b=%d", a,b);
        ptr_a = &a;
        ptr_b = &b;
        *ptr_a = a + 10;
        *ptr_b = b + 10;
        printf("\n Valorile finale  a=%6.3f  b=%d", *ptr_a, *ptr_b);
        getch();
}
```

Programul din acest exemplu realizează, cu ajutorul pointerilor, operaţiile a = a + 10 şi respectiv b = b + 10. Instrucţiunea *float a = 5.2, *ptr_a;* declară variabila *a* de tip real şi o iniţializează cu valoarea 5.2 , iar variabila ptr_a de tipul pointer la un real. Instrucţiunea *int *ptr_b, b = 10;* declară *ptr_b* de tipul pointer la întreg, iar variabila *b* de tipul întreg, iniţializând-o cu valoarea 10.

Din aceste două exemple de instrucţiuni de declarare a variabilelor observăm că putem declara, în cadrul aceleaşi instrucţiuni, atât variabile simple, cât şi variabile pointer, în orice ordine, separate prin virgulă.

După ce, prin apelul funcţiei printf, sunt afişate valorile iniţiale ale variabilelor *a* şi *b*, variabilei *ptr_a* îi este atribuită valoarea adresei variabilei *a,* iar variabilei *ptr_b* îi este atribuită valoarea adresei variabilei *b*, folosind instrucţiunile de atribuire ptr_a = &a;
şi ptr_b = &b;

În continuare, programul modifică valorile variabilelor *a* şi *b*, rezultând noile valori 15.2 şi respectiv 20. Pentru aceste operaţii se utilizează adresele variabilelor *a* şi *b*, memorate în *ptr_a* şi respectiv *ptr_b*, precum şi caracterul de indirectare * în faţa numelui variabilelor pointer.

În cadrul instrucţiunii **ptr_a = a + 10;* se accesează zona de memorie a variabilei *a* (prin intermediul adresei conţinute în variabila

pointer *ptr_a*) în care se depune valoarea *a* + 10. Această instrucţiune este identică, ca efect, cu instrucţiunea *a = a + 10*; Din această instrucţiune putem deduce că operatorul de indirectare * este complementar operatorului adresă, adică instrucţiunea **&a = a + 10;* este echivalentă cu instrucţiunea *a = a + 10;*.

Penultima instrucţiune din exemplul 8.2. foloseşte în cadrul funcţiei printf acelaşi mecanism de indirectare pentru afişarea valorilor finale ale variabilelor. Astfel, prin intermediul operatorului de indirectare *, valorile care vor înlocui descriptorii de format se obţin de la adresele memorate de pointerii *ptr_a* şi respectiv *ptr_b*, adică de la adresele variabilelor *a* şi respectiv *b*. Dacă nu era utilizat operatorul *, adică dacă instrucţiunea ar fi fost scrisă sub forma printf("\n Valorile finale a=%6.3f b=%d", ptr_a,ptr_b);, descriptorii de format ar fi fost înlocuiţi ca valorile conţinute în *ptr_a* şi respectiv *ptr_b*, deci cu valorile adreselor variabilelor *a* şi *b*.

8.2. Alocarea dinamică a memoriei

Utilizarea pointerilor permite ca în timpul execuţiei unui program să folosim o parte din memoria rămasă liberă pentru memorarea temporară a unor valori, după care, zona poate fi eliberată şi folosită în alte scopuri.

Pentru a descrie acest mecanism ne vom folosi de exemplele următoare.

Exemplul 8.3.
Se modifică programul din exemplul 8.1. astfel:

```
#include <alloc.h>
main()
{
        int  *ptrint;
        ptrint = (int*)malloc(sizeof(int));
        *ptrint = 200;
        printf("valoare ptrint: %p\n", ptrint);
        printf("valoare adresată de ptrint: %d\n",*ptrint);
}
```

Rezultatul execuţiei va fi de genul:

```
        valoare ptrint: 053E
        valoare adresată de ptrint: 200
```

Spaţiul de memorie necesar stocării valorii variabilei pointer *ptrint* este alocat prin apelul funcţiei malloc, definită în fişierul alloc.h. Această funcţie, a cărei formă generală de apel este

malloc(expresie de tip întreg);

solicită sistemului de operare alocarea pentru programul care o apelează a unei zone de memorie având un număr de octeţi egal cu valoarea expresiei primite ca parametru. Dacă solicitarea este satisfăcută, funcţia returnează adresa primului octet al zonei de memorie alocată. În cazul în care, datorită lipsei de spaţiu de memorie liber, solicitarea nu este satisfăcută, funcţia returnează valoarea NULL.

La apelul funcţiei malloc se remarcă următoarele:

- utilizarea funcţiei sizeof în cadrul expresiei ce determină numărul de octeţi solicitaţi. Această funcţie are următoarea formă generală de apel:

  sizeof(tip);

  şi returnează un întreg reprezentând lungimea exprimată în octeţi a tipului de date pe care îl primeşte ca parametru. De exemplu, sizeof(int) returnează valoarea 2 (numărul de octeţi necesar pentru a memora o dată de tip întreg), iar sizeof(float) retrunează valoarea 4, din aceleaşi considerente.
- utilizarea expresiei (int*) în faţa numelui funcţiei. Funcţia malloc returnează o adresă pe care o putem atribui unui pointer de orice tip. Expresia (int*) specifică faptul că adresa returnată de malloc va fi atribuită unei variabile de tipul pointer la întreg şi constituie un exemplu de utilizare a operatorului cast.

În exemplul 8.3. lui *ptrint* îi este atribuită valoarea returnată de funcţia malloc declarată în alloc.h. Se observă că valoarea lui *ptrint* diferă faţă de cea din exemplul 8.1., dar valoarea adresată este aceeaşi.

Expresia sizeof(int) returnează numărul de octeţi necesari unei variabile de tip întreg, şi anume 2.

Funcţia malloc(valoare) grupează un număr de *valoare* octeţi consecutivi din memoria disponibilă, returnând adresa lor de început (adresa primului octet).

Expresia (int*) indică faptul că adresa de început va foi un pointer de tip int, reprezentând o conversie de tip (cast). Expresia nu este necesară în limbajul C, ea fiind scrisă pentru portabilitatea programului.

Adresa returnată este memorată în *ptrint*. A fost astfel creată dinamic o variabilă întreagă, care poate fi referită prin **ptrint*.

Dacă nu s-ar utiliza instrucţiunea descrisă detaliat mai sus, *ptrint* ar conţine o valoare reprezentând adresa utilizată, dar nu există siguranţa că acea zonă de memorie este liberă. Concluzia evidentă este următoarea: înaintea utilizării unui pointer, acestuia trebuie întotdeauna să i se atribuie o adresă.

Exemplul 8.4.
Se calculează valoarea polinomului de gradul II, $P(x) = a_0x^2 + a_1x + a_2$, pentru o anumită valoare a nedeterminatei *x* citită de la tastatură.

```
#include <stdio.h>
#include <alloc.h>
void main (void)
{
        float *a, val, x;
        int i;
        clrscr();
        if ((a=(float*)malloc(3*sizeof(float))) = = NULL)
                {
                        printf("Memorie insuficientă");
                        exit(1);
                }
        printf("|nIntroduceţi valorile coeficienţilor \n");
        for (i = 0; i <=2; i++)
                {
                        printf("a%d=",i);
                        scanf("%f",&val);
                        *(a+i) = val ;
                }
        printf("Introduceţi valoarea lui x");
        scanf("%f", &x);
        val = *a*x*x+*(a+1)*x+(a+2);
        printf("Valoarea polinomului P(%f) =%f",x,val);
        free(a);
        getch();
}
```

Noutatea acestui program constă în modul în care sunt memorate şi utilizate valorile coeficienţilor polinomiali a_0, a_1 şi a_2. Programul evită declararea a trei variabile de tipul float, în care să memoreze coeficienţii

polinomului. În locul acestora se foloseşte variabila *a*, de tipul pointer la real, şi mecanismul de accesare indirectă.

În program, instrucţiunea

```
if ((a=(float*)malloc(3*sizeof(float))) = = NULL)
```

solicită alocarea unui spaţiu de 12 octeţi, necesar memorării valorilor celor trei coeficienţi polinomiali şi testează dacă solicitarea a fost satisfăcută. În acest sens, valoarea returnată de funcţia malloc este atribuită pointerului *a* şi apoi comparată cu constanta NULL. Dacă solicitarea nu a fost satisfăcută, programul afişează mesajul "Memorie insuficientă" şi execuţia se încheie prin instrucţiunea exit, care solicită revenirea în mediul de programare.

În cadrul buclei for sunt citite şi memorate succesiv valorile celor trei coeficienţi polinomiali. Mai întâi, valoarea corespunzătoare unui coeficient este citită cu funcţia scanf în variabila *val* şi apoi conţinutul acesteia este depus la adresa *a + i*, prin execuţia instrucţiunii **(a+i) = val ;*.

A doua valoare, obţinută pentru *i* =1, nu se va memora la adresa *a* + 1, adică în al doilea octet al primei valori, aşa cum am fi tentaţi să credem la o privire fugară. Compilatorul, ştiind că tipul pointerului *a* este float, va interpreta expresia *a + i* ca fiind *a+i*sizeof(float)* şi deci cele trei valori vor fi memorate corect, fiecărei valori fiindu-i rezervaţi 4 octeţi.

Valorile coeficienţilor astfel memorate sunt utilizate în aceeaşi manieră în cadrul instrucţiunii *val = *a*x*x+*(a+1)*x+(a+2);* pentru a evalua valoarea polinomului.

Din acest exemplu este pusă în evidenţă importanţa specificării tipului de pointer în instrucţiunile de declarare a acestora. O instrucţiune de forma **(ptr + i*) este interpretată ca **(ptr + i *sizeof(tip))* în care *tip* este tipul cu care a fost declarat pointerul *ptr*.

În finalul exemplului este utilizată instrucţiunea free(a); care are rolul de a elibera zona de memorie alocată. Dacă nu am utiliza această funcţie şi am executa programul în cadrul unei bucle, după un anumit timp se va obţine mesajul "Memorie insuficientă" datorită consumării întregului spaţiu de memorie liberă prin apelul repetat al funcţiei malloc.

Mecanismul de alocare şi eliberare a memoriei poartă numele de alocare dinamică şi constituie una din facilităţile puternice oferite de limbajul C.

8.3. Pointeri şi funcţii

Mecanismul de indirectare şi utilizarea pointerilor ca parametri de funcţii constituie una din modalităţile prin care se pot transmite mai multe

valori de la o funcţie la alta. Pentru a înţelege mai bine acest mecanism, considerăm exemplele următoare.

Exemplul 8.5.

```
#include<stdio.h>
#include <conio.h>
void permută(int, int);
void main (void)
{
        int a, b;
        clrscr();
        printf("Introduceţi două valori întregi");
        scanf("%d%d",&a, &b) ;
        permută(a, b);
        printf("Valorile variabilelor în funcţia apelantă a = %d,
                b = %d\n",a,b);
        getch();
}

permută(int x, int y)
{
        int temporar;
        printf("\nValorile variabilelor transmise funcţiei sunt: a=%d,
                 b=%d",x,y);
        temporar = x;
        x = y;
        y = temporar;
        printf("\nNoile valori ale variabilelor sunt: a=%d b=%d",x,y);
}
```

În cadrul acestui program sunt citite de la tastatură două valori întregi care sunt memorate în variabilele *a* şi *b*, locale funcţiei *main*. Aceste valori sunt apoi transmise funcţiei *permută* care are rolul de a le schimba între ele. Pentru a realiza schimbarea, este folosită variabila locală *temporar* în care este salvată mai întâi valoarea primei variabile. Apoi se depune valoarea celei de a doua variabile în prima şi, în final, valoarea salvată în *temporar* se depune în a doua variabilă, realizându-se astfel permutarea celor două valori.

Cu ajutorul funcţiilor *printf* se poate urmări execuţia programului. Apelarea succesivă a acestei funcţii are ca rezultat afişarea valorilor

transmise funcţiei *permută*, a valorilor obţinute la sfârşitul acesteia şi a valorilor după revenirea în funcţia principală.

Dacă este lansat în execuţie programul, pentru valorile introduse 10 şi 20, pe ecran vor apare mesajele:

Valorile variabilelor transmise funcţiei sunt: a=10, b=20
Noile valori ale variabilelor sunt: a=20 b=10
Valorile variabilelor în funcţia apelantă a = 10, b = 20

Se observă că, deşi în funcţia *permută* cele două valori au fost schimbate între ele, în funcţia apelantă *main* această schimbare nu s-a produs. Explicaţia constă în faptul că, la apelul funcţiei, valorile variabilelor *a* şi *b* cu care se face apelul sunt copiate în zona de memorie alocată funcţiei. Funcţia operează asupra valorilor copiate, lăsând neschimbate valorile iniţiale. Acesta este mecanismul transmiterii parametrilor prin valoare.

Pentru a realiza permutarea valorilor şi în funcţia apelantă, vom utiliza pointeri ca parametri ai funcţiei. Exemplul 8.5. devine:

Exemplul 8.6.

```
#include<stdio.h>
#include <conio.h>
void permută(int*, int*);
void main (void)
{
        int a, b;
        clrscr();
        printf("Introduceţi două valori întregi");
        scanf("%d%d",&a, &b) ;
        permută(&a, &b);
        printf("Valorile variabilelor în funcţia apelantă a = %d,
                b = %d\n",a,b);
        getch();
}

permută(int *x, int *y)
{
        int temporar;
        printf("\nValorile variabilelor transmise funcţiei sunt: a=%d,
                b=%d",*x, *y);
        temporar = *x;
```

```
        *x = *y;
        *y = temporar;
        printf("\nNoile valori ale variabilelor sunt: a=%d b=%d",*x, *y);
}
```

Deosebirea esenţială faţă de programul anterior constă în faptul că funcţia *permută* este definită ca având drept parametri două variabile de tip pointer la întreg. La declararea funcţiei, pentru a preciza faptul că parametrii formali sunt pointeri, după tipul acestora se inserează caracterul * precedat de un spaţiu.

Mecanismul de transmitere este tot prin valoare, dar de această dată valorile transmise şi preluate de funcţie în pointerii *x* şi *y* sunt valorile adreselor variabilelor *a* şi *b*, obţinute cu operatorul &.

Utilizând mecanismul de indirectare, funcţia *permută*, prin intermediul celor doi pointeri, va accesa zona de memorie în care se află valorile variabilelor *a* şi *b*, astfel încât schimbarea se va efectua de această dată în funcţia apelantă. Acesta este mecanismul cunoscut sub numele de transmitere a parametrilor prin adresă.

8.4. Tipuri de pointeri

Modelul de memorie determină tipul implicit de pointer, dar acesta poate fi modificat de către utilizator.

8.4.1. Pointeri near

Pointerii near sunt pointeri pe 16 biţi, conţinând doar deplasamentul, deci se bazează pe un registru de segment pentru calculul adresei. Sunt uşor de manipulat, întrucât orice operaţie aritmetică se poate executa fără a ţine cont de segment.

8.4.2. Pointerii far

Pointerii far sunt pointeri pe 32 de biţi, conţinând adresa de segment şi deplasamentul. Permit segmente de cod multiple, programe mai mari de 64 Kocteţi, şi adresarea a mai mult de 64 Kocteţi de date.

În cazul pointerilor far pot apare unele probleme. Pornim de la faptul că o aceeaşi adresă poate fi referită prin perechi diferite segment:deplasament.

De exemplu, să considerăm o aceeaşi adresă scrisă în trei moduri diferite:

a- 0000:0120
b- 0010:0020
c- 0012:0000

În cazul operatorilor relaţionali (<, >, <=, >=) se foloseşte la comparare doar deplasamentul, astfel încât:

a > b
b > c
a > c

Operatorii = = şi != folosesc toţi cei 32 de biţi ca o valoare de tip unsigned long şi nu ca o adresă de memorie completă, astfel încât:

a != b
a != c
b != c

Dacă se adună o valoare la un pointer far, se modifică doar deplasamentul, iar la depăşire acesta sare la valoarea de început (acelaşi gen de inconvenient apare şi la operaţia de scădere).

Exemplu: 5031:FFFF + 1 = 5031:0000 în loc de 5032:0000.

8.4.3. Pointerii huge

Pointerii huge sunt tot pe 32 de biţi, conţinând adresa de segment şi deplasamentul, dar sunt normalizaţi, adică în partea de segment intră cât se poate de mult din valoare. Deci deplasamentul va fi cuprins între 0 şi 15 (0 şi FH), întrucât un segment poate începe din 16 în 16 biţi.

Pentru normalizarea unui pointer, acesta se converteşte în valoare pe 20 biţi, din care primii 16 reprezintă adresa de segment, iar ultimii 4 deplasamentul.

Exemplu:

Valoare	Adresa normalizată
2F8:0532	2FD3:0002
0000:0123	0012:0003
500D:9407	594D:0007

Se observă că prin normalizare se pot elimina inconvenientele pointerilor de tip far. În acest caz, pentru o adresă de memorie există o unică

adresă huge (pereche segment:deplasament) posibilă. Cu pointeri normalizaţi, orice operator relaţional lucrează corect. La fel se întâmplă în cazul operaţiilor de adunare şi scădere, deoarece la depăşire segmentul este modificat corespunzător. Aceste particularităţi ale pointerilor huge permit manipularea structurilor de date mai mari de 64 Kocteţi.

Exemplu: 5031:FFFF + 1 = 5032:0000.

Utilizarea pointerilor normalizaţi costă însă timp, deoarece aritmetica se realizează prin apel de subrutine speciale, ceea ce o face mult mai lentă decât în cazul pointerilor near sau far.

8.4.4. Declararea pointerilor ca near, far sau huge

Modificarea tipului pointerilor este utilă pentruevitarea pierderilor de timp şi spaţiu şi/sau pentru a se beneficia de posibilitatea adresării în afara segmentului implicit. Trebuie reţinut însă că în astfel de declaraţii pot apare erori!

Exemplul 8.7.

Fie, într-un model de memorie mic, programul:

```
#include<stdio.h>

void scrie(s)
{
        int i;
        for (i =0; s[i] != 0; i++)
                printf(s[i]);
}

void main (void)
{
        char near *şir;
        şir = “Salut!\n” ;
        scrie(şir) ;
}
```

Programul este corect, declaraţia near fiind chiar redundantă. Într-un model de memorie mare, pointerul *şir* va fi tot de tip near, iar pointerul *s* din main va fi implicit far, astfel încât adresa pusă pe stivă pentru *şir* va fi formată din două cuvinte, ceea ce este greşit.

Soluţia constă în folosirea modului modern de definire:

```
void scrie(char *s)
{
        /* corpul funcţiei */
}
```

Cu această modificare, se va aştepta un pointer la char, iar modelul de memorie fiind mare, acesta va trebui să fie de tip far. Ca urmare, pe stivă se pune automat segmentul de date DS alături de valoarea pe 16 biţi pentru *şir*.

Problema este şi inversă: dacă nu se foloseşte un prototip ca în C-ul modern, un model de memorie mic va pune doi octeţi pe stivă pentru adresă, întrucât funcţia aşteaptă doar deplasamentul.

```

# 9
# Tablouri. Şiruri de caractere

Tablourile, similare matricelor din algebră, constituie colecţii de date de acelaşi tip, care pot fi accesate prin intermediul aceluiaşi nume de variabilă, folosind indici.

## 9.1. Tablouri unidimensionale

Ca orice tip de variabilă, înainte de a fi utilizate, variabilele de tipul tablou trebuie declarate. Un tablou este format dintr-un număr finit de elemente de acelaşi tip, numit tip de bază al tabloului, stocate într-o zonă de memorie compactă. Sintaxa unei declaraţii de tablou este:

tip_de_bază nume_tablou [dimensiune]

unde:

- *tip_de_bază* reprezintă tipul valorilor ce vor fi memorate în tablou. Acesta poate fi oricare dintre tipurile de date fundamentale (char, int, long, float, double) cu sau fără semn.
- *nume_tablou* reprezintă numele tabloului şi este format, ca orice nume de variabilă, dintr-un şir de caractere, dintre care primul este o literă;
- o pereche de paranteze drepte care semnalează compilatorului că variabila ce o precede este un tablou.
- *dimensiune* precizează, sub forma unei expresii constante, numărul de elemente ale tabloului.

De exemplu,

```
float t[12];
```

defineşte un tablou numit *t*, cu 12 elemente de tip float (real). Dacă fiecărui element de tip float îi sunt alocate 4 unităţi de memorare (sizeof(float)=4), atunci tabloului *t* îi va fi alocată o zonă de 12*4 = 48 de unităţi de memorare.
```

De multe ori numărul elementelor unui tablou este specificat printr-o constantă simbolică, introdusă cu ajutorul directivei #define a preprocesorului. Se preferă această metodă, deoarece:

- astfel, programul sursă este mai uşor de citit, iar un program uşor de citit este uşor de întreţinut;
- modificarea dimensiunii tabloului afectează un singur punct din programul sursă.

Indicii elementelor de tablou încep întotdeauna de la 0 (zero).; dacă un tablou *t* este declarat cu *n* elemente, atunci primul element este *t[0]*, iar ultimul element este *t[n-1]*.

Un tablou poate fi declarat incomplet, prin omiterea dimensiunii, în următoarele situaţii:

- declaraţia se referă la un tablou extern, pentru care s-a rezervat spaţiu în alt fişier sursă;

Exemplul 9.1.

```
/*Fişier sursă A*/
…
int x[10];
/*alocă spaţiu pentru tabloul x*/
…


/*Fişier sursă B*/
…
extern int x[];
/*de face referire la tabloul definit în A*/
…
```

- tabloul declarat este parametru formal al unei funcţii; în acest caz, declaraţia este echivalentă cu cea a unui pointer;

Exemplul 9.2.

```
int  F (int N, char T[ ]);
/*declaraţie echivalentă cu: int F (int N, char *T); */
```

- declaraţia conţine o iniţializare, pe baza căreia se determină numărul de elemente din tablou.

Exemplul 9.3.

```
double T [ ] = {1, 2, 3, 4};
/*declaraţia este echivalentă cu: double T [4] = {1, 2, 3, 4};*/
```

Pentru a selecta un element de tablou se utilizează operatorul de indexare [ ], ai cărui operanzi sunt tabloul din care face parte elementul selectat şi expresia de tip întreg a cărei valoare indică poziţia elementului în tablou. Elementele de tablou pot fi folosite în orice loc în care sintaxa permite apariţia unei variabile.

Exemplul 9.4.

Se citesc de la tastatură şapte valori reale, reprezentând temperaturile înregistrate în cursul unei săptămâni, acestea sunt memorate şi este calculată temperatura medie a săptămânii.

```
#include<stdio.h>
include<conio.h>
void main (void)
{
        float temp[7];
        float media = 0;
        int i;
        clrscr();
        printf("\n Introduceţi temperaturile zilnice\n\n");
        for (i=0; i<7; i++)
        {
                        printf("Temperatura din ziua %1d",i+1);
                        scanf(„%f",&temp[i]);
                        media += temp[i];
        }
        media /= 7;
        printf("Temperatura   medie %f",media);
        getch();
        }
```

Valorile temperaturii au fost memorate într-un tablou cu numele *temp*. Fiecare element al tabloului constituie o variabilă şi deci are o valoare şi o adresă. Valoarea se obţine prin accesarea directă a acestuia folosind indicele asociat, iar adresa prin intermediul operatorului &. Astfel, *temp[1]* reprezintă valoarea celui de al doilea element al tabloului *temp*, iar

&temp[1] adresa acestuia. În program s-au folosit adresele elementelor tabloului furnizate de expresia *&temp[i]* pentru memorarea valorilor citite cu funcţia scanf şi, respectiv, valorile acestora furnizate de expresia *temp[i]* pentru calculul mediei.

Întrucât microprocesorul lucrează cu adrese şi nu cu indici, în momentul în care declarăm o variabilă tablou, compilatorul va considera numele acesteia ca pointer, căruia îi atribuie valoarea adresei primului octet al zonei de memorie în care se memorează valorile elementelor, adică adresa primului element. Alocarea zonei de memorie se face de către compilator, dimensiunea acesteia fiind evaluată conform relaţiei **dimensiune*sizeof(tip)**, în care *dimensiune* este dimensiunea tabloului, iar *tip* tipul acestuia. În concluzie, declararea şi utilizarea tablourilor este transformată de compilator într-un mecanicm de alocare a memoriei, iar accesarea acesteia se face în mod indirect prin intermediul pointerilor. În acest sens, expresia **nume_tablou[i]** este echivalentă cu **(nume_tablou+i),** iar expresia **&nume_tablou[i]** este echivalentă cu **nume_tablou+i*sizeof(tip)**. Particularizând pentru programul din exemplul 9.4., expresia *temp[i]* este echivalentă cu **(temp+i),** iar expresia *&temp[i]* este echivalentă cu **temp+i*sizeof(float)**.

Exemplul 9.5.
Programul solicită utilizatorului să introducă o sumă exprimată în lei şi determină numărul de bancnote sau monede necesare formării acestei sume.

```
#include <stdio.h>
#include <conio.h>
#define N_VALORI 7
int valori[ ] = {500000, 100000, 50000, 10000, 5000, 1000, 500} ;
void main (void)
{
        int suma, i;
        clrscr();
        printf("\nIntroduceţi o sumă exprimată în lei");
        scanf(„%d", &suma);
        for (i=0; i<N_VALORI; i++)
        {
                printf("\nValoarea bancnotei %6d   Numărul de bancnote
                %4d",valori[i], suma/valori[i]);
                suma %= valori[i];
        }
        getch();
}
```

Este utilizat tabloul *valori* de tipul întreg şi operatorii aritmetici / (împărţire), respectiv % (rest). În corpul buclei for, pentru fiecare valoare a lui *i* se afişează valoarea corespunzătoare a bancnotei sau monedei conţinută în elementul *valori[i]* din tabloul *valori* şi numărul acestora, obţinut prin împărţirea sumei la valoarea afişată anterior. Suma este apoi modificată, căpătând valoarea rămasă (obţinută cu ajutorul operatorului %).

Ceea ce trebuie pus în evidenţă este modul în care se face iniţializarea elementelor tabloului. De aceea trebuie remarcată instrucţiunea de declarare a tabloului. Valorile pe care elementele le vor primi sunt specificate în interiorul unei acolade precedate de semnul egal. Prima valoare dintre acolade va fi atribuită primului element al tabloului, a doua celui de al doilea şi aşa mai departe. Astfel, *valori[0]* va primi valoarea 500000,.., iar *valori[6]* va primi valoarea 500. Se poate observa că nu a fost specificată dimensiunea tabloului. Aceasta este stabilită de către compilator ca fiind egală cu numărul de valori, separate prin virgulă, existente în interiorul acoladelor.

Un alt aspect important care se poate observa este faptul că declararea tabloului s-a făcut în afara funcţiei main, astfel variabila *valori* fiind o variabilă globală. Acest mod de declarare nu a fost întâmplător. Se ştie că variabilele declarate în interiorul unei funcţii sunt variabile locale, Acestea sunt create în momentul apelării funcţiei şi sunt distruse în momentul în care funcţia revine în programul apelant. Din acest motiv variabilele locale ale unei funcţii nu pot fi iniţializate, deoarece se încearcă atribuirea unor valori pentru nişte variabile care nu au fost încă create.

Exemplul 9.6.
Programul următor arată cum se poate face totuşi o iniţializare a unei variabile simple sau a unui tablou în cadrul unei funcţii.

```
#include <stdio.h>
#include <conio.h>
#define N_VALORI 7
void nr_bancnote (int)

void main (void)
{
        int suma;
        clrscr();
        printf("\nIntroduceţi o sumă exprimată în lei");
        scanf(„%d", &suma);
```

```
        nr_bancnote (suma):
        getch();
}

void nr_bancnote (int x)
{
        int i;
        static int valori[ ] = {500000, 100000, 50000, 10000, 5000,
                                1000, 500} ;
        for (i=0; i<N_VALORI; i++)
        {
                printf("\nValoarea bancnotei %6d   Numărul de bancnote
                        %4d",valori[i], x/valori[i]);
                x %= valori[i];
        }
}
```

Se observă că la declararea tabloului *valori* în funcţia *nr_bancnote* s-a folosit prefixul *static*. Reamintim că acest prefix este utilizat pentru declararea variabilelor statice. Acestea sunt variabile declarate în interiorul funcţiilor, sunt create la lansarea în execuţie a programului şi nu sunt distruse în momentul în care se părăseşte corpul funcţiei. Ele rămân în memorie pe toată durata execuţiei programului, sunt similare variabilelor globale sau externe şi pot fi iniţializate.

9.2. Tablouri multidimensionale

În limbajul C, un tablou multidimensional este tratat ca un tablou ale cărui elemente sunt tablouri; în declaraţia unui tablou multidimensional, parantezele pătrate se asociază de la stânga la dreapta. Declaraţia unei matrice cu elemente de tip float, cu *N* linii şi *M* coloane este de forma următoare:

```
#define N 10
#define M 6
float A [N] [M];
```

Declaraţia de mai sus este interpretată astfel: obiectul declarat este *A*, un tablou cu *N* elemente, fiecare element *A[i]* al lui *A* este un tablou cu *M* elemente, fiecare element *A[i] [j]* al lui *A[i]* este de tip float.

Din ordinea în care sunt luate în considerare cele două dimensiuni ale matricei *A* se pot trage două concluzii :

- matricea este memorată pe linii;
- numai prima dimensiune a unui tablou multidimensional poate rămâne neprecizată, în caz contrar, compilatorul nu mai are informaţiile necesare pentru calculul adresei unui element al tabloului.

Ca şi la tablourile unidimensionale, tablourile multidimensionale pot fi iniţializate în momentul declarării. Pentru un tablou bidimensional se foloseşte o pereche de acolade în interiorul cărora sunt plasate alte perechi de acolade separate prin virgulă, ce conţin elementele liniilor tabloului.

Exemplul 9.7.

Definirea şi iniţializarea unei matrice unitate cu 3 linii şi 3 coloane.

```
double matrice_unitate [3] [3] = {   {1, 0, 0},
                                     {0, 1, 0},
                                     {0, 0, 1}
                                 } ;
```

Dacă dimensiunea tabloului iniţializat nu este precizată explicit, atunci ea este dictată de numărul de valori prezente în partea de iniţializare. Dacă dimensiunea tabloului este precizată, iar în partea de iniţializare se specifică mai puţine valori decât numărul de elemente ale tabloului, atunci restul elementelor se iniţializează cu zero (toţi biţii zero, dar interpretarea depinde de tipul de bază al tabloului).

Standardul ANSI precizează că prezenţa acoladelor nu este obligatorie în cazul iniţializării tablourilor multidimensionale. În acest caz, iniţializarea elementelor de tablou se face în ordinea păstrării acestora în memorie, toate dimensiunile trebuind precizate în declaraţie.

Accesarea elementelor din tablou se face cu ajutorul unei perechi de indici în care primul este indicele liniei, iar al doilea este indicele coloanei. Numerotarea acestora, ca şi în cazul tablourilor unidimensionale, se face începând cu 0. Astfel, elementul a_{21} al tabloului *A* va fi accesat prin *a[1][0]* şi nu prin *a[2][1]*.

Exemplul 9.8.

Program care calculează produsul a două matrice *A* şi *B*, rezultatul fiind depus în matricea *C*.

```
#include <stdio.h>
#include <conio.h>
int a [3] [2] = {{1, 2},
                 {-1, 1},
                 {5, 7}};
```

```
int b [2] [2] = {{-1, 0},
                 {0, -1}};
int c [3] [2];

void prod_mat (int, int, int);

void main (void)
{
        int i, j;
        clrscr();
        printf("\nMatricea A");
        for (i=0; i<3; i++)
        {
                printf("\n");
                for (j=0; j<2; j++)
                        printf("   %2d",a[i] [j]);
        }
        printf("\nMatricea B");
        for (i=0; i<2; i++)
        {
                printf("\n");
                for (j=0; j<2; j++)
                        printf("   %2d",b[i] [j]);
        }
        prod_mat (3, 2, 2);
        printf("\nMatricea produs A*B");
        for (i=0; i<3; i++)
        {
                printf("\n");
                for (j=0; j<2; j++)
                        printf("   %2d",c[i] [j]);
        }
        getch();
}

void prod_mat (int m, int n, int l)
{
        int i, j, k;
        for (i=0; i<m; i++)
                for (j=0; j<2; j++)
```

```
        {
                c [i] [j] = 0;
                for (k=0; k<n; k++)
                        c [i] [j] += a [i] [k] * b [k] [j] ;
        }
}
```

În cadrul funcţiei *prod_mat* sunt utilizate trei bucle for. Prima buclă stabileşte indicele *i* al liniei primei matrice, a doua indicele *j* al coloanei celei de a doua matrice, iar ultima buclă calculează elementul c_{ij} al matricei produs, prin însumarea produselor dintre elementele omoloage de pe linia *i* şi coloana *j*.

Este important de precizat că, la declararea unei variabile de tipul matrice, compilatorul va aloca spaţiu de memorie, având dimensiunea exprimată în octeţi dată de relaţia **nr_linii * nr_coloane * sizeof(tip)**, şi tratează numele tabloului ca un pointer căruia îi atribuie adresa primului octet din memoria alocată. Acest pointer este utilizat pentru accesarea elementelor tabloului, deoarece microprocesorul nu lucrează cu indici, ci cu adrese de memorie.

Pentru a explica modul în care sunt folosiţi pointerii şi mecanismul de indirectare pentru accesarea elementelor unui tablou bidimensional (matrice), vom considera exemplul 9.9.

Exemplul 9.9.
Înmulţirea unei matrice cu un scalar.

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

void main (void)
{
        int mat [4] [4] = {{1, 3, 5, 7},
                           {2, 4, 6, 8},
                           {-1, -2, -3, -4}
                           {-5, -6, -7, -8}};
        int alfa = 10, i, j;
        clrscr();
        printf("\nMatricea iniţială");
        for (i=0; i<4; i++)
        {
                printf("\n");
```

```
            for (j=0; j<4; j++)
                printf("   %3d", mat[i] [j]);
        }
        for (i=0; i<4; i++)
            for (j=0; j<4; j++)
                mat [i] [j] *= alfa;
        printf("\nMatricea modificată");
        for (i=0; i<4; i++)
        {
            printf("\n");
            for (j=0; j<4; j++)
                printf("   %3d", mat [i] [j]);
        }
        getch();
}
```

Pentru accesarea elementelor prin intermediul pointerilor, compilatorul C consideră fiecare linie a matricei ca un element al unui tablou unidimensionalla care se poate face referire printr-un singur indice. Astfel, *mat [0]* desemnează prima linia a matricei, *mat [1]* a doua linie şi aşa mai departe. Adresa unei linii *i* este adresa primului element al acesteia şi se obţine prin intermediul unei expresii de indirectare de forma *(mat+i).

Adresa elementului *mat [i] [j]* este furnizată de expresia *(mat+i)+j, iar valoarea acestuia se obţine prin indirectare folosind expresia *(*(mat+i)+j).

Mecanismul de accesare a elementelor unui tablou bidimensional cu ajutorul pointerului asociat numelui acestuia poartă numele dublă indirectare.

9.3. Tablouri şi pointeri

Cu excepţia cazului în care apare ca argument al operatorilor sizeof (determinarea spaţiului de memorie ocupat) sau & (calculul adresei), numele unui tablou este echivalent cu un pointer constant către primul element al tabloului. Pentru orice tablou *T*, expresiile

T &T &T[0]

sunt echivalente. Tipul oricăreia dintre aceste expresii este "pointer către tipul de bază al tabloului *T* ", iar valoarea sa este reprezentată de adresa la care este memorat tabloul *T*. Accesul la primul element al tabloului *T* se poate realiza sub două forme: T[0] sau *T.

Rezultă că un pointer poate fi folosit ca şi cum ar fi numele unui tablou. Fac excepţie pointerii cu tipul de bază void şi pointerii către funcţii. Această restricţie este extrem de rezonabilă, deoarece în primul caz am avea un tablou de elemente care nu ar avea nici o valoare, iar în al doilea caz ar însemna că un număr de funcţii ar putea fi "turnate" într-o structură omogenă. Pot exista tablouri care au elemente pointeri către void sau pointeri către funcţii. Un pointer către un pointer către o funcţiepoate fi folosit ca şi cum ar fi numele unui tablou.

În contextul declaraţiilor

```
int t [10], * p = & t [0] ;
int (* ftab [5]) (char *) ;
```

variabila *p* este un pointer către primul element din tabloul de întregi *t* (în loc de ** p = & t [0]* se putea scrie mai scurt *p = t*), iar *ftab* este definit ca un tablou cu 5 elemente de tip pointer către char şi rezultat de tip întreg.

Numele unui tablou este un pointer **constant** (ca şi cum ar fi declarat cu modificatorul **const**), deci valoarea sa nu poate fi modificată. Această restricţie nu se aplică pentru numele de parametri formali de tip tablou: un parametru formal de tipul tablou este o simplă variabilă pointer, iniţializată cu valoarea parametrului actual corespunzător.

Deoarece numele de tablouri sunt interpretate ca pointeri, rezultă că tablourile nu pot fi copiate printr-o singură atribuire, ci element cu element.

Înrudirea strânsă între tablouri şi pointeri devine evidentă în regulile după care se efectuează în limbajul C operaţiile aritmetice între pointeri, precum şi între pointeri şi întregi. Limbajul C, spre deosebire de alte limbaje de nivel înalt, permite o serie de operaţii aritmetice (rezervate în general limbajelor de asamblare) asupra pointerilor, operaţii definite independent de maşina pe care se rulează programul, păstrând eficienţa codului generat.

Rezultatul adunării sau al scăderii dintre un pointer şi un întreg are sens dacă valoarea pointerului reprezintă adresa unui element de tablou.

Adunarea între un pointer şi un întreg se defineşte astfel: dacă *p* este un pointer la entităţi de tip *tab* şi *i* este un întreg, atunci rezultatul expresiei *p + i* este un pointer de acelaşi tip cu *p*, iar valoarea sa se obţine adunând la valoarea pointerului *p* valoarea întreagă *i ** sizeof(*tab*). Rezultă că expresia * (p + i) este tratată de către compilator exact ca şi expresia p [i].

Considerând *t* un tablou cu elemente de tip *tab*, şi *p* un pointer la *tab*, după execuţia atribuirii

```
p = & t [i]
```

valoarea expresiei *p + n* este reprezentată de adresa elementului de tablou *t [i+n]*.

În mod asemănător se defineşte şi scăderea unui întreg dintr-un pointer, expresia *p – n* fiind interpretată ca *p +(-n)*. Deci, dacă *p* are ca

valoare adresa elementului de tablou *t[i]*, atunci *p-n* are ca valoare adresa elementului *t [i-n]*.

Dacă pointerul *p* are ca valoare adresa elementului de tablou *t[i]*, atunci expresia *p + n* nu are sens decât dacă *i + n* este mai mare sau egal cu zero şi mai mic sau egal cu numărul de elemente din tabloul *t*. Trebuie reţinut că nu poate exista un pointer către elementul dinaintea primului element de tablou, dar că un pointer către elementul următor ultimului element de tablou este perfect corect.

Pe multe calculatoare aceste reguli nu pun mari probleme, dar pe calculatoare construite cu microprocesoare din familia 80x86, respectarea lor este vitală. Structura unei adrese folosite de aceste microprocesoare constă într-o pereche de cuvinte de 16 biţi: o adresă de segment şi o deplasare (offset). Adresa fizică referită este calculată în hardware după formula 16*adresa_segment + deplasare. Un pointer dintr-un program C poate reprezenta fie o adresă completă, fie numai partea de deplasare.

Fie declaraţiile:

double t [10], * p = & t [1];

Să presupunem că tabloul *t* are adresa 7000:0003 (adresa de segment 7000H, deplasare 0003H). Pointerul *p* are deci, ca valoare, adresa 7000:000B (deoarece 0003+sizeof(double)=000B). Expresia *p*-1 are valoarea 7000:0003 (adresa lui *t[0]*), dar expresia *p*-2 are valoarea 7000:FFFB, foarte departe de valoarea "aşteptată" 6FFF:000B (pentru aflarea căreia ar trebui efectuate calcule suficient de complicate, care ar compromite eficienţa operaţiilor aritmetice cu pointeri).

De vreme ce un pointer se poate aduna cu un întreg, înseamnă că pointerilor li se pot aplica şi operatorii de incrementare şi decrementare. Combinarea acestor operatori cu cei de indirectare poate pune probleme celor care nu sunt siguri de ordinea în care sunt aplicaţi operatorii.

Exemplul 9.10.

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

void main (void)
{
        int t [4] = {0, 1, 2, 3};
        int *p = & t [1] ;
        printf(" %d\n ", * p++);
        printf(" %d\n ", * ++p);
        printf(" %d\n ", ++ *p);
}
```

Ca rezultat al execuţiei programului, vor fi afişate numerele 1, 3, 4.

Expresia ***p++** se execută astfel: pointerul *p* este incrementat (valoarea lui devine *& t [2]*), apoi se aplică operatorul de indirectare asupra valorii lui *p* dinainte de incrementare (*& t [1]*), obţinându-se valoarea lui *t [1]*, adică 1.

Evaluarea expresiei ***++p** începe prin incrementarea pointerului p (care devine *& t [3]*), pentru a continua prin aplicarea operatorului de indirectare asupra valorii incrementate; rezultatul este valoarea lui *t [3]*, adică 3.

Expresia ++ ***p** implică incrementarea elementului *t [3]*, obţinut prin aplicarea operatorului de indirectare asupra pointerului *p*; rezultatul expresiei este valoarea lui *t [3]* după incrementare.

Scăderea a doi pointeri către entităţi de acelaşi tip este definită după cum urmează: fie p1 şi p2 doi pointeri cu acelaşi tip de bază, având ca valori adresele a două elemente ale aceluiaşi tablou *t*. Dacă *p1 = & t [1]* şi *p2 = & t [2]*, atunci expresia *p1 – p2* are ca rezultat valoarea întreagă *i1 – i2*. Rezultatul scăderii a doi pointeri nu este definit dacă valorile lor nu reprezintă adresele a două elemente din acelaşi tablou.

Pe baza scăderii se definesc operatorii relaţionali (care nu trebuie confundaţi cu operatorii de comparaţie) aplicaţi pointerilor, după următoarele reguli:

p1 < p2	înseamnă	p1-p2 < 0
p1 <= p2	înseamnă	p1-p2 <= 0
p1 > p2	înseamnă	p1-p2 > 0
p1 >= p2	înseamnă	p1-p2 >= 0

Rezultatul unui operator de comparaţie aplicat unor pointeri nu este definit decât dacă rezultatul operaţiei de scădere a celor doi pointeri este definit.

9.4. Şiruri de caractere

În limbajul C, un şir de caractere se reprezintă printr-un tablou cu elemente de tip char, dintre care cel puţin unul este nul (caracterul ASCII NULL, '\0' în C). Acest caracter este interpretat ca terminator al şirului. Toate funcţiile standard care prelucrează şiruri de caractere presupun că sfârşitul şirurilor prelucrate este marcat în acest fel.

O constantă şir de caractere este tratată de către compilator ca şi cum ar fi numele unui tablou de caractere iniţializat cu respectiva constantă (echivalent, am putea spune că se tratează ca şi cum ar fi un pointer către

char, având ca valoare adresa zonei de date unde este depus şirul de caractere). Pentru un şir conţinând *n* caractere, compilatorul rezervă *n*+1 locaţii; în primele *n* sunt depuse caracterele din şir, iar ultima conţine terminatorul '\0'.

Şirurile de caractere, fiind tablouri de caractere, pot fi iniţializate cu şiruri de caractere între ghilimele. Fiecare element al şirului, inclusiv caracterul '\0', ataşat automat de compilator, iniţializează câte un element de tablou.

Astfel, declaraţia

char tab_1 [ ] = "Salut", tab_2 [10] = "Salutare";

este echivalentă cu

char tab_1 [6] = { 'S', 'a', 'l', 'u', 't', '\0'};
char tab_2 [10] = { 'S', 'a', 'l', 'u', 't', 'a', 'r', 'e', '\0', '\0'};

Funcţiile de prelucrare a şirurilor de caractere se grupează în următoarele categorii:

- de citire / scriere a şirurilor (declarate în stdio.h);
- de comparare, copiere, concatenare, iniţializare a şirurilor (declarate în string.h);
- de transformare a şirurilor (declarate în stdlib.h).

Pentru scrierea, respectiv citirea şirurilor de caractere, în limbajul C se utilizează funcţiile **puts** şi respectiv **gets**. Acestea sunt definite în fişierul antet stdio.h şi au ca unic parametru un şir de caractere, care poate fi o constantă sau o variabilă. O constantă de tipul şir de caractere este o succesiune de caractere delimitată de o pereche de ghilimele, în timp ce o variabilă de tipul şir de caractere este un nume de tablou unidimensional de tipul caracter.

În continuare sunt prezentate câteva exemple de funcţii aflate în biblioteca string.h care acţionează asupra şirurilor terminate în maniera standard, cu un caracter nul.

- char *strcat (char *s1, const char * s2); concatenează *s2* la *s1* şi întoarce *s1*;
- char * strchr (const char *s, int c); caută prima apariţie a lui *c* în şirul de caractere *s* şi întoarce poziţia lui sau NULL;
- char strcmp(const char *s1, const char *s2); compară şirurile de caractere *s1* şi *s2* şi întoarce <0, =0, sau >0 după cum *s1*<*s2*, *s1*=*s2* sau *s1*>*s2*;
- char *strcpy(char *s1, const char *s2); copiază şirul *s2* în *s1* şi întoarce *s1*;
- size_t strlen (const char *s); întoarce numărul de caractere din şirul *s*, inclusiv terminatorul;

- char *strstr (const char *s1, const char *s2); caută în *s1* un subşir egal cu *s2* şi întoarce adresa lui.

Exemplul 9.11.

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

void main (void)
{
        char nume [50];
        clrscr();
        puts("Numele>");
        gets(nume);
        printf("Salut %s, Numele tău are %3d caractere", nume,
strlen(nume));
        getch();
}
```

La începutul programului este declarat tabloul *nume*, de tipul caracter, în care se va memora şirul de caractere ce reprezintă numele utilizatorului. În general, pentru memorarea numelui se vor utiliza numai o parte din cei 50 de octeţi ai tabloului *nume*. Deoarece nu există restricţii de accesare a memoriei, atunci când declarăm o variabilă şir de caractere (un tablou de tip caracter), aceasta trebuie dimensionată la valoarea maximă a lungimii şirurilor pe care anticipăm că i le vom atribui, plus un octet pentru secvenţa escape '\0' adăugată automat. Astfel, dacă în exemplul 9.11. declarăm variabila *nume[10]*, atunci ultimele caractere ale numelui Popescu Ion, citit de la tastatură, se suprapun peste o zonă de memorie utilizată şi rezultatul execuţiei programului este imprevizibil.

Într-un şir de caractere, care este un tablou, fiecare caracter poate fi accesat fie individual prin intermediul indicilor, fie prin intermediul pointerului asociat şi al mecanismului de indirectare.

Exemplul 9.12.
Corectarea unei litere introduse greşit de la tastatură într-un şir de caractere.

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

void main (void)
{
        char fraza[81], litera, *p_sir;
        int poz ;
        clrscr() ;
        printf(" \nIntroduceţi o frază:");
        gets(fraza);
        printf("Introduceţi litera greşită");
        litera = getche();
        p_sir = strchr(fraza, litera);
        poz = p_sir – fraza;
        strcpy(&fraza[poz], &fraza[poz+1]);
        printf("\nFraza corectă:");
        puts(fraza);
        getch();
}
```

Programul solicită introducerea unei fraze şi se consideră că, din greşeală, în corpul frazei s-a tastat o literă în plus care trebuie eliminată. Pentru aceasta, se solicită tastarea literei greşite, care este citită cu ajutorul funcţiei getche() şi memorată în variabila *litera.* Pentru a depista poziţia acesteia în şirul de caracterese foloseşte funcţia strchr. Funcţia primeşte ca argumente şirul în care se face căutarea şi caracterul urmărit, returnând un pointer ce reprezintă adresa primei apariţii a caracterului în şir. Poziţia acestuia se determină scăzând din pointerul returnat valoarea adresei de început a şirului furnizată de numele acestuia. Astfel, prin instrucţiunea *poz = p_sir – fraza;* se scad doi pointeri şi se obţine o valoare întreagă ce reprezintă indicele caracterului căutat.

În numeroase situaţii este necesară stocarea şi manevrarea mai multor şiruri de caractere, cum ar fi o listă de nume. Pentru aceasta, se folosesc tablourile de tipul şir de caractere, care, de fapt, constituie matrice (tablouri bidimensionale) de tipul caracter.

Exemplul 9.13.
Memorarea unei liste de nume şi afişarea acesteia în ordine alfabetică, utilizând un tablou de pointeri la caracter şi un tablou de tip şir de caractere.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 30

void main (void)
{
        char nume[MAX] [50], *p_nume[MAX], *min_nume;
        int nr_nume=0, i, j, ind_min;
        clrscr();
        while (nr_nume < MAX)
        {
                printf("Numele %2d:  ", nr_nume + 1);
                gets(nume[nr_nume]);
                if (strlen(nume[nr_nume]) = = 0)
                        break;
                p_nume[nr_nume++]= nume[nr_nume];
        }
        /* ordonare alfabetică*/
        for(i=0; i<nr_nume-1; i++)
        {
                ind_min=i;
                min_nume=p_nume[i];
                for(j=i+1; j<nr_nume; j++)
                        if (strcmp(p_nume[j], min_nume) <0)
                        {
                                ind_min = j;
                                min_nume=p_nume[j];
                        }
                if(ind_min !=i)
                {
                        p_nume[ind_min]=p_nume[i];
                        p_nume[i]=min_nume;
                }
        }
        /* afişare lista ordonată*/
        puts("Lista ordonată alfabetic:");
```

```
        for(i=0; i<nr_nume-1; i++)
                printf("\nNumele %2d :  %s", i+1, p_nume[i]);
        getch();
}
```

În cadrul programului s-a considerat că lungimea maximă a listei de nume este 30. Valoarea poate fi foarte uşor modificată prin schimbarea valorii atribuite constantei MAX din directiva #define.

Numele ce alcătuiesc lista iniţială sunt citite în cadrul unei bucle while care se termină fie în momentul în care valoarea variabilei *nr_nume* a atins valoarea maximă, fie în momentul în care lungimea şirului de caractere introdus de la tastatură este nulă. Solicitarea terminării forţate a introducerii de nume în listă (apăsarea tastei <Enter>) este detectată prin intermediul instrucţiunii if care are în expresia logică apelul funcţiei strlen.

Deşi *nume* este un tablou bidimensional, în cadrul programului este utilizat folosind un singur indice. Acest lucru are loc deoarece, prin utilizarea unui singur indice asociat cu numele unui tablou bidimensional, este posibilă tratarea liniilor acestuia ca elemente ale unui tablou unidimensional. Astfel, *nume[i]* este pointerul la şirul de caractere ce va fi memorat pe linia *i* a matricei *nume*.

Instrucţiunea gets(nume[nr_nume]); va prelua de la tastatură numele cu numărul de ordine *nr_nume* +1 şi îl va depune pe linia *nr_nume* din matrice, iar instrucţiunea p_nume[nr_nume++]= nume[nr_nume]; atribuie elementului *p_nume[nr_nume]* din tabloul de pointeri *p_nume* valoarea adresei liniei *nr_nume*. După atribuire, valoarea contorului este incrementată folosind operatorul ++.

Algoritmul de ordonare foloseşte în expresia logică a instrucţiunii de decizie care determină elementul minim funcţia strcmp. Pentru ordonare este utilizat tabloul de pointeri *p_nume* ce conţine adresele numelor, şi nu matricea *nume*. Astfel, pentru fiecare valoare a lui *i* din bucla exterioară se determină, în cadrul buclei interioare, elementul minim din şirurile de caractere indicate de *p_nume[i], p_nume[i+1],..,p_nume[nr_nume]* precum şi poziţia acestuia. Aceste informaţii sunt memorate în variabilele *min_nume* şi *ind_min*.

La sfârşitul buclei interioare, dacă este cazul, se modifică conţinutul şirului de pointeri, schimbând între ele valorile *p_nume[i]* şi *min_nume* .

După terminarea procesului de ordonare este modificat doar tabloul pointerilor la liniile matricei, astfel că acestea sunt accesate în ordine alfabetică.

```

# 10
# Structuri şi uniuni

Pentru memorarea datelor s-au folosit până acum variabile simple şi tablouri. Acestea sunt limitate de faptul că permit tratarea numai a datelor de acelaşi tip. Situaţiile în care este necesară manevrarea împreună a datelor de tipuri diferite, care să poată fi tratate fie individual, fie ca o entitate pot fi rezolvate cu ajutorul structurilor.

O altă situaţie specială care poate apare este aceea în care este necesară utilizarea unei aceleaşi zone de memorie de către variabile de tipuri diferite. De exemplu, este posibil ca la începutul unui program o anumită zonă de memorie să fie accesată prin intermediul unei variabile de tip întreg, pentru ca, ulterior, aceeaşi zonă de memorie să fie accesată cu ajutorul unei variabile de tip real. Acest mecanism este posibil prin utilizarea uniunilor.

## 10.1. Structuri

O structură este o colecţie de valori eterogene stocate într-o zonă de memorie compactă. Structurile ajută la organizarea datelor complexe (de regulă din programe mari) permiţând tratarea unitară a unui grup de variabile.

Elementele sau variabilele menţionate într-o structură se numesc membri ai structurii. Un membru de structură, o etichetă sau o variabilă oarecare ce nu aparţine structurii pot avea acelaşi nume fără a genera conflicte, deoarece ele vor fi întotdeauna deosebite una de alta din context.

Structurile pot fi imbricate, caz în care o structură include o altă structură, şi autoreferite, caz în care structura conţine o referinţă la o structură de acelaşi tip.
```

10.1.1. Definirea structurilor

Forma generală de definire a unei structuri este:

```
struct   nume_structură
         {
                  elementele structurii
         };
```

şi constă din:

- cuvântul cheie **struct** urmat de numele structurii;
- o pereche de acolade între care sunt declarate elementele structurii;
- caracterul ";" care marchează sfârşitul instrucţiunii de definire a structurii.

Trebuie reţinut faptul că numele care urmează cuvântului cheie struct în instrucţiunea de definire nu este un nume de variabilă, ci numele unui tip de date şi se mai numeşte şi eticheta structurii.

În cadrul unui program se poate utiliza un număr oarecare de instrucţiuni de definire de structuri, ele putând fi plasate la începutul funcţiei în cadrul căreia se utilizează noile tipuri de date, înaintea declaraţiilor de variabile. În cazul în care fişierul sursă conţine mai multe funcţii în cadrul cărora dorim să utilizăm noile tipuri de date, instrucţiunile de definire a structurilor vor fi plasate la începutul fişierului, în afara oricărei funcţii. Este recomandată plasarea structurilor într-un fişier antet şi includerea acestuia în fişierul sursă cu ajutorul directivei include.

Exemplul 10.1.

Un program care foloseşte o structură pentru a memora informaţiile referitoare la un candidat la un concurs de admitere. Datele care compun structura vor fi:

- un şir de caractere pentru memorarea numelui candidatului,
- un întreg pentru memorarea numărului legitimaţiei de concurs,
- trei reali pentru a memora rezultatele obţinute la cele două probe de concurs şi media finală de admitere.

```
#include <stdio.h>
#include<conio.h>
#include <string.h>
void main (void)
{
        struct candidat
```

```
        {
                char nume[80];
                int nr_leg ;
                float nota1 ;
                float nota2 ;
                float media ;
        } ;
        struct candidat x ;
        strcpy(x.nume, ”Georgescu G. George”);
        x.nr_leg = 444;
        x.nota1=7.32;
        x.nota2= 8.59 ;
        x.media=(x.nota1+x.nota2)/2 ;
}
```

Exemplul 10.1. pune în evidenţă trei aspecte ale utilizării structurilor: definirea lor, declararea variabilelor de tip structură şi accesarea elementelor structurii. Definirea structurii am discutat-o mai devreme. Celelalte două aspecte urmează să le abordăm în continuare.

10.2. Declararea variabilelor de tip structură

Ca orice altă variabilă, şi variabila de tip structură trebuie să fie declarată înainte de a fi utilizată. Limbajul C permite comprimarea într-o singură instrucţiune a instrucţiunilor de definire a structurii şi de declarare a variabilelor de tip structură.

Forma generală a unei instrucţiuni de declarare a variabilelor de tip structură este:

```
struct   nume_structură
        { listă_câmpuri} listă_variabile;
```

şi constă din:

- cuvântul cheie *struct* urmat de numele structurii,
- *listă_câmpuri* formată din elemente cu structura *tip_câmp declarator_câmp*,
- *lista variabilelor* formată din nume de variabile separate prin virgulă.

Exemplul 10.2.

În exemplul 10.1 instrucţiunea *struct candidat x ;* poate fi eliminată rescriind instrucţiunea de definire a structurii sub forma:

```
struct candidat
        {
                char nume[80];
                int nr_leg ;
                float nota1, float nota2, media ;
        } x;
```

Nu este obligatoriu ca orice tip de structură să aibă un nume. Pot apare structuri anonime, dar în acest caz trebuie ca declaraţia să conţină cel puţin un identificator de variabilă.

În funcţie de contextul utilizării, din declaraţie pot lipsi atât *listă_câmpuri*, cât şi *listă_variabile,* însă *nume_structură* şi *listă_câmpuri* nu pot lipsi simultan.

Câmpurile unei structuri se declară după sintaxa obişnuită a declaraţiilor de variabile, dar nu pot avea valori iniţiale. În plus, sintaxa este extinsă,. Permiţând specificarea, prin intermediul unei expresii constante, a numărului de biţi alocat pentru un anumit câmp, sub forma:

tip_câmp selector: număr_biţi;

O astfel de componentă, pentru care se specifică numărul de biţi alocat, se numeşte **câmp de biţi**. Câmpurile de biţi nu pot să apară decât în interiorul unei structuri sau uniuni, iar tipul lor nu poate fi decât **signed** sau **unsigned int**. Se poate omite specificarea tipului de date şi a selectorului câmpului de biţi; construcţia este utilă pentru a descrie un câmp rezervat, dar neutilizat. O convenţie specială tratează cazul în care numărul de biţi este 0: un câmp de biţi de lungime 0 nu poate avea nuci tip de date, nici selector, iar prezenţa lui impune alinierea următorului câmp de biţi la cea mai apropiată graniţă de cuvânt, aceasta fiind dependentă de implementare.

Exemplul 10.3.

```
Struct
        {
                signed char int1;
                unsigned char int2;
                unsigned a: 2;
                          : 3;
                unsigned b: 5;
        } registru_auxiliar;
```

Variabila *registru_auxiliar* conţine un întreg cu semn reprezentat pe 1 octet, un întreg fără semn reprezentat tot pe 1 octet, un întreg fără semn reprezentat pe 2 biţi, 3 biţi liberi şi un întreg cu semn reprezentat pe 6 biţi.

Modul de reprezentare a biţilor în cadrul unui cuvânt şi lungimea unui cuvânt sunt dependente de implementare. De aceea, utilizarea câmpurilor de biţi nu este portabilă. Ele au fost introduse în limbaj pentru a permite descrierea diferitelor registre hardware gestionate de programe sistem.

Câmpurile de biţi pot fi folosite în orice context în care este permisă apariţia unei variabile, cu singura restricţie că nu li se poate aplica operatorul de calcul al adresei **&**.

Numele de structuri se află într-un spaţiu de nume separat de cel al numelor de variabile, de aceea putem avea simultan o variabilă şi o structură cu acelaşi nume. Ele pot fi deosebite de către compilator, deoarece numele structurii apare întotdeauna după cuvântul cheie **struct**. Selectorii câmpurilor unei structuri se află într-un spaţiu de nume specific acelei structuri, astfel încât apariţia de câmpuri cu acelaşi nume în două structuri diferite nun constituie o eroare, compilatorul putând întotdeauna decide dacă este vorba de un câmp al unei structuri sau al alteia.

10.3. Operaţii permise asupra structurilor

Operaţia principală care poate fi efectuată asupra unei variabile de tip structură este selectarea unui câmp, utilizând **operatorul de selecţie** (**operatorul membru**) ".", conform sintaxei:

variabilă_structură . selector

Câmpul selectat se comportă ca o variabilă şi i se pot aplica toate operaţiile care se pot aplica variabilelor de acel tip. Deoarece structurile se prelucrează frecvent prin intermediul pointerilor, a fost introdus un operator special, care combină operaţiile de indirectare şi selectare a unui câmp, anume "->". Expresia p -> selector este interpretată de compilator la fel ca expresia **(*p).selector**.

Întotdeauna, unei variabile de tip structură i se pot aplica operatorii **&** (calcul al adresei) şi **sizeof** (calculul mărimii zonei de memorie ocupate de variabilă).

Sunt permise următoarele operaţii:

- unei variabile de tip structură i se poate atribui valoarea unei variabile de acelaşi tip structură;
- variabilele de tip structură pot fi transmise ca parametri funcţiilor;

- o funcţie poate avea ca rezultat o valoare de tip structură.

Exemplul 10.4.
Se citesc de la tastatură informaţiile referitoare la candidaţii la concursul de admitere, se calculează media şi se afişează rezultatul sub forma unui mesaj de tipul *admis* sau *respins*.

```
#include <stdio.h>
#include<conio.h>
#include <string.h>
void main (void)
{
        struct candidat
        {
                char nume[80];
                int nr_leg ;
                float nota1, float nota2, media ;
        } x;
        while (1)
        {
                clrscr();
                printf("Numele candidatului :");
                gets(x.nume);
                if (strlen(x.nume) = = 0)
                        exit(0);
                printf("Numărul legitimaţiei de concurs:");
                scanf("%d", &x.nr_leg);
                printf("Notele obţinute: ");
                scanf("%f%f", &x.nota1,&x.nota2);
                x.media=(x.nota1+x.nota2)/2 ;
                if (x.media >= 5)
                        printf("%s admis", x.nume);
                else
                        printf("%s respins", x.nume);
                getch();
        }

}
```

În prima parte a programului, în cadrul unei instrucţiuni compuse, este definită structura *candidat* şi este declarată o variabilă *x* de tipul structură candidat. Datele sunt citite de la tastatură folosind funcţiile *gets* şi

scanf în cadrul unei bucle infinite. Instrucţiunile sunt similare cu cele folosite pentru citirea şi scrierea variabilelor simple, deosebirea constând în utilizarea operatorului **punct** pentru a putea accesa elementele variabilei *x*. Astfel, prin expresia *x.nume* este accesat primul element al structurii, care constituie pointerul la şirul de caractere în care se memorează numele candidatului. Prin instrucţiunea *gets(x.nume);* este preluat de la tastatură numele candidatului şi depus în structură.

Pentru a citi valori cu ajutorul funcţiei scanf, acesteia trebuie să-i transmitem în lista parametrilor adresele variabilelor în care se vor memora valorile. Şi în cazul structurilor, adresa unui element se obţine folosind tot operatorul de adresă & plasat în faţa expresiei ce desemnează membrul structurii.

Rezultatele obţinute sunt afişate cu ajutorul funcţiei printf.

Terminarea programului se realizează prin apăsarea tastei *<Enter>,* la solicitarea introducerii numelui candidatului, care semnifică introducerea unui şir de caractere de lungime 0. Condiţia respectivă este testată prin intermediul instrucţiunii de decizie if, care foloseşte în expresia logică funcţia **strlen**, având ca parametru elementul *nume* al variabilei x de tip structură.

10.4. Iniţializarea structurilor

Dintre facilităţile oferite de limbajul C, vor fi analizate în continuare utilizarea unei structuri ca element al altei structuri şi iniţializarea structurilor.

Exemplul 10.5.

```
#include <stdio.h>
#include<conio.h>
struct membru
        {
                char nume[80];
                int nr_ore_zi;
                float sal_ora;
        }
struct echipa
        {
                struct membru sef;
                struct membru munc1;
                struct membru munc2;
                int nr_zile;
```

```
}

void main (void)
    {
        struct echipa e1 =
            { {"Popescu Viorel", 8, 425.50},
              {"Ionescu Dan   ", 6, 375.25},
              {"Popa Eugen    ", 7, 315.30}, 4
            };
        float suma;
        clrscr();
        printf("Sumele cuvenite membrilor echipei");
        suma = e1.sef.nr_ore * e1.sef.sal_ora * e1.sef.nr_zile;
        printf("\n 1. %s%12.2f lei",e1.sef.nume, suma);
        suma = e1.munc1.nr_ore * e1.munc1.sal_ora *
    e1.munc1.nr_zile;
        printf("\n 2. %s%12.2f lei",e1.muinc1.nume, suma);
        suma = e1.munc2.nr_ore * e1.munc2.sal_ora *
    e1.munc2.nr_zile;
        printf("\n 3. %s%12.2f lei",e1.munc2.nume, suma);
        getch();
    }
```

Programul calculează sumele cuvenite membrilor unei formaţii de lucru constituită din trei persoane (şef de echipă plus doi muncitori) care efectuează o anumită lucrare într-un număr de zile. Pentru realizarea acestor cerinţe, în prima parte a programului sunt definite două structuri. Prima structură, numită *membru*, este destinată memorării informaţiilor referitoare la un membru al echipei. Ea conţine numele persoanei (un şir de caractere), numărul de zile lucrate (un întreg) şi salariul pe oră lucrată (un real). A doua structură, numită *echipa* , conţine trei variabile de tip *structură membru* destinate memorării informaţiilor referitoare la fiecare membru al echipei şi o variabilă de tip întreg *nr_zile* în care se memorează numărul de zile în care s-a efectuat lucrarea.

Cele două structuri au fost definite în afara funcţiei *main*, dar definirea se poate face la fel de bine şi în interiorul acesteia.

Prima instrucţiune a funcţiei principale defineşte variabila *e1* de tipul *structură echipă* şi îi atribuie un set de valori iniţiale. După cum se observă, iniţializarea unei structuri este similară cu cea a unui tablou. Valorile care urmează să fie atribuite variabilelor care sunt membre ale structurii sunt plasate în interiorul unei perechi de acolade şi sunt separate

prin virgulă. Este strict necesară păstrarea corespondenţei dintre ordinea în care au fost declaraţi membrii structurii şi ordinea în care sunt plasate valorile acestora în interiorul acoladelor. In cazul variabilelor de tipul structuri înlănţuite se folosesc mai multe perechi de acolade pentru precizarea valorilor iniţiale.

Ca urmare a instrucţiunii de iniţializare, deoarece variabila *şef* este primul membru al variabilei *e1*, membrii ei vor primi valorile conţinute în prima pereche de acolade interioare. Şeful de echipă se numeşte Popescu Viorel, a lucrat câte 8 ore pe zi şi are un salariu de 425.50 lei pe oră.

Similar, membrii variabilelor *munc1* şi *munc2* vor primi valorile conţinute în interiorul următoarelor perechi de acolade, iar variabila de tip întreg *nr_zile* va primi valoarea 4.

Pentru accesarea elementelor unei structuri se foloseşte operatorul punct, care are rolul de a conecta o variabilă de tip structură cu elementele sale. În cazul structurilor care au ca elemente alte structuri, se foloseşte o expresie de forma

struct1.struct2.membru_struct2

în care *membru_struct2* este un element al unei structuri *struct2*, care la rândul ei este membru al altei structuri *struct1*.

10.5. Tablouri de structuri

În cadrul acestui paragraf este prezentat modul de declarare şi de utilizare a tablourilor de structuri, operaţia de atribuire cu variabile de tip structură şi utilizarea structurilor ca parametri ai funcţiilor.

Exemplul 10.6.

```
#include <stdio.h>
#include<conio.h>
#include <string.h>
#define MAX_CAND 200
candidat citire(int)
void main (void)
{
        struct candidat
        {
                char nume[80];
                int nr_leg ;
                float nota1, float nota2, media ;
        };
```

```
        candidat tab[MAX_CAND];
        candidat cand_x;
        int nr_cand, i;
        clrscr();
        printf("Introduceţi numărul de candidaţi:");
        scanf(„%d", &nr_cand);
        for (i=0; i<=nr_cand; i++)
                {
                        cand_x = citire(i);
                        tab[i] = cand_x;
                        if (tab[i].media >=5)
                                printf("%s admis", tab[i].nume);
                        else
                                printf("%s respins", tab[i].nume);
                        getch();
                }
}
 candidat citire (int i)
{
        candidat x ;
        clrscr() ;
        printf("DATELE CANDIDATULUI  %d\n",i+1) ;
        printf("Numele candidatului :");
        gets(x.nume);
        printf("Numărul legitimaţiei de concurs:");
        scanf("%d", &x.nr_leg);
        printf("Notele obţinute: ");
        scanf("%f%f", &x.nota1,&x.nota2);
        x.media=(x.nota1+x.nota2)/2 ;
        return(x);
}
```

Programul constituie o variantă îmbunătăţită a programului din exemplul 10.4. , în sensul că informaţiile referitoare la candidaţi sunt memorate într-un tablou de structuri. El permite citirea şi memorarea datelor referitoare la un număr maxim de 200 candidaţi. Capacitatea de memorare poate fi modificată simplu, prinj înlocuirea valorii constantei MAX_CAND din directiva de definire.

Instrucţiunea *candidat tab[MAX_CAND];* declară variabila *tab* ca fiind un tablou unidimensional ale cărui elemente sunt variabile de tipul *structură candidat.* Declararea tablourilor de structuri se realizează în

aceeaşi manieră în care sunt declarate tablourile de variabile simple, adică prin instrucţiuni de forma:

struct nume_structură nume_tablou [n];

în cazul tablourilor unidimensionale, respective de forma:

struct nume_structură nume_tablou [n][m];

în cazul tablourilor bidimensionale.

În cazul în care structura a fost redenumită, în locul perechii *struct nume_structură* se va utiliza noul cuvânt atribuit ca identificator al structurii. În exemplul prezentat, pentru declararea tabloului se poate utiliza instrucţiunea *struct candidat tab[MAX_CAND];* dar s-a preferat varianta mai scurtă *candidat tab[MAX_CAND];*

Pentru citirea datelor, programul foloseşte funcţia *citire*. Aceasta are ca parametru un întreg reprezentând numărul de ordine al candidatului în lista candidaţilor şi returnează variabila *x* de tipul *structură candidat*. Acesta este motivul pentru care funcţia a fost declarată de tipul *candidat*.

O facilitate oferită de limbajul C o constituie atribuirea unei variabile de tipul structură altei variabile de acelaşi tip. În urma unei operaţii de atribuire cu structuri, valorile membrilor variabilei structură aflate în partea dreaptă a semnului egal sunt atribuite membrilor variabilei de tip structură din stânga operatorului de atribuire. Instrucţiunea *cand_x = citire(i);* atribuie valorile variabilei structură candidat *x*, returnată de funcţia de citire, variabilei *cand_x* de acelaşi tip, iar instrucţiunea *tab[i] = cand_x;* atribuie valorile variabilei *cand_x* elementului *i* din tabloul de structuri *tab*. Cele două instrucţiuni pot fi înlocuite printr-o singură instrucţiune de atribuire de forma:

tab[i] = citire(i);

Pentru a decide dacă un candidat a obţinut media minimă de promovare se utilizează expresia *tab[i].media* în cadrul instrucţiunii de decizie. Aceasta accesează membrul *media* al elementului *i* din tabloul de structuri *tab*.

10.6. Uniuni

Din punct de vedere sintactic, uniunile seamănă cu structurile (declaraţia unei uniuni se face la fel ca declaraţia unei structuri, înlocuind cuvântul cheie **struct** cu **union**). Deosebirea dintre ele constă în aceea că, în timp ce, în cazul unei structuri, fiecărui câmp i se alocă spaţiu de memorie (mărimea zonei de memorie ocupată de structură fiind suma mărimilor zonelor ocupate de câmpurile componente), în cazul uniunilor nu se alocă decât atâta memorie cât este necesară pentru memorarea câmpului de

dimensiune maximă. În acest fel este posibil ca o aceeaşi zonă să fie interpretată în mai multe feluri deosebite. Mecanismul este necesar programelor de sistem care au nevoie să mânuiască valorile unui tip de date ca şi cum ar fi de alt tip.

Fie următoarele declaraţii:

```
struct a
        {
                int i;
                char c;
                float f;
        };
union u
        {
                int i;
                char c;
                float f;
        };
```

Mărimea spaţiului de memorie ocupat de o variabilă de tip *struct s* va fi de sizeof(struct s) = 7 octeţi, pe când dimensiunea spaţiului de memorie ocupat de o variabilă de tip *union u* va fi de sizeof(union u) = 4 (dimensiunea celui mai mare câmp, în cazul de faţă *f*).

Accesul la un câmp al unei uniuni se face tot cu ajutorul operatorului punct, ca şi în cazul structurilor. La atribuirea unei valori către un câmp al unei uniuni sunt afectate toate câmpurile uniunii.

Numele de uniuni împarte acelaşi spaţiu de nume cu numele de structuri şi cu numele de enumerări, separat de numele de variabile. Rezultă că putem avea o variabilă şi o uniune cu acelaşi nume, fără pericol de confuzie. Numele câmpurilor unei uniuni sunt valabile numai în interiorul uniunii, fiind invizibile în exterior; ca urmare, putem avea câmpuri omonime în două uniuni diferite, fără pericol de confuzie. La fel ca şi structurile, uniunile pot fi anonime; în acest caz este obligatorie prezenţa unui identificator de variabilă, pentru ca declaraţia să fie validă.

Iniţializarea uniunilor se face cu o singură valoare (expresie constantă), închisă între acolade, care iniţializează primul câmp declarat:

```
union u
        {
                int i;
                char c;
                float f;
        }= {15};
```

10.7. Uniuni de structuri

O structură poate fi element membru al altei structuri, dar poate fi şi element membru al unei uniuni.

Exemplul 10.7.

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

void main (void)
{
        struct octet
        {
                        har octet1;
                char octet2;
        };
        union întreg
        {
                struct octet val;
                int i_val;
        } x;
        clrscr();
        printf(”Dimensiunea în octeţi a variabilei x este %1d”, sizeof(x));
        x.val.octet1=5;
        x.val.octet2=1;
        printf(”\nValoarea conţinută în primul octet :     %d”, x.val.octet1);
        printf(”\nValoarea conţinută în al doilea octet :     %d”, x.val.octet2);
        printf(”\nValoarea accesată de i_val :     %d”, x.i_val);
        getch();
}
```

Mai întâi estedefinită structura *octet*, care are ca elemente variabile de tipul caracter *octet1* şi *octet2.* Apoi, această structură este utilizată pentru a declara variabila *val* ca membru al uniunii *întreg* alături de variabila de tipul întreg *i_val*. Deoarece fiecare membru al uniunii ocupă câte doi octeţi, dimensiunea zonei de memorie alocată variabilei *x* de tipul

union întreg va fi de 2 octeţi. Acest fapt este confirmat de rezultatul afişat de primul apel al funcţiei **printf**.

Prin operaţiile de atribuire :

x.val.octet1=5;
x.val.octet2=1;

în primul octet se va depune valoarea 1, iar în al doilea valoarea 5. Deoarece prin intermediul elementelor structurii sunt accesate zone de memorie diferite, valorile 1 şi 5 se găsesc simultan în memorie şi pot fi interpretate ca elemente componente ale unei valori întregi care ocupă doi octeţi. Primul dintre aceştia se numeşte **octetul mai puţin semnificativ**, iar al doilea se numeşte **octetul cel mai semnificativ**. Având în vedere reprezentarea binară a valorilor 1 şi 5 (00000001 şi respectiv 00000101), valoarea accesată de *x.i_val* va fi 261, fapt confirmat şi de ultimul apel al funcţiei **printf**, care are în lista de variabile pe *x.i_val*.

Din exemplul prezentat se constată că, pentru a accesa elementele membre ale unei structuri care la rândul ei este un element membru al unei uniuni, se foloseşte de două ori operatorul punct.

De menţionat că procesul de înlănţuire este general, în sensul că aşa cum o structură poate fi element membru al altei structuri sau uniuni, o uniune poate fi la rândul ei element membru al unei structuri sau altei uniuni.

```

# 11
# Fişiere

În majoritatea aplicaţiilor apare necesitatea păstrării informaţiilor şi după terminarea execuţiei programului. Informaţiile obţinute în urma execuţiei programului pot fi stocate (memorate) fie pe disc, fie pe dischetă , fie pe CD-ROM, în vederea unor prelucrări ulterioare.

Unitatea de memorare utilizată în acest scop, constând dintr-o succesiune de octeţi cărora li se asociază un nume este **fişierul**.

Numele fişierului este format din două părţi:

- prima parte, formată dintr-un şir de maximum 8 caractere, conţine numele propriu-zis al fişierului;
- a doua parte, numită extensie, constă dintr-un şir de maximum 3 caractere şi este separată de numele propriu-zis prin caracterul punct.

Prin intermediul unui set complet de funcţii standard, limbajul C oferă utilizatorilor modalităţi de programare a operaţiilor de citire şi scriere a fişietrelor sub controlul sistemului de operare.

Sistemul standard de intrare / ieşire permite efectuarea transferului de date între program şi următoarele patru tipuri de fişiere:

- **fişiere caracter**, pentru care datele sunt citite sau scrise caracter cu caracter, similar cu citirea unui caracter de la tastatură sau scrierea acestuia pe ecran;
- **fişiere şir de caractere**, pentru care datele sunt citite sau scrise sub forma şirurilor de caractere, similar modului în care un şir de caractere este citit de la tastatură sau afişat pe ecran;
- **fişiere formatate**, pentru care datele citite sau scrise formează colecţii mixte de caractere şi valori numerice întregi sau reale transferate în conformitate cu descriptorii de format identici celor folosiţi în cadrul funcţiilor **scanf** şi **printf**;
- **fişiere înregistrare**, pentru care datele sunt citite sau scrise sub forma unor blocuri de lungime fixă numite **înregistrări**. În mod uzual, o înregistrare conţine fie elementele unui tablou, fie ale unei structuri.
```

În funcţie de modul în care informaţia este reprezentată în cadrul fişierelor, acestea pot fi grupate în următoarele două categorii:

- **fişiere text**, în care informaţiile sunt reprezentate sub forma unui text, pentru fiecare caracter fiind utilizat un octet. Acest tip de fişiere au avantajul că pot fi vizualizate şi modificate cu ajutorul editoarelor de text, inclusiv cu ajutorul editorului din mediul integrat TURBO C++/ BORLAND C. Au însă dezavantajul că, în cazul memorării datelor de tip numeric, fiind folosit un octet pentru fiecare cifră, necesită un spaţiu mare de memorare.
- **fişiere binare**, în care informaţiile sunt memorate în formă binară, similar cu memorarea valorilor variabilelor în memoria internă. Avantajul acestui tip de fişiere constă în spaţiul mult mai mic necesar pentru stocarea informaţiilor. Dezavantajul constă în faptul că nu pot fi vizualizate sau modificate cu editoarele de text.

Având clasificările de mai sus, se poate face precizarea că modurile **caracter, şir** şi **formatat** sunt destinate în general prelucrării fişierelor de tip **text**, iar modul **înregistrare** este destinat prelucrării fişierelor de tip **binar**.

Înainte de a fi citit sau scris, un fişier trebuie deschis. Deschiderea unui fişiere realizează prin apelul funcţiei **fopen**. Aceasta are ca parametri două şiruri de caractere prin intermediul cărora se transmit sistemului de operare numele fişierului şi modul în care acesta urmează să fie accesat şi returnează un pointer la o structură de tip **FILE**. Valoarea pointerului este furnizată de sistemul de operare şi poate fi 0 (NULL), dacă operaţia de deschidere a eşuat, sau o valoare diferită de 0, în cazul în care deschiderea fişierului a reuşit. Structura **FILE** este definită în fişierul antet **stdio.h** şi conţine variabile în care se memorează informaţiile referitoare la fişier.

Pentru deschiderea unui fişier în cadrul unui program C se foloseşte următoarea secvenţă de instrucţiuni:

```
FILE *p_fişier;
…
p_fişier = fopen(nume_fişier, mod_acces);
if (p_fişier = = NULL)
{
…
}
```

La terminarea operaţiilor de scriere – citire a unui fişier, acesta trebuie închis. Pentru aceasta se foloseşte funcţia **fclose**, care are ca parametru pointerul de tipul structură **FILE** utilizat la deschiderea

fişierului. Astfel, prin instrucţiunea **fclose(p_fişier);** se solicită sistemului de operare închiderea fişierului având ca pointer variabila *p_fişier*.

11.1. Scrierea şi citirea fişierelor de tipul caracter

Pentru exemplificarea modului în care sunt efectuate operaţiile de scriere şi citire a fişierelor în modul caracter, vom analiza programul din exemplul 11.1.

Exemplul 11.1.

Programul citeşte de la tastatură o succesiune de caractere pe care o scrie într-un fişier. Ulterior, conţinutul fişierului este analizat pentru a determina câte caractere au fost introduse şi câte dintre acestea sunt cifre. Sunt utilizate pentru obţinerea acestor acţiuni funcţiile **scrie_c** şi **citeşte_c**.

```
#include <stdio.h>
void scrie_c (char*);
void citeşte_c(char *);


void main (argc, argv)
{
        int argc;
        char *argv[];

        if (argc!=2)
        {
                printf("\nEroare ! Tastaţi: nume_program     nume_fişier");
                exit(1);
        }
        scrie_c(argv[1]);
        citeşte_c(argv[1]);
}

void scrie_c (char *nume _fişier);
{
        char car;
        FILE *fis;      /* pointer la structura fişier*/
        fis = fopen(nume _fişier, "w");         /* deschide fişierul*/
```

```
        if (fis = = NULL)       /* testare deschidere reuşită*/
        {
        printf("\nEroare! Fişierul %s nu poate fi deschis", nume _fişier);
        exit(1);
        }
        /* citirea caracterelor de la tastatură şi scrierea lor în fişier*/
        clrscr();
        printf("Tastaţi succesiunea de caractere:") ;
        while ((car = getche()) != 13)
                putc(car, fis);
        fclose(fis);
}

void citeşte_c(char * nume _fişier)
{
        int car, nr_car = 0, nr_cifre = 0;
         FILE *fis;      /* pointer la structura fişier*/
        fis = fopen(nume _fişier, "w");         /* deschide fişierul*/
        if (fis = = NULL)       /* testare deschidere reuşită*/
        {
        printf("\nEroare! Fişierul %s nu poate fi deschis", nume _fişier);
        exit(1);
        }
        /* citirea caracterelor din fişier*/
        clrscr();
        printf("\nConţinutul fişierului :");
        while ((car = getche(fis)) != EOF)
        {
                nr_car ++;
                printf("%c",car);
                if(car>=48 && car <= 57) nr_cifre ++;
        }
        printf("\nNumărul total de caractere %d\nNumărul de cifre %d",
nr_car, nr_cifre);
        fclose(fis);
}
```

Se remarcă utilizarea variabilelor *argc* şi *argv* ca parametri ai funcţiei main. În mod uzual, pentru lansarea în execuţie a unui program se tastează , în continuarea mesajului sistemului de operare **MS-DOS** conţinând directorul curent, numele programului. Prin utilizarea

parametrilor funcţiei main, limbajul C oferă posibilitatea transmiterii unor valori programului, în momentul lansării în execuţie. Aceste valori, constituite în şiruri de caractere, urmează numelui programului şi sunt separate de acesta şi între ele prin spaţiu. Primul parametru al funcţiei main este de tip întreg şi conţine numărul şirurilor de caractere care alcătuiesc linia de lansare în execuţie a programului (inclusiv numele acestuia). Al doilea parametru este un tablou de pointeri la şiruri de caractere şi conţine adresele din memorie la care sunt memorate şirurile de caractere introduse de la tastatură în momentul lansării în execuţie (inclusiv numele programului).

În corpul funcţiei main mai întâi este verificată introducerea corectă a liniei de comandă pentru lansarea în execuţie a programului. Aceasta trebuie să conţină două şiruri de caractere (numele programului şi numele fişierului). Dacă valoarea variabilei *argv* este diferită de 2, este afişat un mesaj de eroare care sugerează forma corectă a comenzii de lansare în execuţie.

Nu este posibilă lansarea în execuţie a programului din mediul integrat **TURBO C ++/ BORLAND C** cu ajutorul comenzii **RUN**. Pentru a ajunge în **MS-DOS**, se selectează din meniul principal al mediului de programare funcţia „**FILE**”, iar din submeniul acesteia funcţia „**DOS shell**”. La afişarea prompterului **MS-DOS** se poate tasta comanda de punere în execuţie.

Programul utilizează funcţiile *scrie_c* şi *citeşte_c*. Fiecare dintre ele conţine în prima parte setul de instrucţiuni destinat deschiderii fişierului cu care lucrează.

În funcţia *scrie_c*, caracterul tastat este preluat cu ajutorul funcţiei *getche* şi memorat în variabila *car* de tipul caracter. În expresia logică a funcţiei *while* este testat conţinutul acestei variabile pentru a detecta momentul în care s-a apăsat tasta <**ENTER>**, adică momentul în care utilizatorul a terminat de introdus toate caracterele. Cât timp nu se întâmplă acest lucru, caracterul este scris în fişier prin apelul funcţiei *putc*. Forma generală a funcţiei putc este:

putc(caracter, pointer_fişier);

unde

caracter este o constantă sau o variabilă de tipul caracter,

pointer_fişier este pointerul la fişierul în care aceasta urmează să fie scrisă.

În funcţia *citeşte_c* este utilizată funcţia *getc* pentru a citi conţinutul fişierului creat prin apelul funcţiei *scrie_c*. Funcţia getc, care este perechea funcţiei putc pentru accesarea fişierelor de tip caracter, primeşte ca

parametru pointerul la fişierul din care se face citirea şi returnează codul caracterului citit. Forma generală de apel a acestei funcţii este:

rezultat = getc(pointer_fişier);

Variabila *rezultat* trebuie să fie de tipul caracter sau întreg fără semn. Această necesitate este impusă de mecanismul de detectare a sfârşitului de fişier. Mesajul de sfârşit de fişier îl constituie valoarea întreagă –1 (constanta EOF- end of file- definită în fişierul antet stdio.h).

11.2. Scrierea şi citirea fişierelor de tipul şir de caractere

Operaţiile de citire şi scriere a fişierelor de tipul şir de caractere sunt similare cu cele pentru citirea şi scrierea fişierelor de tipul caracter, de această dată utilizându-se funcţiile **fputs** şi **fgets**.

Pentru exemplificarea modului în care sunt efectuate operaţiile de scriere şi citire a fişierelor de acest tip, vom analiza programul din exemplul 11.2.

Exemplul 11.2.

Programul creează fişierul cu numele *text.txt* în care va scrie o succesiune de caractere citite de la tastatură, apoi conţinutul fişierului este citit şi afişat pe ecran.

```
#include <stdio.h>
#include <conio.h>
void main (void)
{
        FILE *pointer_fişier;
        char buf(s1);
        if ((pointer_fişier = fopen(“text.txt”, “w”)) = = NULL)
        {
                printf(“\nEroare! Nu poate fi deschis fişierul!”);
                exit();
        }
        clrscr();
        printf(“\nIntroduceţi un text \n”);
        while(strlen(gets(buf))>0)
        {
                fputs(buf, pointer_fişier);
                fputs(”\n”, pointer_fişier);
```

```
        }
        fclose(pointer_fişier);
/* citirea fişierului*/
        if(pointer_fişier=fopen("text.txt", "r")) = = NULL)
        {
                printf("\nEroare! Nu poate fi deschis fişierul!");
                exit();
        }
        clrscr();
        printf("\nConţinutul fişierului\n");
        while(fgets(buf,80, pointer_fişier) != NULL)
                printf("%s", buf);
        fclose(pointer_fişier);
        gtech();
}
```

Scrierea fişierului se realizează în cadrul primei bucle while. Fiecare şir tastat de către utilizator este preluat prin apelul funcţiei **gets** şi memorat în variabila tablou de caractere *buf*. Aceasta, având dimensiunea 81, permite memorarea unor şiruri cu lungimea maximă de 80 caractere, adică a unor şiruri care ocupă cel mult o linie de ecran. Dacă lungimea şirului introdus este 0 (s-a acţionat tasta <Enter> chiar la începutul unei linii), programul interpretează această acţiune ca semnal al încetării operaţiei de scriere, părăsind bucla şi închizând fişierul.

Scrierea în fişier a fiecărui şir de caractere se realizează cu ajutorul funcţiei **fputs**, care are următoarea formă generală de apel:

fputs(şir_caractere, pointer_fişier);

unde

şir_caractere este o constantă sau o variabilă de tipul şir de caractere,

pointer_fişier reprezintă un pointer la fişierul în care se va înscrie informaţia conţinută în primul parametru.

În programul prezentat, bucla while care realizează scrierea fişierului conţine două apeluri ale funcţiei fputs. Primul realizează scrierea efectivă a şirului conţinut în *buf*. Deoarece funcţia fputs nu adaugă automat caracterul "new line"(\n) la sfârşitul şirului scris, acesta este adăugat explicit de către al doilea apel de funcţie fputs. Această acţiune este necesară pentru a se facilita citirea fişierului în cea de a doua buclă while, prin apelul funcţiei **fgets**. Forma generală de apel a acestei funcţii este:

fgets(nume_şir, l_max, pointer_fişier);

unde

nume_şir specifică locul unde va fi memorat şirul de caractere citit (numele unui tablou de caractere sau numele unei variabile pointer la un şir de caractere),

l_max specifică lungimea maximă a şirului de caractere citit, având rolul de a împiedica citirea unor şiruri mai lungi decât capacitatea de memorare alocată prin intermediul primului parametru,

pointer_fişier este pointerul la fişierul din care se face citirea.

Funcţia fgets returnează o valoare întreagă care reprezintă lungimea şirului de caractere citit. Valoarea returnată este utilizată în expresia logică a buclei while pentru a detecta sfârşitul fişierului. În momentul în care această valoare este 0 (NULL) se părăseşte bucla şi se închide fişierul.

11.3. Scrierea şi citirea fişierelor de tipul formatat

Modul formatat permite scrierea şi citirea simultană atât a caracterelor şi a şirurilor de caractere, cât şi a valorilor numerice. Datele, constituite în colecţii mixte de caractere şi valori numerice, sunt transferate între program şi fişier în conformitate cu descriptorii de format, identici cu cei folosiţi în cadrul funcţiilor printf şi scanf.

Exemplul 11.3.

Programul reprezintă o variantă îmbunătăţită a programului din exemplul , deoarece informaţiile referitoare la un candidat la concursul de admitere sunt scrise în fişierul *baza.can* în vederea prelucrărilor ulterioare.

```
#include <stdio.h>
#include<conio.h>
#include <string.h>
struct candidat
{
        char nume[25], prenumeă25î;
        int nr_leg ;
        float nota1, float nota2, media ;
};
typedef struct candidat candidat;
candidat x;
int citeşte (void) ;
void main (void)
{
        FILE *pointer_fişier;
        if ((pointer_fişier = fopen(“baza.can”, “w”)) = = NULL)
```

```
        {
                printf("\nEroare! Nu poate fi deschis fişierul!");
                exit();
        }
        /*înscrierea datelor în fişier*/
        while (citeşte())
        {
                fprintf(pointer_fişier, "%s %s %d %f %f %f", x.nume,
x.prenume,
                        x.nr.leg, x.nota1, x.nota2, x.media);
                if (x.media >= 5)
                        printf("%s %s admis", x.nume, x.prenume);
                else
                        printf("%s %s respins", x.nume, x.prenume);
                getch();
        }
        fclose(pointer_fişier);
        clrscr()
        /* citirea datelor din fişier şi afişarea lor pe ecran*/

        if ((pointer_fişier = fopen("baza.can", "r")) = = NULL)
        {
                printf("\nEroare! Nu poate fi deschis fişierul!");
                exit();
        }
        while ((fscanf(pointer_fişier, "%s %s %d %f %f %f", x.nume,
x.prenume,
                x.nr.leg, x.nota1, x.nota2, x.media)) != EOF)
        {
                printf("\nNumele candidatului : %s %s\nNumărul legitimaţiei
                de
                        concurs: %d", x.nume, x.prenume, x.nr_leg);
                printf("\nNotele obţinute : %f %f\nMedia: %f", x.nota1,
        x.nota2, x.media);
                if (x.media >= 5)
                        printf(" admis ");
                else
                        printf(" respins ");
                getch();
        }
        fclose(pointer_fişier);
```

```
}

int citeşte();
{
        float a, b;
        clrscr();
        printf("Numele candidatului :");
        gets(x.nume);
        if (strlen(x.nume) = = 0)
                exit(0);
        printf("Prenumele candidatului :");
        gets(x.prenume);
        printf("Numărul legitimaţiei de concurs:");
        scanf("%d", &x.nr_leg);
        printf("Notele obţinute: ");
        scanf("%f%f", &a, &b);
        x.nota1= a;
        x.nota2 = b;
        x.media=( a + b)/2 ;
        return (1);

}
```

În zona de înscriere a datelor în fişier, în cadrul buclei while, informaţiile sunt citite de la tastatură cu ajutorul funcţiei **citeşte**, sunt prelucrate pentru a afişa rezultatul obţinut de candidat şi apoi sunt scrise în fişier prin apelul funcţiei **fprintf**. Această funcţie, similară cu **printf**, are următoarea formă generală de apel:

fprintf(pointer_fişier, şirul_descriptorilor_de_format, lista_variabile);

Singura deosebire faţă de funcţia **printf** o constituie prezenţa ca prim argument a pointerului *pointer_fişier* la fişierul în care se va face scrierea.

La părăsirea buclei while (tastarea <Enter> la solicitarea numelui candidatului), fişierul este închis, astfel încât, dacă programul s-ar încheia în acest moment, informaţiile referitoare la candidaţi ar fi disponibile. Pentru a ilustra acest fapt, am introdus a doua zonă de program în care se face citirea fişierului şi afişarea datelor pe ecran.

Citirea înregistrărilor din fişier se realizează în cadrul unei bucle while cu ajutorul funcţiei **fscanf (**similară funcţiei **scanf)**, constituind perechea funcţiei **fprintf** şi având următoarea formă generală de apel:

fscanf(pointer_fişier, şirul_descriptorilor_de_format,
lista_adreselor_variabilelor);

Şi de această dată, deosebirea faţă de funcţia **scanf** o constituie prezenţa ca prim argument a pointerului *pointer_fişier* la fişierul din care se va face citirea.

Atât funcţia **fprintf**, cât şi funcţia **fscanf** returnează un întreg a cărui valoare reprezintă numărul de octeţi scrişi, respectiv citiţi din fiţier. În cazul funcţiei **scanf**, după citirea ultimei înregistrări, la o nouă tentativă de citire aceasta va returna valoarea –1, adică indicatorul de sfârşit de fişier EOF. De aceea, pentru a detecta sfârşitul fişierului, apelul funcţiei **fscanf** este utilizat ca expresie logică a buclei while. Atât timp cât valoarea returnată de fscanf este diferită de valoarea –1 (EOF), programul afişează pe ecran informaţiile citite din fişier.

Funcţia fscanf nu poate fi utilizată pentru citirea şirurilor de caractere ce conţin spaţii albe, acestea fiind interpretate ca terminatoare de şir, citirea făcându-se incorect. Pentru a evita această acţiune, în structura *candidat* a fost introdus un nou tablou de caractere în care se va memora prenumele candidatului, separat de numele său.

Modul formatat de scriere şi citire a datelor prezintă două dezavantaje importante. Un prim dezavantaj îl constituie utilizarea ineficientă a spaţiului de memorare datorită faptului că fiecare cifră a datelor numerice este memorată pe un octet. Al doilea dezavantaj rezultă din faptul că pentru scrierea şi citirea datelor de tipul structură şi tablou este necesară accesarea fiecărui element în parte al acestora, nefiind posibilă tratarea lor ca o entitate.

11.4. Scrierea şi citirea fişierelor de tipul înregistrare

Inconvenientelor prezentate de modul înregistrare sunt eliminate cu ajutorul modului înregistrare. În acest mod de exploatare a fişierelor datele sunt memorate în formă binară şi este permisă citirea şi scrierea oricărei cantităţi de informaţie în cadrul unei singure instrucţiuni.

Exemplul 11.4.

Programul citeşte de la tastatură informaţiile referitoare la candidaţii la un concurs de admitere, folosind funcţia **citeşte** şi apoi le depune într-un fişier.

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

```
#include <string.h>
struct candidat
{
        char nume[25], prenumeă25î;
        int nr_leg ;
        float nota1, float nota2, media ;
};
typedef struct candidat candidat;
candidat x;
int citeşte (void) ;
void main (void)
{
        FILE *pointer_fişier;
        if ((pointer_fişier = fopen("baza.can", "w")) = = NULL)
        {
                printf("\nEroare! Nu poate fi deschis fişierul!");
                exit();
        }
        /*înscrierea datelor în fişier*/
        while (citeşte())
        {
                        fwrite(&x, sizeof(x), 1, pointer_fişier);
        }
        fclose(pointer_fişier);
}

int citeşte();
{
        float a, b;
        clrscr();
        printf("Numele candidatului :");
        gets(x.nume);
        if (strlen(x.nume) = = 0)
                exit(0);
        printf("Prenumele candidatului :");
        gets(x.prenume);
        printf("Numărul legitimaţiei de concurs:");
        scanf("%d", &x.nr_leg);
        printf("Notele obţinute: ");
        scanf("%f%f", &a, &b);
        x.nota1= a;
```

```
        x.nota2 = b;
        x.media=( a + b)/2 ;
        return (1);

}
```

De această dată fişierul *baza.can* este un fişier binar, iar scrierea informaţiilor în el se realizează în modul înregistrare prin intermediul funcţiei **fwrite** a cărei formă generală de apel este:

```
fwrite(&înregistrare, lung_înreg, nr_întreg, pointer_fişier);
```

unde

&înregistrare este adresa de memorie la care se află înregistrările care vor fi scrise în fişier (poate fi adresa unei structuri sau a unui tablou),

lung_întreg reprezintă lungimea înregistrării exprimată în octeţi,

nr_înreg reprezintă numărul de înregistrări de acelaşi tip care vor fi scrise în fişier; dacă *înregistrare* este un tablou de structuri, iar o structură reprezintă o înregistrare, atunci *nr_înreg* va avea valoarea egală cu dimensiunea tabloului.

pointer_fişier este pointerul la fişierul în care se face scrierea.

La terminarea buclei while fişierul este închis, astfel încât informaţiile pot fi regăsite şi după terminarea programului. Pentru a ilustra acest fapt, este prezentat programul din exemplul 11.5.

Exemplul 11.5.

Programul citeşte conţinutul fişierului *baza.can* scris în exemplul 11.4. afişând pe ecran rezultatele obţinute de candidaţii la concursul de admitere.

```
#include <stdio.h>
#include<conio.h>
#include <string.h>
struct candidat
{
        char nume[25], prenumeă25î;
        int nr_leg ;
        float nota1, float nota2, media ;
};
typedef struct candidat candidat;
candidat x;
int citeşte (void) ;
```

```
void main (void)
{
        FILE *pointer_fişier;
        if ((pointer_fişier = fopen(“baza.can”, “r”)) = = NULL)
        {
                printf(“\nEroare! Nu poate fi deschis fişierul!”);
                exit();
        }
        while ((fread(&x, sizeof(x), 1, pointer_fişier) != 1)
        {
                printf(“\nNumele candidatului : %s %s\nNumărul legitimaţiei
                de
                        concurs: %d”, x.nume, x.prenume, x.nr_leg);
                printf(“\nNotele obţinute : %f %f\nMedia: %f”, x.nota1,
        x.nota2, x.media);
                if (x.media >= 5)
                        printf(“ admis ”);
                else
                        printf(“ respins ”);
                getch();
        }
        fclose(pointer_fişier);
}
```

Citirea fişierului se realizează cu ajutorul funcţiei **fread** a cărei formă generală de apel este:

fread(&înregistrare, lung_înreg, nr_întreg, pointer_fişier);

unde parametrii au aceleaşi semnificaţii ca şi parametrii funcţiei **fwrite**, singura deosebire constând în aceea că *&înregistrare* constituie de această dată adresa de memorie în care vor fi depuse cele *re_înreg* citite din fişier.

Funcţia fread returnează un întreg care reprezintă numărul de înregistrări citite efectiv. În condiţiile unor citiri corecte, valoarea returnată este egală cu valoarea celui de al treilea parametru de apel. Dacă în tentativa sa de citire funcţia fread întâlneşte indicatorul EOF, atunci valoarea returnată este mai mică decât valoarea specificată prin cel de al treilea parametru de apel.

În cadrul buclei while se citeşte o înregistrare (*nr_înreg* =1) din fişierul referit de *pointer_fişier* atât timp cât valoarea returnată de fread este 1. Înregistrarea citită este memorată în variabila *x* de tipul **structură candidat** care este apoi prelucrată pentru a decide dacă respectivul candidat a obţinut sau nu o medie mai mare decât 5.

11.5. Accesarea directă a unor înregistrări

În numeroase aplicaţii, cum ar fi cele de creare, actualizare şi prelucrare a bazelor de date, modul de acces secvenţial este ineficient datorită faptului că, prin timpul mare de acces la o înregistrare, reduce viteza de execuţie a programului. Pentru a elimina acest dezavantaj, sistemul standard de intrare /ieşire al limbajului C oferă utilizatorului, ca o alternativă la modul de acces secvenţial, posibilitatea accesării aleatoare a înregistrărilor unui fişier. Aceasta înseamnă posibilitatea de a accesa direct o anume înregistrare, indiferent de poziţia pe care o ocupă aceasta în fişier.

Exemplul 11.6.

```
#include <stdio.h>
#include<conio.h>
void main (void)
{
        long int offset;
        int i;
        int mat [4][4] =
{{11,12,13,14},{21,22,23,24},{31,32,33,34},{41,42,43,44}};
        FILE *p_fis;
        if ((p_fis = fopen(“stoc.mat”, “wb”)) = = NULL)
        {
                printf(“\nEroare! Nu poate fi deschis fişierul!”);
                exit();
        }
/* scrie liniile matricei în fişier*/
        for(i=0; i<4 ; i++)
                fwrite(&mat [i] [0], sizeof(int), 4, p_fis);
        floes(p_fis);
/*deschiderea fşierului în vederea citirilor*/
        if ((p_fis = fopen(“stoc.mat”, “rb”)) = = NULL)
        {
                printf(“\nEroare! Nu poate fi deschis fişierul!”);
                exit();
        }
 /*citeşte a patra înregistrare şi o depune în a patra linie a matricei*/
        offset = 3*4*sizeof(int);
        fseek(p_fis, offset, 0);
```

```
        fread(&mat [3] [0] , sizeof(int), 4, p_fis);
/*citeşte prima înregistrare şi o depune în prima linie a matricei*/
        offset = 0;
        fseek(p_fis, offset, 0);
        fread(&mat [0] [0] , sizeof(int), 4, p_fis);
/*citeşte a treia înregistrare şi o depune în a doua linie a matricei*/
        offset = 2*4*sizeof(int);
        fseek(p_fis, offset, 0);
        fread(&mat [1] [0] , sizeof(int), 4, p_fis);
/*citeşte a doua înregistrare şi o depune în a treia linie a matricei*/
        offset = 1*4*sizeof(int);
        fseek(p_fis, offset, 0);
        fread(&mat [2] [0] , sizeof(int), 4, p_fis);
        fclose(p_fis);
        clrscr();
        printf("Matricea citită din fiţier:");
        for (i=0; i<4; i++)
                printf(„\n%d%d%d%d", mat[i][0], mat[i][1], mat[i][2],
mat[i][3]);
        getch()
}
```

În prima parte a programului este creat fişierul binar cu numele *stoc.mat* în care se vor scrie elementele matricei de tip întreg *mat,* având patru linii şi patru coloane. Scrierea se face în mod secvenţial, cu ajutorul funcţiei **fwrite** în cadrul buclei **for**. La fiecare apel al funcţiei fwrite se scriu în fişier cele patru elemente de tip întreg care alcătuiesc linia *i* a matricei *mat*, constituite într-o înregistrare.

S-a ales acest mod de a scrie matricea (linie cu linie), deşi nu este eficient, pentru a uşura înţelegerea modului de acces direct la înregistrările unui fişier.

Cea mai eficientă variantă de scriere a unei matrice într-un fişier este aceea de a o scrie printr-un singur apel al funcţiei fwrite. Astfel, în locul secvenţei de instrucţiuni

```
        for(i=0; i<4 ; i++)
                fwrite(&mat [i] [0], sizeof(int), 4, p_fis);
```

se poate folosi instrucţiunea

```
                fwrite(&mat [0] [0], sizeof(int), 16, p_fis);
```

Se observă că valorile cu care s-a făcut iniţializarea elementelor matricei au fost alese astfel încât prima cifră constituie indicele liniei, iar a doua cifră indicele coloanei. În a doua parte a programului este folosit

accesul direct la fişierul *stoc.mat* pentru a citi liniile matricei şi a le depune în tabloul *mat* schimbând ordinea liniilor a doua cu a treia.

Adresa unei înregistrări din fişier (adresa unui octet) se obţine adăugând la adresa de început deplasamentul acesteia, numit **offset**. Deplasamentul primei înregistrări este 0, iar deplasamentul unei înregistrări *n* se obţine cu expresia *(n-1)*lungime_înregistrare*.

Dacă în modul de acces secvenţial poziţionarea pointerului în fişier se face în mod automat, în modul de acces direct programatorul trebuie mai întâi să poziţioneze pointerul în fişier la înregistrarea dorită şi apoi să efectueze operaţia de scriere sau citire. Acest lucru se realizează cu ajutorul funcţiei **fseek**, care are următoarea formă generală de apel:

fseek(p_fis, deplasament, mod);

unde

p_fis este pointerul la structura de tip FILE asociată fişierului la deschiderea acestuia,

deplasament este o variabilă de tip long int în care se memorează deplasamentul,

mod este un întreg care arată faţă de cine se măsoară deplasamentul:

mod = 0 deplasament relativ la începutul fişierului,

mod = 1 deplasament relativ la poziţia curentă în fişier,

mod = 2 deplasament relativ la sfârşitul fişierului.

Pe baza acestui mecanism, mai întâi este citită ultima înregistrare (a patra) din fişierul *stoc.mat* şi apoi memorată pe ultima linie a matricei. Pentru aceasta, se atribuie deplasamentului valoarea 24 (cu instrucţiunea offset = 3*4*sizeof(int);), se poziţionează pointerul în fişier (apelul funcţiei **fseek** cu modul 0) şi se efectuează citirea (apelul funcţiei **fread**).

În continuare, prin atribuire de valori corespunzătoare deplasamentului şi apelul funcţiilor fseek şi fread, sunt citite toate înregistrările. Acestea sunt memorate în liniile corespunzătoare ale matricei *mat*, aşa cum rezultă din comentariile inserate în program. În urma execuţiei programului, pe ecran va fi afişat noul conţinut al matricei *mat* în care se constată că liniile 2 şi 3 au fost schimbate între ele.

12
Structuri de date şi algoritmi. Introducere

12.1. Definiţia algoritmilor

Studiul algoritmilor acoperă patru domenii distincte:

1. **maşini pentru execuţia algoritmilor**: acest domeniu include o gamă largă de echipamente, de la cel mai mic calculator de buzunar până la calculatorul digital de largă utilizare.Scopul acestui domeniu este determinarea formei optime de construcţie a maşinii, astfel încât algoritmii să poată fi efectiv implementaţi.

2. **limbaje pentru descrierea algoritmilor**.

Se pot utiliza:

- limbaje de nivel scăzut (limbaje de asamblare, apropiate de nivelul maşinii);
- limbaje de nivel înalt, proiectate pentru realizarea unor probleme complexe;
- pseudocod;
- organigrame;
- limbaj natural.

În cadrul acestui domeniu, se pot distinge **două faze**:
- descrierea limbajului;
- traducerea.

Prima fază face apel la **metodele de specificare a sintaxei şi semanticii limbajului.**

A doua fază impune găsirea unui limbaj alcătuit dintr-un **set de comenzi mai simple**.

Initial algoritmul este descris în limbaj natural. Pentru eliminarea oricărei ambiguităţi în descrierea algoritmului acesta va fi exprimat sub forma unei organigrame (sau în pseudocod). Aceasta descriere este o descriere formală a algoritmului care va fi utilizată în implemetarea acestuia.

Plecînd de la descrierea formală se vor scrie secventele de program (modulele de program) utilizînd un limbaj de nivel înalt (C, Pascal, Fortran, Matlab). Fisierele cu aceste module poarta numele de **programe sursă**.

În continuare programele sursă vor fi convertite în **programe obiect** – ca urmare a etapei de traducere (compilare, asamblare). Programele obiect vor fi legate între ele împerună cu fisierele de date (care contin variabilele utilizate de algoritm) printr-un proces numit **editare de legături** (**link-are, link editare**) . Rezultatul obtinut in urma procesului de ediatre de legaturi este un fisier numit **program (cod) executabil** care este direct interpretabil de către maşina de calcul.

Programul executabil trebuie sa indeplinesacă două cerinţe de performantă :

- să se execute într-un timp minim posibil
- să utilizeze o zonă de memorie minim posibilă

Rezultă că dezvoltarea teoretică a algoritmilor in sine şi utilizarea unor structuri de date eficiente (din punct de vedere al accesului la fiecare componentă a structurii de date şi a memoriei utile ocupate) sînt deosebit de importante pentru obţinerea unor programe executabile eficiente.

3. **fundamentarea algoritmilor**. În cadrul acestui domeniu trebuie definite următoarele aspecte:

- dacă o problemă particulară poate fi rezolvată cu ajutorul unei maşini de calcul;
- numărul minim de operaţii necesare rezolvării unei probleme date.

4. **analiza algoritmilor**. Dacă un algoritm poate fi specificat, are sens să se discute despre "comportarea" sa. Această analiză a fost realizată pentru prima dată de Charles Babbage, "părintele computerelor", în 1830. "Comportarea" unui algoritm sau "profilul performanţelor sale" poate fi apreciată prin măsurarea timpului de calcul şi a spectrului de memorie consumate în timpul procesării algoritmului.

Se poate observa că în definirea ştiinţei calculatoarelor ("**computer science**"), **algoritmul** este o noţiune fundamentală, care trebuie definită cu precizie.

Definiţie: un algoritm este un **set finit de instrucţiuni a căror parcurgere rezolvă o problemă dată**.

Orice algoritm trebuie să satisfacă următoarele criterii, în ceea ce priveşte:

a) **intrarea**: trebuie să existe n≥0 cantităţi furnizate din exterior;

b) **ieşirea**: trebuie să producă n≥1 cantităţi;

c) **claritatea**: fiecare instrucţiune trebuie să fie clară şi neambiguă;

d) **finitudinea**: dacă se parcurg toate instrucţiunile unui algoritm, atunci, în toate cazurile, algoritmul trebuie să se termine după un număr finit de paşi;

e) **efectiv**: fiecare instrucţiune trebuie să fie suficient de simplă, încât algoritmul să poată fi, în principiu, executat de o persoană, folosind numai creion şi hârtie.

În "computer science", noţiunile de **algoritm** şi **program** sunt distincte.

Astfel, **programul nu trebuie să satisfacă condiţia de finitudine**. Un exemplu important de program care nu trebuie să satisfacă această condiţie, este **sistemul de operare**, care nu se termină niciodată (cu execepţia opririi calculatorului), ci rămâne în buclă în aşteptarea introducerii de noi job-uri.

Programul poate fi definit ca forma codificată a unui algoritm, folosind un limbaj de programare "înţeles" de către procesor.

Deoarece algoritmii prelucrează date, se poate spune că **ştiinţa calculatoarelor este ştiinţa studierii datelor şi analizează următoarele aspecte**:

a) maşinile care păstrează datele;

b) limbajele pentru descrierea operaţiilor asupra datelor;

c) metodele care descriu ce tipuri de date prelucrate pot fi obţinute din datele primare;

d) structuri pentru reprezentarea datelor.

Observaţii: există o strânsă interdependenţă între structurarea datelor şi sinteza algoritmilor. De fapt, o structură de date şi un algoritm trebuie gândite ca o unitate, nici una dintre ele neavând sens fără cealaltă. Rezultă pentru fiecare problemă, necesitatea alegerii adecvate a structurilor de date şi în conformitate cu acestea, a algoritmilor optimali pentru operarea asupra acestor structuri.

12.2. Definiţia structurilor de date

Trebuie determinate definiţiile pentru următoarele noţiuni:

1. tip de date;
2. obiect de date;
3. structură de date;
4. reprezentarea datelor.

Observaţie: aceste denumiri nu sunt standardizate în limbajul ştiinţei compute-
relor. Se pot întâlni situaţii în care sunt folosite interschimbat.

1. Tipul de date - este un termen care se referă la mulţimea valorilor pe care le pot lua variabilele într-un limbaj de programare.

De exemplu, în **C** există:

- **tipuri simple: char, int, float şi double**, reprezentând caractere (1 byte), întregi cu semn, numere în virgulă mobilă în simplă şi respectiv dublă densitate;
- **tipuri structurate**: tip enumerare, tip tablou, tip fişier, tip referinţă.

2. Obiectul de date - este un termen care se referă la o mulţime de elemente, D, de exemplu:

- pentru obiectul de date "numere întregi", D este:

$$D = \{0, \pm1, \pm2, ...\}$$

- pentru obiectul de date "şir de caractere de lungime mai mică decât 5", D este:

$$D = \{'A', 'B', ... , 'Z', 'AA', ... , 'ZZ', ...\}$$

Mulţimea D poate fi finită sau infinită. În cazul în care D este infinită, sunt necesare metode speciale de reprezentare a elementelor mulţimii în calculator.

3. Structura de date - spre deosebire de obiectul de date, structura de date se referă atât la mulţimile de obiecte, cât şi la **setul de operaţii** care pot fi aplicate obiectului de date.

Exemplu: obiectul de date "numere întregi", împreună cu descrierea modului de comportare la operaţiile: +, - , * , / , constituie definirea **structurii de date "numere întregi"**.

4. Implementarea structurii de date - d (sau "reprezentarea datelor") este **maparea** din structura **d** într-o altă structură de date, de exemplu **e**.

Operaţia de mapare specifică modul în care se reprezintă fiecare obiect din **d** în funcţie de obiectele din **e**.

Se impune ca fiecare funcţie din **d** să fie definită utilizând funcţii din **e**.

Exemplu: - **structura de numere** întregi este reprezentată prin **şiruri de biţi**;

- un şir este reprezentat printr-o **mulţime de cuvinte consecutive în memorie**.

12.3. Principii generale de scriere a programelor

Operaţia de creare a unui program se desfăşoară în cinci etape:

a) specificarea problemei;
b) proiectarea;
c) analiza;
d) codarea;
e) verificarea.

a) **Specificarea problemei** presupune o descriere riguroasă a intrărilor şi ieşirilor, astfel încât să se precizeze toate situaţiile posibile.

b) **Proiectarea** unui algoritm poate fi realizată independent de limbajul de programare care urmează a fi utilizat. Fiind cunoscute obiectele de date şi presupunând disponibile (sub formă de proceduri) operaţiile ce se execută asupra lor, se va defini un algoritm care să rezolve problema dată.

c) **Analiza** presupune determinarea şi a altor algoritmi posibili pentru aceeaşi problemă, alegându-se în final algoritmul optim.

Criteriile generale de analiză a programelor sunt următoarele:

- funcţionarea conform specificaţiilor;
- existenţa documentaţiei de folosire şi funcţionare;
- asigurarea posibilităţii rezolvării unor subprobleme logice de către procedurile definite.

Acestea sunt criterii **subiective**. Există însă şi criterii **obiective** de apreciere, corelate în mod direct cu performanţele programului:

- timpul de calcul;
- dimensiunea memoriei necesare;
- numărul de execuţii pentru fiecare instrucţiune.

Evaluarea performanţelor unui program se realizează în două faze: **evaluarea apriorică** şi **evaluarea ulterioară**.

Pentru o analiză calitativă apriorică a performanţelor unui program, este suficientă evaluarea numărului de execuţii pentru fiecare instrucţiune.

Se presupune că undeva, în programul analizat, există instrucţiunea:

x:=x+1;

Pentru această instrucţiune, trebuie determinate:

- timpul de execuţie;
- numărul de execuţii.

Produsul acestor numere reprezintă **timpul total de execuţie** pentru această instrucţiune.

Determinarea acestui timp nu se poate realiza fără următoarele informaţii:

- maşina pe care se rulează programul;
- limbajul de asamblare;
- durata execuţiei fiecărei instrucţiuni-maşină;
- compilatorul folosit.

Se consideră următoarele trei exemple prezentate în figura 12.1.

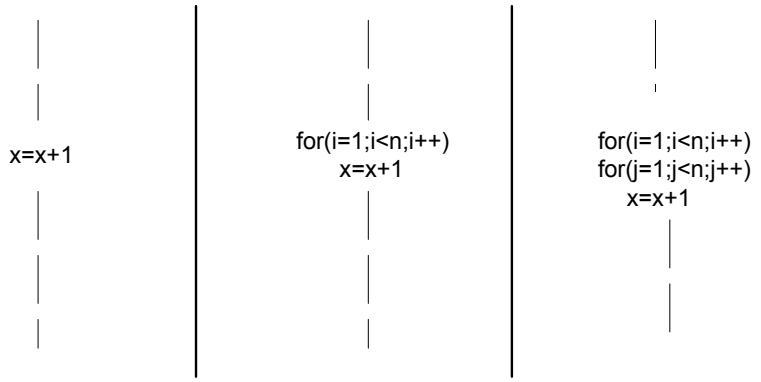


Fig.12.1. Determinarea numărului de execuţii ale unei instrucţiuni

În programul a), instrucţiunea x=x+1 nu este conţinută în nici o buclă, implicit sau explicit, deci este executată o singură dată. În programul b), este executată de n ori, iar în c), de n^2 ori. Numerele 1, n, n^2 sunt definite ca diferite **ordine de mărime** pentru N. În analiza execuţiei programelor trebuie determinate aceste ordine de mărime. În calculul efectuat pentru determinarea lor apar deci sume de tipul:

$$\sum_{i=1}^{n} i, \quad \sum_{i=1}^{n} i^2,$$

ale căror valori corespunzătoare sunt:

$n(n+1)/2$ şi respectiv $n(n+1)(2n+1)/6$.

În general:

$$\sum_{i=1}^{n} i^k = n^{k+1}/k+1 + \text{termeni de grad mai mic,} \qquad \text{unde } k\geq 0.$$

În situaţia în care timpul ce execuţie estimat la evaluarea apriorică nu este corespunzător se va modifica algoritmul şi structurile de date astfel încît timpul de execuţie să scadă.

Evaluarea ulterioară furnizează informaţii exacte asupra timpului de calcul în urma unor măsuratori efectuate in timpul execuţiei programului.

d) **Rafinarea şi codarea** presupun alegerea reprezentărilor obiectelor de date şi scrierea algoritmilor pentru operaţiile care trebuie efectuate asupra acestor obiecte.

Observaţie: algoritmii depind esenţial de reprezentarea datelor, deci trebuie definiţi **după** alegerea structurii de date. Este recomandabil să se aleagă mai multe moduri de reprezentare a datelor şi să se compare algoritmii corespunzători rezultaţi.

e) **Verificarea** unui program se realizează în trei etape: **demonstraţia**, **testarea** şi **depanarea**. Dacă s-a **demonstrat corectitudinea** programului, rezultă că pentru orice set de intrări, programul şi specificaţia coincid. **Testarea** presupune crearea de eşantioane de date la intrare pentru care se rulează programul. Dacă într-o situaţie dată rezultatele nu corespund specificaţiei, programul trebuie **depanat**. Setul de date de intrare ales pentru realizarea testării trebuie definit astfel încât să se forţeze

execuţia tuturor instrucţiunilor şi să creeze situaţiile specifice în care fiecare condiţie ia valoarea **TRUE** sau **FALSE,** cel puţin o dată.

Notaţii asimptotice

Pentru analiza complexităţii programelor, se utilizează următoarele notaţii:

Definiţie: **f(n) = *O*(g(n))**, dacă există două constante c, $n_o > 0$, astfel încât:

$|f(n)| \leq c|g(n)|, \ \forall n \geq n_o$.

f(n) reprezintă în mod curent, timpul de calcul pentru un algoritm. Se spune că timpul de calcul al unui algoritm este ***O*(g(n))**, dacă execuţia sa nu durează mai mult decât un timp constant **g(n)**, **n** fiind un parametru care caracterizează intrările şi/sau ieşirile.

Deci, notaţia ***O*** este utilizată pentru a descrie performanţele unui algoritm.

13
Noţiuni generale despre algoritmi recursivi

Definiţie: un algoritm recursiv este un algoritm a cărui descriere conţine direct sau indirect un apel la algoritmul respectiv.

Instrumentul necesar şi suficient pentru a implementa programe recusive, este **procedura** (sau **funcţia**), deoarece acordă un **nume** unei **acţiuni**, care poate fi activată prin invocarea numelui respectiv.Dacă o procedură **P** conţine o referinţă explicită la ea însăşi, atunci se spune că este **direct recursivă**.; dacă **P** conţine o referinţă la altă procedură **Q**, care la rândul său, conţine o referinţă (directă sau indirectă) la **P**, atunci se spune că **P** este **indirect recursivă.**

Exemple:

- **definirea numerelor naturale**:

a) 1 este un număr natural;
b) succesorul unui număr natural este un număr natural;

- **definirea unor funcţii:**
  - **funcţia factorial:**

a) 0!=1;
b) dacă n>0, atunci n!=n*(n-1)!

Utilitatea algoritmilor recursivi constă în posibilitatea de reprezentare finită a unor mulţimi de obiecte infinite (structuri de date); sunt utilizaţi în special, în cazurile în care problema sau structura de date este definită recursiv.

Definiţia unui program recursiv este următoarea:

P = f[Si, P],

unde **Si** este o secvenţă de instrucţiuni, iar **f** reprezintă funcţia care asociază secvenţa **Si** şi apelul **P**, în definirea procedurii **P**.

Definiţia anterioară evidenţiază problema fundamentală care apare în cazul algoritmilor recursivi: **problema încheierii lor (ieşirea din recursivitate)**. Este evident că definiţia poate conduce la o secvenţă infinită de prelucrări. Problema terminării algoritmului se rezolvă **dacă apelul recursiv este realizat din cadrul unei instrucţiuni condiţionate** (la un moment dat condiţia devine falsă), ca de exemplu:

P = f[Si, if (B) P]

sau

P = f[Si, while (B) {Sj, P}].

O altă condiţie de "ieşire din recursivitate" poate fi de exemplu:

$\mathbf{f(x) \leq 0}$,

unde **f** este o funcţie convenabil definită, iar **x** sunt variabilele procedurii. Astfel, se asigură terminarea programului dacă şi numai dacă la fiecare apel al procedurii valoarea funcţiei este decrementată.

Un alt mod particular de a garanta terminarea unui program recursiv, este de a asocia procedurii un parametru **n** şi de a apela recursiv procedura, cu parametrul **n-1**:

P(n) = f[Si, if (n>0) P(n-1)]

Recusivitate sau iteraţie?

Recursivitatea poate fi întotdeauna transformată în iteraţie (implementarea de algoritmi recursivi pe maşini cu organizare nerecursivă este o dovadă în acest sens). În majoritatea implementărilor, forma nerecursivă a unui program este mai eficientă decât cea recursivă, în ceea ce priveşte timpul de execuţie şi memoria ocupată:

$\mathbf{t_{recursiv} > t_{nerecursiv}}$

În alegerea modalităţii - recursivă sau iterativă - de implementare a unui program, utilizatorul trebuie să stabilească prioritatea elementelor care trebuie rezolvate în timpul conceperii programului, analizând complexitatea acestuia, naturaleţea exprimării, simplitatea proiectării, testării şi întreţinerii programului, eficienţa în execuţie.

Se acceptă soluţii recursive:

- în cazurile în care nu există probleme critice de timp;
- pentru compilatoarele optimizate, la care $\mathbf{t}_{recursiv} < \mathbf{t}_{nerecursiv}$;
- în situaţiile în care înlocuirea recurenţei cu iteraţia determină depunerea unui efort deosebit sau utilizarea unor tehnici speciale de programare, algoritmul pierzându-şi astfel claritatea exprimării (forma nerecursivă este în general mai greu de citit), testarea şi întreţinerea lui devenind deosebit de dificile.

Există un set de reguli care asigură posibilitatea translatării unui algoritm din forma recursivă, în forma iterativă.

De asemenea, **există o clasă de algoritmi recursivi cu un echivalent direct într-o formă iterativă.** Un astfel de algoritm recursiv este definit sub forma:

```
void P;
{
if (B) {
        Po; P;
}
};
```

în care **Po** reprezintă o secvenţă de instrucţiuni în care nu se apelează procedura **P**.

Această definiţie este echivalentă cu:

```
while (B)
        Po;
```

Din cele prezentate anterior, rezultă că pentru algoritmii recursivi la care apelul recursiv este realizat o singură dată, la începutul (sau la sfârşitul) procedurii, se obţine în mod direct un **algoritm echivalent iterativ**.

Exemple:

a) Funcţia factorial

***Varianta recursivă*:**

```
int fact(int n)
{
        if (n= =0) return1;
                 else return (n*fact(n-1));
}
```

***Varianta iterativă*:**

```
int fact(int n)
{
        int i,j;
        if (n= =0) return 1;
                else {
                        j=1;
                        for (i=1; i=n; i++);
                        j*=i;
                        return j;
                }
}
```

b) Generarea şirului Fibonacci

Relaţia de recurenţă este:
- pentru $n \geq 2$, $x_n = x_{n-1} + x_{n-2}$, cu $x_0 = 0$, $x_1 = 1$.

Varianta recursivă

```
int fibn(int n)
{
        if (n= =0) return 0;
        if (n= =1) return 1;
                else
                return (fibn(n-1) + fibn(n-2));
}
```

Varianta recursivă prezintă dezavantajul calculului funcţiei de m≥2 ori (pentru un argument <n). De exemplu, pentru determinare **fibn**(5), se obţine secvenţa de apeluri prezentată în figura 13.1.

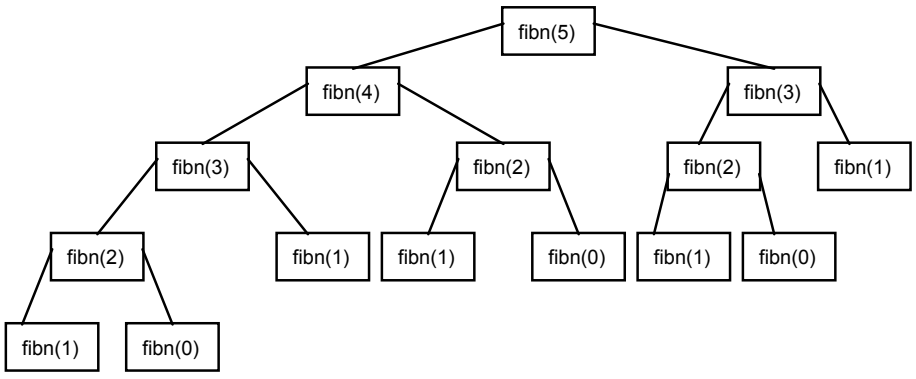


Fig. 13.1. Determinarea funcţiei fibn(5)

Varianta iterativă

```
int fibn(int n)
        {
                int s, i, x, y;
                if (n= =0) return 0;
                if (n= =1) return 1;
                        else {
                                x=1; y=0;
                                for (i=2; i=n; i++){
                                        s=x; x=x+y; y=s;
                                }
                                return x;
                        }
        }
```

14
Liste de tip stivă şi coadă

14.1. Definiţii liste liniare

O formă uzuală de organizare a datelor în programare, este **lista ordonată** sau **liniară**, care este definită sub forma: $\mathbf{A=\{a_1, a_2, ... ,a_n\}}$.

Definiţie: o listă liniară este o mulţime $\mathbf{A=\{a_1, a_2, ... ,a_n\}}$ cu $\mathbf{n\geq 0}$ elemente (**noduri**), între care **există o relaţie de ordine determinată de poziţia relativă (unidimensională) a nodurilor în mulţimea A**.

Astfel, orice element $\mathbf{a_k}$ are un predecesor $\mathbf{a_{k-1}}$ şi un succesor $\mathbf{a_{k+1}}$, cu excepţia nodurilor $\mathbf{a_1}$ (nu are predecesor) şi $\mathbf{a_n}$ (nu are succesor).

Reprezentarea listelor
Listele se pot reprezenta:

a) Prin tablou (*mapare secvenţială*), în care :

- se asociază elementului $\mathbf{a_i}$ indicele **i**;
- elementele listei se stochează în locaţii succesive de memorie.

Avantajul acestui mod de reprezentare constă în timpul de accesare constant pentru oricare element al listei.

Dezavantajele reprezentării listelor prin tablouri sunt următoarele:
- lungimea fixă a listei;
- introducerea/ştergerea dificilă a unui element în/din listă (deoarece presupune deplasarea spre stânga/dreapta a unui număr variabil de elemente din listă, deci un timp de execuţie variabil şi mare).

b) Prin liste înlănţuite (*mapare nesecvenţială*)

Avantajele reprezentării prin liste înlănţuite sunt următoarele:
- lungimea variabilă a listei;
- memorarea elementelor listei în locaţii neadiacente de memorie;
- inserarea/ştergerea simplă a unui element din listă (deoarece nu impune deplasarea elementelor listei).

Dezavantajul utilizării acestei metode de reprezentare constă în timpul de acces variabil pentru elementele listei.

Pentru a accesa un element din listă, în ordinea corectă, în fiecare element se memorează adresa sau locaţia următorului nod.

Aplicaţiile uzuale cu liste sunt următoarele:

a) **creare (l)** - crează lista **l**;
b) **suprimă (l)** - şterge lista **l**;
c) **citire (l)** - citeşte lista **l** de la stânga la dreapta sau de la dreapta la stânga;
d) **card (l)** - returnează numărul de elemente ale listei **l**;
e) **read (l, i)** - întoarce elementul de rang **i** din lista **l**;
f) **write (l, i, u)** - memorează un nou element **u**, în poziţia **i**, în lista **l**;
g) **insert (l, i, u)** - inserează un nou nod **u**, în poziţia **i**, în lista **l** şi renumerotează nodurile;
h) **şterge (l, i)** - elimină nodul de rang **i** din lista **l** şi renumerotează nodurile rămase;
i) **sort (l)** - sortează nodurile listei **l** în ordinea crescătoare/descrescătoare a valorii unuia dintre câmpurile nodurilor;
j) **search (u, l)** - caută în lista **l** nodul având valoarea **u**.

Observaţie: în anumite cazuri particulare de liste (reprezentate secvenţial sau nesecvenţial), este posibil să nu se utilizeze toate operaţiile prezentate anterior.

14.2. Stive

Definiţie: stiva reprezintă o listă ordonată, în care toate inserările/eliminările de elemente se realizează printr-un singur capăt, denumit **vârful stivei (TOP)**.

Din această definiţie rezultă că modul de acces la elementele stivei este de tip **LIFO** (**"Last In First Out"**), după cum rezultă şi din figura 14.1.

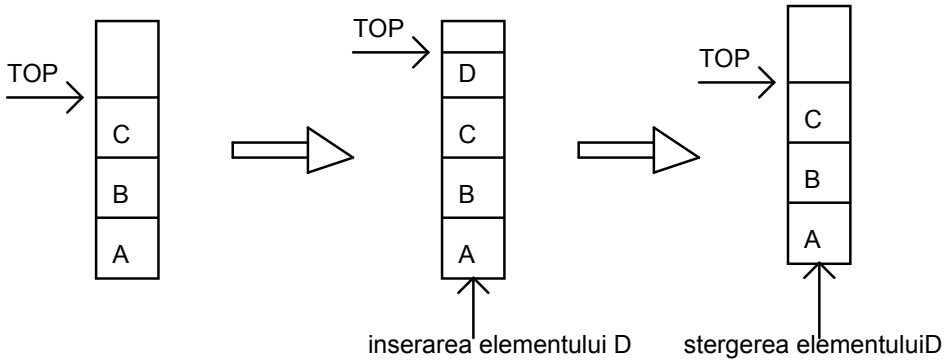


Fig. 14.1. Structura de date de tip stivă

Structura de date de tip stivă este frecvent utilizată în programare, cel mai elocvent exemplu fiind apelurile de subrutine (figura 14.2).

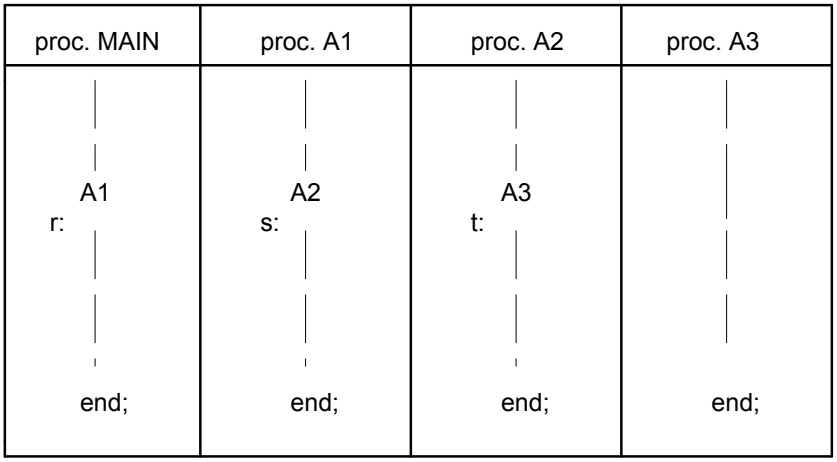


Fig. 14.2. Utilizarea stivelor pentru implementarea apelurilor de proceduri

Procedura **MAIN** apelează procedura **A1**. După execuţia procedurii **A1**, se reia procedura **MAIN** de la adresa **r**. Adresa **r** este memorată de

procedura **A1** (**adresă de revenire**). Procedura **A1** apelează procedura **A2**, care la rândul său apelează procedura **A3**. În fiecare caz, procedura apelantă transmite procedurii apelate adresa de revenire. Dacă se examinează memoria în timp ce se execută procedura **A3**, se observă existenţa unei stive cu structura din figura 14.3.

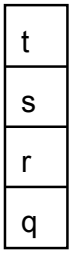


Fig. 14.3. Structura stivei obţinute ca urmare a apelurilor procedurilor MAIN, A1, A2, A3

Primul element al stivei, **q**, reprezintă adresa de revenire a procedurii **MAIN**. Lista prezentată în figura 14.3 operează ca o stivă, deoarece revenirile sunt realizate în ordinea inversă a apelurilor: **t** este "scos" din stivă înainte de **s**, **s** este "scos" înainte de **r** şi **r** înainte de **q**.

Operaţiile uzuale care se execută asupra stivelor sunt următoarele:

- **CREATE ( )** - creare de stivă (iniţial vidă);
- **ADD (item, stack)** - adăugare de element (**item**) în stivă şi returnarea noii stive (**stack**);
- **DELETE (stack)** - eliminarea elementului din topul stivei şi returnarea noii stive;
- **TOP (stack)** - determinarea elementului din vârful stivei;
- **ISEMTS (stack)** - funcţie care returnează valoarea 1 dacă stiva este vidă şi 0 în caz contrar.

În continuare sunt prezentate implementările celor cinci funcţii, în cazul reprezen- tării secvenţiale a stivei prin tabloul **stack[n]**, de maxim **n** elemente.

Se defineşte variabila **top** (de tip **int**) ca pointer spre vârful stivei şi se presupune cunoscut tipul **items**, ca tip de bază al tabloului. Ca urmare,

rezultă următoarele **implementări ale operaţiilor definite pentru structurile de date de tip stivă**:

```
void CREATE ( )
        {
                items stack[n];
                int top;
                top=0;
        }

- int ISEMTS (stack)
        {
                 if (top= =0) return 1;
                                  else return 0;
        }

- items TOP (stack)
        {
                  if (top= =0)  error;
                           else
                                  return stack[top];
        }
```

Inserările şi **eliminările** din stivă sunt realizate prin intermediul următoarelor proceduri (pentru care se consideră variabilele: **top, stack** şi **n**, variabile globale).

Inserarea unui element în stivă: se inserează elementul **item** în stiva **stack**; **top** reprezintă vârful curent al stivei, iar **n**, dimensiunea maximă a stivei.

```
void ADD(items item)
        {
                if (top= =n) stackfull
                          else {
                                  top++;
                                  stack[top]=item;
                          }
        }
```

Eliminarea unui element din stivă:

```
items DELETE()
        {
        items item;
                if (top= =0) stackempty
                        else{
                                item=stack[top];
                                top--;
                                return(item);
                                 }
        }
```

Se precizează că implementarea procedurilor **stackempty** şi **stackfull** care semnalizează starea de stivă vidă, respectiv stivă plină, depinde de aplicaţie.

14.3. Cozi

Definiţie: coada reprezintă o listă ordonată, în care toate inserările sunt realizate printr-un capăt denumit **coadă (rear)**, iar eliminările sunt realizate prin celălalt capăt denumit **cap (front)**.

Se defineşte astfel o structură de tip **FIFO** (**"First In First Out"**). Cea mai răspândită aplicaţie a cozilor constă în **planificarea proceselor**: procesele care trebuie executate de un program aşteaptă în cozi, din care sunt extrase pentru a fi executate, pe rând, în ordinea în care au fost înscrise în cozi (se ignoră în acest caz eventualele priorităţi).

Operaţiile uzuale care se execută asupra cozilor sunt următoarele:

- **CREATEQ ( )** - creare de coadă (iniţial vidă)
- **ADDQ (i, Q)** - adăugarea unui element **i** în capătul **rear** al unei cozi **Q** şi returnarea noii cozi
- **DELETEQ (Q)** - eliminarea elementului din capătul **front** al cozii **Q** şi returnarea noii cozi

- **FRONT (Q)** - returnarea elementului din capătul **front** al cozii **Q**;
- **ISEMTQ (Q)** - funcţie care returnează valoarea 1 dacă **Q** este vidă şi 0 în caz contrar.

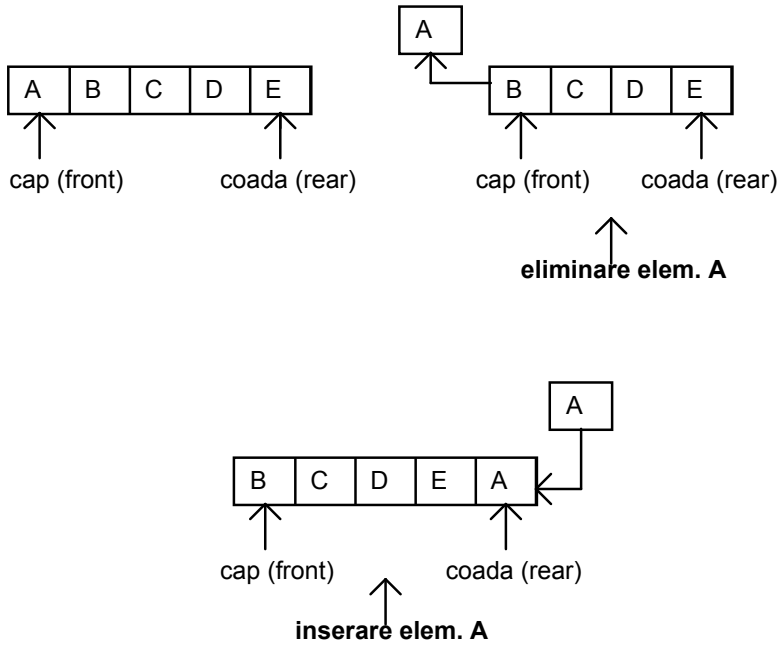


Fig. 14.4. Structura de date de tip coadă

Se consideră reprezentarea secvenţială a cozii sub forma unui tablou **q**, de maximum **n** elemente, având ca tip de bază, tipul **items**. Se definesc variabilele întregi **front, rear**, ca pointeri spre capul, respectiv sfârşitul cozii.

Ca şi în cazul stivei, variabilele **q, front, rear** şi **n** se consideră globale.

Implementările operaţiilor menţionate anterior sunt următoarele:

```
- void CREATEQ( )
        {
                items q[n];
                int front, rear;
```

```
            front=0;
            rear=0;
        }

- int ISEMTQ (q)
        {
            if (front= =rear) return 1;
                        else return 0;
        }

- items FRONT (q)
        {
            if (ISEMTQ (q))  error;
                            else q[front+1];}
```

S-au adoptat următoarele convenţii:

- variabila **front** este întotdeauna mai mică cu o unitate decât actualul **front** al cozii;
- **rear** este pointer către ultimul element al cozii;
- valorile iniţiale pentru aceste variabile sunt: **front=rear=0.**

În figura 14.5 este reprezentat un exemplu de inserare şi eliminare a unor elemente (J), dintr-o coadă.

Exemplu:

		Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	Q[6]	Q[7]	Obs.
front	rear								
0	0			coadă	vidă				iniţial
0	1	J1							se inserează J1
0	2	J1	J2						se inserează J2
0	3	J1	J2	J3					se inserează J3
1	3		J2	J3					se elimină J1
1	4		J2	J3	J4				se inserează J4
2	4			J3	J4				se elimină J2

Fig. 14.5. Inserarea/eliminarea elementelor J, dintr-o coadă Q

Procedurile de inserare şi eliminare a unui element dintr-o coadă, sunt următoarele:

Inserarea unui element item în coada Q:

```
void ADDQ (items item)
        {
                if (rear= =n) queuefull;
                        else {
                                rear++;
                                q[rear]:=item;
                        }
        }
```

Eliminarea unui element item din coada Q:

```
Items DELETEQ ()
        {
        items item;
                if (front= =rear)  queueempty;
                        else {
                                front++;
                                item=q[front];
                                return(item);
                                }
        }
```

Procedurile **queuefull** şi **queueempty** care semnalizează starea de coadă plină sau vidă, sunt implementate în funcţie de aplicaţia corespunzătoare.

În implementarea anterioară a structurii de coadă, se remarcă faptul că testul de coadă vidă este echivalent cu testul **front=rear**. De asemenea, se constată fenomenul de deplasare spre dreapta a cozii. Aceasta prezintă dezavantajul semnalizării eronate a stării de coadă plină (**rear=n**), deşi în realitate nu toate elementele tabloului sunt ocupate (**1<front<n**).

În această situaţie, pentru a permite adăugarea de noi elemente în coadă, este necesară translaţia apriorică spre stânga a tuturor elementelor cozii, ceea ce necesită un timp de procesare mare (dacă elementele care trebuie translatate sunt numeroase).

O reprezentare mai eficientă a cozilor, din acest punct de vedere, este ca un tablou cu **n elemente, circular**. În momentul în care **rear=n-1**,

următorul element este introdus în **coada circulară**, în poziţia **q[0]**, dacă locaţia respectivă este liberă.

În acest caz, operaţiile de incrementare pentru variabilele **front** şi **rear** sunt realizate **modulo n**:

front=(front++) % n; rear=(rear++) % n.

În aceste condiţii, procedurile **ADDQ** şi **DELETEQ** se modifică în modul următor:

```
void ADDQ (items item)
        {
                rear=(rear++) % n;
                if (front= =rear) queuefull;
                        else q[rear]=item;
        }
```

şi

```
items DELETEQ ()
        {
        if (front= =rear) queueempty;
                else {
                        front=(front++) % n;
                        item=q[front];
                        return(item);
                        }
        }
```

În figura 14.6 este reprezentată o **coadă circulară** în două dintre configuraţiile posibile:

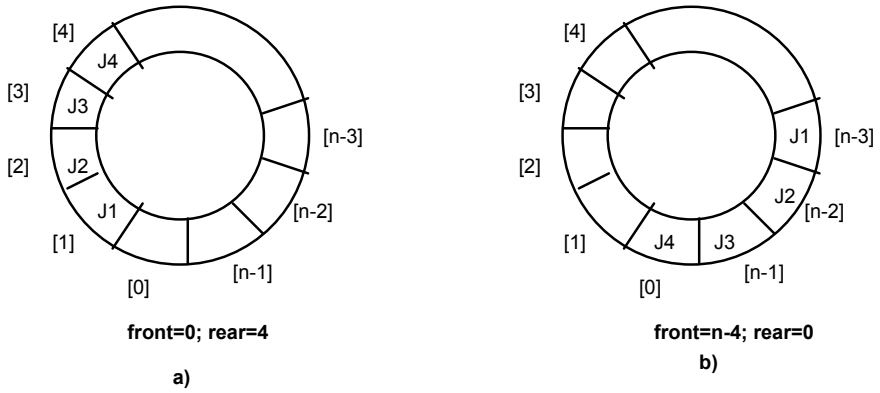


Fig.14.7. Reprezentarea cozilor circulare

Se observă că în cazul cozilor circulare, condiţia de coadă vidă şi cea de coadă pilnă coincid (**front=rear**). Pentru tratarea independentă a celor două situaţii, se utilizează următoarele metode:

a) definirea unei variabile suplimentare **tag** de tip **int**, care are valoarea **1** în situaţia **coadă vidă** şi **0** în caz contrar (**coadă plină**). Dezavantajul acestei soluţii constă în mărirea timpului de execuţie a procedurilor **ADDQ** şi **DELETEQ**, ca urmare a modificării condiţiei de testare, sub forma:

if ((front= =rear) && (!tag)) **queuefull**,

respectiv:

if ((front= =rear) && (tag)) **queueempty**

b) utilizarea a numai **n-1** poziţii din lungimea totală a tabloului. Astfel, condiţia de **coadă vidă** rămâne:

rear=front,

iar cea de **coadă plină** devine:

(rear-front) % n = n-1

c) introducerea unui contor **cnt** , care va fi incrementat la fiecare nou element introdus în coadă şi decrementat la fiecare extragere din coadă.

Condiţia de coadă plină devine **cnt = n** , iar condiţia de coadă vidă **cnt = 0** .

14.4. Utilizarea stivelor pentru evaluarea expresiilor

14.4.1. Formele infixate şi postfixate de scriere a expresiilor aritmetice

Forma uzuală de scriere a expresiilor aritmetice este cea în care operatorul se află poziţionat **între** operanzii asupra cărora acţionează, formă denumită **infixată**. Pentru evaluarea unei expresii aritmetice scrise sub forma infixată, trebuie luată în considerare prioritatea operatorilor şi existenţa parantezelor, ceea ce complică algoritmul de evaluare.

Forma **postfixată** corespunde situaţiei în care operatorul este poziţionat **după** operanzii asupra cărora acţionează. În această reprezentare, operatorii nu au aprioric priorităţi diferite, prioritatea lor fiind determinată de **poziţia** lor în cadrul expresiei analizate (expresia aritmetică parcurgându-se de la stânga la dreapta, rezultă descreşterea priorităţii operatorilor de la stânga la dreapta). Acest avantaj determină simplificarea considerabilă a algoritmului de evaluare a expresiilor aritmetice scrise în forma postfixată. Ca urmare, pentru evaluarea unei expresii aritmetice, orice compilator realizează mai întâi **conversia expresiei din forma infixată în forma postfixată**.

În tabelul 14.1 sunt prezentate câteva exemple de expresii aritmetice scrise în **forma infixată** şi corespondenţa lor în **forma postfixată.**

Tabelul 14.1.

Forma infixată	Forma postfixată
A+B	AB+
(A+B)*C	AB+C*
A+B*(C-D)	ABCD-*+
A/B**C+D*E-A*C	ABC**/DE*+AC*-

unde pentru **operaţia de ridicare la putere** a fost utilizată notaţia **.

14.4.2. Evaluarea expresiilor aritmetice scrise în forma postfixată

În continuare este descris (într-un limbaj pseudo-C) algoritmul de evaluare a expresiilor aritmetice în forma postfixată. Se presupun definite tipurile: **expression, token,** iar declaraţia de variabile este următoarea:

```
token stack[n];
```

Tipul **expression** specifică un şir de caractere; expresiile scrise în forma infixată sau postfixată sunt variabile de tipul **expression**.

Tipul **token** specifică un şir de caractere şi corespunde variabilelor de tip operator sau operand.

Algoritmul de evaluare pentru o expresie scrisă în forma postfixată, este următorul:

```
void eval(expression e)

/*e=expresie aritmetica în forma postfixata si încheiata cu
caracterul '#';
   functia nexttoken extrage din sirul e, operanzii si
operatorii*/
{
        token x;
        top=0; /*initializeaza stiva*/
        x=nexttoken(e);
        while (x!='#'){
                if (x este operand) ADD(x);
                                else /*operator*/ {
                                        extrage din stiva numarul
                                        corespunzator de operanzi
                                        aplica operatorul operanzilor
                                        depune rezultatul în stiva
                                }
        x=nexttoken(e);
        }
}
```

14.4.3. Conversia expresiilor aritmetice din forma infixată în forma postfixată

Algoritmul de conversie a expresiilor aritmetice din forma infixată în forma postfixată, se desfăşoară în trei etape (forma infixată conţine operanzii în aceeaşi ordine ca şi cea postfixată):

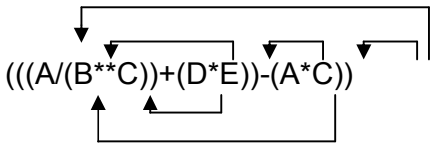
- completarea întregii expresii cu paranteze;
- deplasarea tuturor operatorilor în poziţiile parantezelor corespunzătoare;
- eliminarea tuturor parantezelor.

Exemplu:

Se consideră expresia infixată:

A/BC+D*E-A*C**

care se completează cu paranteze:



Săgeţile reprezintă deplasările operatorilor în poziţiile parantezelor corespunzătoare.

Se obţine expresia în forma **postfixată**:

ABC/DE*+AC*-**

Rafinarea algoritmului precedent presupune parcurgerea următoarelor etape:

a) explorarea expresiei infixate de la stânga la dreapta;
b) scrierea operanzilor la ieşire;
c) depunerea operatorilor într-o stivă, în care se memorează până în momentul în care trebuie scrişi la ieşire, pentru obţinerea expresiei aritmetice în forma postfixată.

Este necesară definirea explicită a momentului în care operatorii se extrag din stivă. În acest scop, sunt analizate exemplele prezentate în continuare.

Exemple:

1. expresia infixată: A+B*C

Tabelul 14.2

Următorul token	Stiva	Ieşire
1. nimic	vidă	nimic
2. A	vidă	A
3. +	+	A
4. B	+	AB
5. *	+*	AB
6. C	+*	ABC
7. nimic	+*	ABC
8. nimic	vidă	ABC*+

Se constată că după încheierea parcurgerii expresiei aritmetice (liniile 7 şi 8), stiva trebuie descărcată şi operatorii depuşi la ieşire.

De asemenea, la linia 5, trebuie determinat dacă operatorul **'*'** se depune în stivă, sau dacă operatorul **'+'** trebuie extras din stivă. Deoarece operatorul **'*'** are prioritate mai mare (conform ierarhiei priorităţilor prezentate în tabelul 14.3, pentru operatorii aritmetici binari şi delimitatori), trebuie depus în stivă (conform regulii care va fi prezentată în continuare).

Tabelul 14.3.

Simbol	Prioritatea în stivă (ps)	Prioritatea la intrare (pi)
)	-	-
**	3	4
*,/	2	2
+ binar	1	1
- binar	1	1
(	0	4

Se consideră valorile din tabelul 14.3. memorate în vectorii **ps** şi **pi**. Utilizând aceste notaţii, a fost determinată următoarea **regulă** care stabileşte momentele în care operatorii se pot extrage din stivă: dacă **ps al operatorului din vârful stivei ≥ pi al operatorului de la intrare**, atunci se extrage operatorul din vârful stivei; în caz contrar, operatorul de la intrare se depune în stivă.

2. expresia infixată: A*(B+C)*D

Următorul token	Stiva	Ieşire
1. nimic	vidă	nimic
2. A	vidă	A
3. *	*	A
4. (	*(	A
5. B	*(	AB
6. +	*(+	AB
6. C	*(+	ABC
7.)	*(+	ABC
8. *	*	ABC+*
9. D	*	ABC+*D
10. nimic	vidă	ABC+*D*

Cu aceste precizări, rezultă următorul **algoritm de conversie a expresiilor aritmetice din forma infixată în forma postfixată:**

```
void postfix (expression e)
/*scrie forma postfixata a expresiei infixate e, care se încheie
cu caracterul '#', având ps('#')=-1*/
{
        token x, y;
        stack[1]='#'; top=1; /*initializeaza stiva*/
        x=nexttoken(e);
        while (x!='#') {
                if (x este operand} scrie (x);
                  else if (x= =')') { /*descarca stiva pâna la '(' */
                                while (stack[top]!='('){
                                        delete(y); //scrie(y);
                                }
                                        delete(y); /*sterge '(' */
                                };
                        else {
                                while (ps(stack[top])>=pi(x)){
                                        delete(y); //scrie(y);
                                }
                                add(x);
                                }
```

```
            x=nexttoken(e);
}
while (top>1){
            delete(y); //scrie(y);
}
printf("#");
}
```

15
Liste înlănţuite

15.1. Liste simplu înlănţuite

Se presupune că s-a definit un tip de date, numit **DATA**, care este specific informaţiei dintr-un element (**nod**) al listei. Acest tip poate fi un tip simplu de date, un tablou, o structură, etc. Pentru a realiza o **încapsulare** a datelor, este indicat ca această informaţie să fie inclusă într-o structură.

Se defineşte structura **C** adecvată unui element al listei, prin:

```
typedef struct elem {
        DATA data;
        struct elem *next;
} ELEMENT, *LINK;
```

Tipul ELEMENT corespunde unui element al listei, iar tipul LINK unui pointer la un element al listei.

Lista este specificată printr-o variabilă de tip LINK care indică primul element al listei. O listă vidă se va specifica printr-un pointer NULL. Câmpul **next** al ultimului element al listei va conţine NULL, indicând că nu mai urmează nici un element.

Se observă că o listă simplu înlănţuită este o structură ordonată, care poate fi parcursă direct de la primul element către ultimul (într-un singur sens). Parcurgerea în ambele sensuri nu se poate face direct, dar se poate realiza prin folosirea unei structuri adiţionale de tip **stivă**. Totuşi, această variantă nu este prea folosită. Când sunt necesare parcurgeri în ambele sensuri, se folosesc listele dublu înlănţuite.

O listă liniară se poate reprezenta grafic ca în figura 15.1.

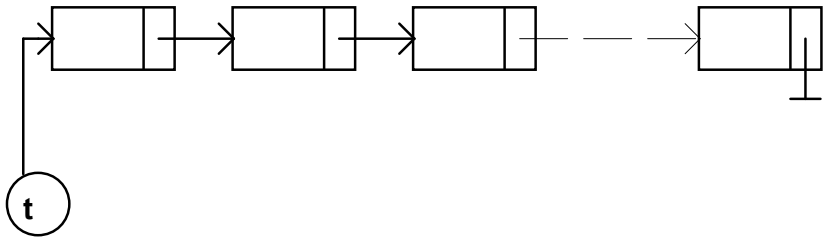


Fig. 15.1. Liste liniare simplu înlănţuite

Simbolul de "punere la masă" din ultimul element indică pointerul NULL.

O altă variantă de implementare este cea cu **header**, în care există un prim element, **header**-ul, care conţine un câmp LINK ce indică primul element util al listei. Această variantă are avantajul că lista nu este niciodată vidă, existând cel puţin header-ul. Dacă nu există nici un element util în listă, câmpul LINK al header-ului este NULL. În câmpul de date al header-ului (care poate avea altă structură decât tipul DATA) se pot memora unele informaţii, de exemplu numărul elementelor utile din listă. În continuare, se va alege reprezentarea **fără header**.

Pentru semnalarea situaţiilor de eroare, se va folosi următoarea funcţie, care tipăreşte şirul primit şi apoi forţează oprirea programului.

```
void err_exit(const char*s)
        {
                printf("\n%s\n",s);
                exit(1);
        }
```

unde s-a utilizat funcţia:

```
void exit(int status);
```

care indică terminarea normală (neforţată) a programului.

O primă operaţie care trebuie implementată este cea de creare a unui element , printr-o funcţie **new_el()**. Spaţiul pentru elementele unei liste se crează la execuţie prin funcţia standard **malloc()**. Funcţia:

```
void *malloc(size_t size);
```

întoarce un pointer la o zonă compactă din memoria dinamică, de dimensiune **size** octeţi, sau NULL dacă nu există spaţiu disponibil. Spaţiul alocat nu este iniţializat. De aceea, este bine ca funcţia de creare, **new_el** să iniţializeze câmpurile **data** şi **next** ale elementului creat. Această funcţie va întoarce un pointer la elementul creat.

```
typedef int DATA;
struct el {DATA data; struct el*next;};
typedef struct el ELEMENT, *LINK;
LINK new_el(DATA x, LINK p)
        {
                LINK t=(LINK) malloc(sizeof(ELEMENT));
                if (t= =NULL)
                        err_exit("new_el:Eroare malloc");
                t->data=x;
                t->next=p;
                return t;
        }
```

Se observă testarea corectitudinii alocării cu **malloc()**, ca şi conversia (**cast**) la tipul LINK şi folosirea lui **sizeof()** pentru a obţine dimensiunea unui element.

În continuare este dată o funcţie de tipărire a unei liste înlănţuite, în varianta recursivă.

```
void print_data(DATA x)
{
        printf("%6d",x);
}

void print_list(LINK t)
{
        if (t= =NULL)
                printf("NULL\n");
        else {
                print_data(t->data);
                printf("->");
                print_list(t->next);
              }
}
```

Funcţia recursivă de parcurgere a unei liste simplu înlănţuite este următoarea:

```
void parcurge (LINK t)
        {
                if (t= =NULL) {
                        /*tratează cazul de bază (NULL)*/
                        ;
                else {
                        prel(t->data);
                        parcurge(t->next);
                       }
        }
```

S-a considerat o funcţie **prel()** care prelucrează un element al listei, primind câmpul **data** al acestuia. Se poate folosi şi o variantă în care **prel()** primeşte un pointer la elementul respectiv.

Pentru o listă liniară nu se poate implementa un algoritm de căutare binară (nu se poate determina poziţia "elementului din mijloc"), ci numai un algoritm de căutare secvenţială (liniară). Funcţia care caută un element dat într-o listă dată, întorcând un pointer la primul element găsit, sau NULL dacă lista nu conţine elementul dat, este următoarea:

```
int cmp(DATA a, DATA b);
LINK linsrc(LINK t, DATA x)
{
        if (t = = NULL||cmp(t->data,x)= = 0)
                return t;
        else return linsrc(t->next, x);
}
```

Funcţia **cmp()** compară două elemente întorcând un întreg. O asemenea funcţie este necesară, deoarece în C structurile nu se pot compara.

Ordinea în care apar condiţiile asociate prin || este esenţială; dacă **t este NULL**, comparaţia **t->data** cu **x** nu trebuie făcută (deoarece **t->data** nu există).

Avantajul esenţial al structurilor înlănţuite faţă de cele secvenţiale (tablouri) apare la operaţiile de inserare şi ştergere de elemente. La un tablou (şir), dacă se şterge un element, toate elementele care urmează trebuie deplasate la stânga cu o poziţie, pentru a păstra forma continuă a tabloului. Similar, dacă se inserează un element, toate elementele care

urmează trebuie în prealabil deplasate la dreapta cu o poziţie. Aceste operaţii depind de locul unde se face inserarea sau ştergerea. Dacă se şterge primul element, trebuie deplasate cele **n-1** elemente care urmează. Dacă se şterge ultimul, nu se face nici o deplasare. În medie, numărul de deplasări este **n/2**, deci ineficient când **n** este mare.

La structurile înlănţuite, operaţiile care se fac nu depind de locul unde se inserează sau se şterge, ordinul algoritmilor fiind constant (nu depinde de **n**).

Principalele operaţii care se efectuează asupra listelor liniare sunt următoarele:

a) Inserarea unui element dat, după un element dat al unei liste

```
void ins_after(LINK p, LINK q)
{
        if (q= =NULL||p= =NULL)
                return;
        q->next=p->next;
        p->next=q;
}
```

Aici **p** este elementul din listă după care se face inserarea, iar **q** este elementul care se inserează. Dacă **q** este NULL, nu există nod de inserat, iar dacă **p** este NULL, nu există nod după care să se insreze.

Dacă **p** este ultimul element din listă (adică **p->next** este NULL), se leagă **q** la acest ultim element şi **q** devine ultimul element din listă. În cazul în care **q** nu reprezintă doar un element, ci o listă întreagă (cazul inserţiei sau concatenării unei liste în/la altă listă), atribuirea **q->next = = p->next** va realiza acest lucru.

Inserţia se face deci, prin două atribuiri de pointeri.

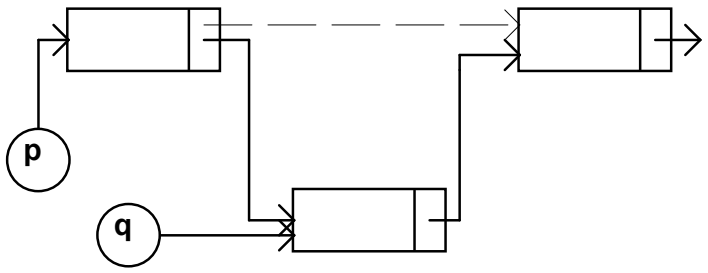


Fig.15.2. Inserarea dupa un element intr-o lista simplu înlănţuită

Trebuie remarcat faptul esenţial că, în instrucţiunile de atribuire, expresiile din membrul stâng sunt toate obţinute cu operatorul **->**, deci se modifică conţinutul unor zone de memorie accesate prin pointeri, adică se modifică efectiv lista respectivă.

b) Inserarea unui element înaintea unui element dat al unei liste

Se observă că nu se poate face direct o inserare **înaintea** unui element dat dintr-o listă (datorită sensului unic de parcurgere a listei, nu se cunoaşte elementul anterior). Este totuşi posibil un artificiu şi anume, să se insereze **după** elementul specificat şi apoi să se schimbe câmpurile de date ale celor două elemente. Metoda funcţionează doar atunci când cele două câmpuri sunt relativ simple (este problematică situaţia în care, de exemplu, câmpul de date conţine, la rândul lui, pointeri către alte liste sau structuri de date). În exemplul din figura 15.3, este prezentat cazul în care se doreşte inserarea nodului cu valoarea 100, înaintea nodului cu valoarea 18.

```
void insert(LINK p, LINK q)
{
        DATA temp;
        if (q= =NULL || p= =NULL)
                return;
        temp=p->data;
        p->data=q->data;
        q->data=temp;
        ins_after(p,q);
}
```

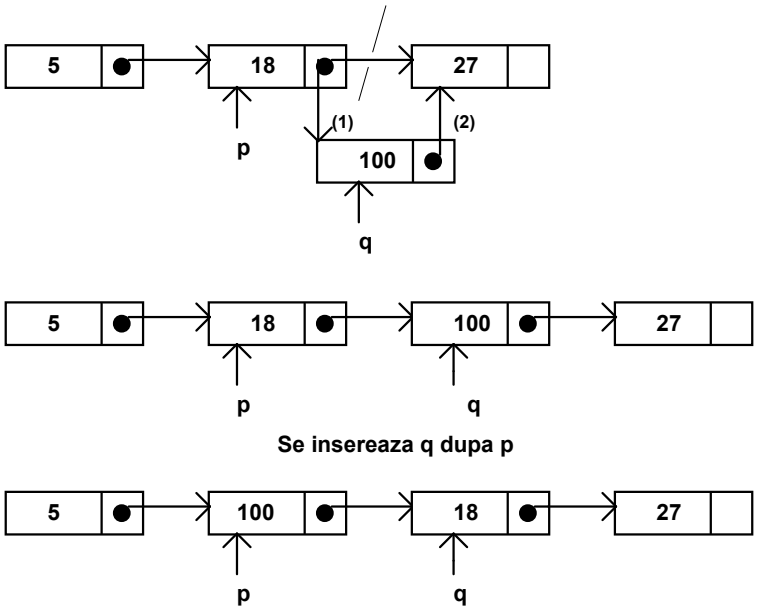


Fig. 15.3. Inserarea unui nod *y* înainte de nodul referit de pointerul *x*

c) Ştergerea elementului de după un element dat

```
void del_after(LINK p)
{
        LINK q;
        if(p= =NULL|| p->next= =NULL)
                return;
        q=p->next;
        p->next=q->next;
        free(q);
}
```

unde funcţia:

void **free**(void *p);

eliberează zona indicată de **p**, care a fost obţinută prin **malloc()**.

Dacă **p** este NULL sau este ultimul element din listă, nu se efectuează nici o operaţie. Altfel, se modifică **p->next** astfel încât să indice elementul de după elementul următor lui **p** şi se eliberează spaţiul de memorie indicat de vechea valoare a lui **p->next**
(figura 15.4).

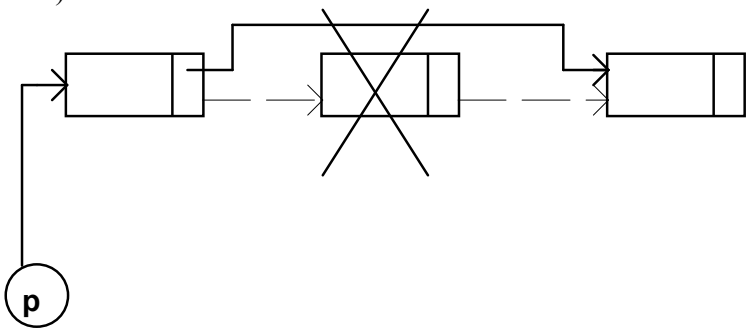


Fig.15.4. Ştergererea uni element după un element dat

d) Artificiu pentru ştergerea unui element dat

Se va utiliza acelaşi artificiu ca la inserare, adică se va copia câmpul DATA al elementului următor în elementul curent şi se va şterge elementul următor. Se observă că artificiul funcţionează numai dacă elementul dat nu este ultimul din listă.

```
void delete(LINK p)
{
if(p= =NULL||p->next= =NULL)
return;
p->data=p->next->data;
del_after(p); }
```

Funcţiile din cazurile a)..d) trebuie privite doar ca modele pentru operaţiile elementare de inserare şi ştergere. În practică, lista va fi memorată totdeauna printr-un pointer la primul element, iar funcţiile de inserare şi ştergere vor primi un asemenea pointer, întorcând un pointer la lista modificată. Elementul care se şterge din listă este specificat fie prin numărul său de ordine, fie prin câmpul DATA. Exemplele următoare reprezintă aceste situaţii.

e) Inserarea unui element dat prin câmpul DATA, înaintea primului element al unei liste, întorcând un pointer către lista modificată

```
LINK insert(LINK t, DATA x)
{
        LINK p=new_el(x, NULL);
        p->next=t;
        return p;
}
```

Apelul va fi de forma:

```
LINK t;
DATA d;
t=insert(t,d);
```

Dacă lista **t** este vidă, atunci în urma apelului, ea va conţine elementul nou creat **p**. Altfel, elementul nou creat este inserat în faţa lui **t**.

f) Inserarea unui element dat după al n-lea element al unei liste date

În această situaţie este evident că trebuie parcurse primele **n** elemente ale listei. Dacă lista are mai puţin de **n** elemente, se va insera la sfârşit; dacă lista este vidă sau **n** este 0, se va insera în faţă. Se întoarce un pointer la lista modificată. Pentru inserarea efectivă, se va folosi funcţia **insert()** din exemplul precedent.

```
LINK insert_n(LINK t, DATA x, int n)
{
        LINK q;
        if (t= =NULL || n= =0)
                return insert(t,x);
        for (q=t; q->next!=NULL&&--n; q=q->next);
        q->next=insert(q->next,x);
        return t;
}
```

Dacă **t** este NULL sau dacă **n** este 0, se inserează elementul **x** înaintea primului element, apelând **insert()**. Dacă nu, se parcurg elementele listei într-un ciclu **for**, în care condiţia de parcurgere este ca **q** să nu indice ultimul element şi să nu fi parcurs n elemente. Valoarea lui **t** trebuie păstrată, pentru că aceasta se va întoarce programului apelant.

După ce s-a încheiat parcurgerea, se inserează elementul **x**, legându-se totodată prima parte a listei (cea parcursă) cu noul element, prin atribuirea:
q->next=insert(q->next,x). În final, se întoarce **t** programului apelant.

g) Ştergerea celui de al n-lea element al unei liste date

Dacă **n** este 0, sau lista are mai puţin de **n** elemente, nu se va face nimic. Altfel, se va şterge al **n**-lea element, întorcând un pointer la lista modificată. Dacă lista avea un unic element, ea va deveni vidă.

```
LINK delete_n(LINK t, int n)
{
        LINK p,q;
        if(t= =NULL||n= =0)
                return t;
        if (n= =1)
        {
                p=t->next;
                free(t);
                return p;
        }
        for (p=q=t; --n&&p->next!=NULL; p=p->next)
                q=p;
        if (n= =0)
        {
                q->next=p->next;
                free(p);
        }
        return t;
}
```

Dacă **t** este NULL sau dacă **n** este 0, nu trebuie executată nici o operaţie. Altfel, dacă **n** este 1, trebuie şters primul element. În toate celelalte cazuri, se va întoarce **t**. Se parcurg elementele listei, condiţia de parcurgere fiind să nu fi ajuns la sfârşitul listei şi **n** să nu fi ajuns 0. Deoarece se va

efectua o operaţie de ştergere, trebuie cunoscut elementul anterior celui care se şterge, deci condiţia este **p->next!=NULL**, reţinând în **q** elementul anterior lui **p**. De asemenea, dorim evaluarea condiţiei **--n** şi în cazul în care **p->next** este NULL, astfel încât se scrie **--n** ca primă condiţie în relaţia **&&**.

Dacă la ieşirea din ciclul **for** n este mai mare ca 0, înseamnă că lista are mai puţin de **n** elemente şi nu se execută nici o operaţie. Altfel, se şterge elementul indicat de **p**.

h) Ştergerea tuturor elementelor unei liste

```
void del_list(LINK t)
{
        LINK p;
        while (t!=NULL){
                p=t;
                t=t->next;
                free(p);
                }
}
```

Trebuie remarcat că după eliberarea spaţiului de memorie indicat de un pointer, acel spaţiu nu mai poate fi folosit. Ca atare, se salvează **t** într-o variabilă intermediară **p**, apoi se execută: **t=t->next** şi **free(p)**.

Observaţie:

Listele al căror ultim nod are câmpul de legătură NULL, se numesc liste **lanţ (chain)**.

i) Aplicaţie

Să se determine numărul de apariţii ale unui întreg într-o secvenţă dată.

O soluţie posibilă constă în introducerea întregilor respectivi într-o listă simplu înlănţuită.

Fiecare număr întreg din secvenţa dată determină:

- declanşarea procesului de căutare în listă;
- incrementarea unui **contor de apariţii**, asociat fiecărui întreg.

După parcurgerea întregii secvenţe, se tipăreşte lista numerelor întregi şi valorile **contorilor de apariţii**.

Programul corespunzător este:

```
typedef struct nr_int{
        int data, count;
        struct nr_int *next;
} ELEMENT, *LINK

void search (LINK root, int a)
{
        LINK x;
        int found;
        x=root;
        found=0;
        while (x!=NULL && !found)
                {
                if (x->data= =a)
                                found=1;
                        else
                                x=x->next;
                if (found= =1)
                                x->count=x->count+1;
                        else
                        {
                        root=(LINK) malloc(sizeof(ELEMENT));
                        root->count=1;
                        root->data=a;
                        next=x;
                        }
                }
}
```

Modul de căutare utilizat în programul prezentat anterior este ineficient, deoarece pentru fiecare nod nou trebuie parcursă lista (în cel mai defavorabil caz, până la sfârşitul său). De aceea, pentru simplificarea operaţiei de căutare, se utilizează în mod curent, **listele ordonate**.

15.2. Liste ordonate

Fie o listă (de tip **chain**) ordonată în sensul descrescător al câmpului **data**. Căutarea unui nod având în câmpul **data** o valoare **a**, se încheie fie prin determinarea nodului respectiv, fie în momentul în care un nod are valoarea **v** din câmpul **data**: **v<a.**

Principiul prezentat anterior poate fi utilizat şi pentru generarea unor **liste simplu înlănţuite ordonate**. Astfel, pentru ordonarea descrescătoare a unei liste, un nod nou se inserează înaintea primului nod cu valoarea din câmpul **data** mai mică decât cea corespunzătoare noului nod. Spre deosebire de procedura **insert** prezentată anterior, în acest caz se utilizează o altă metodă de inserare **înainte**, metodă care utilizează doi pointeri, **x** şi **y**. **Pointerul y** indică spre nodul curent din listă, iar **pointerul x** spre cel anterior; nodul care trebuie introdus în listă este referit de **pointerul z**. Căutarea se încheie în momentul în care are loc inegalitatea:

y->data<z->data.

Ca urmare, nodul **z** se inserează între **x** şi **y**.

Procesul de inserare este ilustrat în figura 15.5.

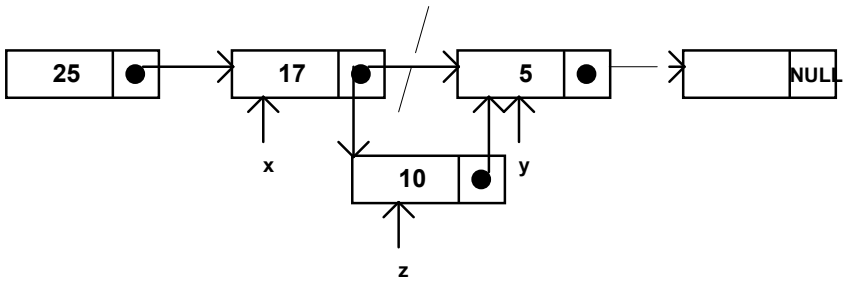


Fig. 15.5. Inserarea unui element într-o listă ordonată

Se consideră următoarele cazuri particulare:

a) z=x->next, cu excepţia cazului în care lista este vidă (când **x=NULL**, deci
x->next nu există). Pentru a nu se obţine o tratare separată a acestei situaţii, se va utiliza un nod **dummy** (lipsit de informaţie) la **începutul listei** (**root**).

b) Utilizarea celor doi pointeri pentru scanarea listei, determină existenţa a cel puţin unui element în interiorul listei (în afară de nodul **dummy**). Din figura 15.5, rezultă **condiţia de continuare a scanării listei**:

```
(y-data!=a) && (y->next!=NULL)
```

Pentru simplificarea condiţiei, se utilizează un nod **dummy** (**sentinel**) la **sfârşitul listei**.

Ca urmare, implementarea cazurilor a) şi b) impune existenţa următoarei secvenţe în programul principal:

```
root=(LINK) malloc(sizeof(ELEMENT);
sentinel=(LINK) malloc(sizeof(ELEMENT));
root->next=sentinel;
```

Astfel, procedura de căutare va utiliza două **sentinele** (**root** şi **sentinel**), la începutul şi respectiv la sfârşitul listei. Este evident că o listă vidă conţine totuşi două noduri: **root** şi **sentinel**.

Procedura de **căutare-inserare ordonată** implementată în conformitate cu observaţiile anterioare, este următoarea:

```
void search (LINK root, int a);
{
        LINK x,y,z;
        x=root;
        y=x->next;
        sentinel->data=a;
        while (y->data>a){
                        x=y;
                        y=x->next;
                        }
        if (y->data=a && y!=sentinel)
                        y->count=y->count+1;
                else
                {
                        z=(LINK) malloc(size of(ELEMENT))
                        z->data=a;
                        next=y;
                        x->next=z;
                }
}
```

15.3. Liste circulare

Listele circulare sunt liste în care câmpul **next** al ultimului nod (**ultim**) indică spre primul nod al listei (**prim**). Un exemplu de listă circulară este reprezentat în figura 15.6.

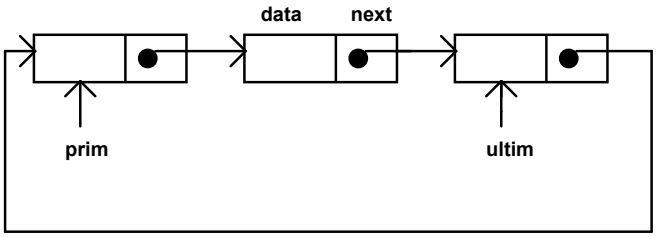


Fig. 15.6. Listă circulară

Listele circulare sunt utilizate în cazul în care programatorul gestionează nodurile libere ale listelor prin intermediul unor **cozi** (**L**iste) de **N**oduri **D**isponibile (**LND**). În acest caz, **LND** este structurată ca **chain**, în timp ce lista curentă este **circulară**.

Eliminarea unui nod din lista curentă este realizată prin inserarea sa în lista **LND** (nu se mai utilizează instrucţiunea **dispose**).

Pentru **generarea unui nou nod** în lista curentă, se examinează **LND**; dacă **LND** este nevidă, se extrage un nod din **LND**; în caz contrar, trebuie apelată procedura **new**.

Pentru a ilustra modalităţile de **generare/eliberare a unui nod dintr-o listă circulară**, se consideră lista **LND** spre care indică **pointerul LND** şi **nodul curent - x,** din lista analizată.

Procedurile corespunzătoare sunt:

```
void getnode (LINK x);
{
        if (LND= =NULL)
                x=(LINK) malloc(sizeof(ELEMENT))
                else {
                        x=LND;
                        LND=LND->next;
                }
}
```

```
void freenode(LINK x)
{
        x->next=LND;
        LND=x;
}
```

În acest context, rezultă evident avantajul utilizării listelor circulare: **ştergerea unei liste circulare se face în timp constant, indiferent de numărul de noduri ale listei.**

Ştergerea unei liste circulare poate fi urmărită în figura 15.7.

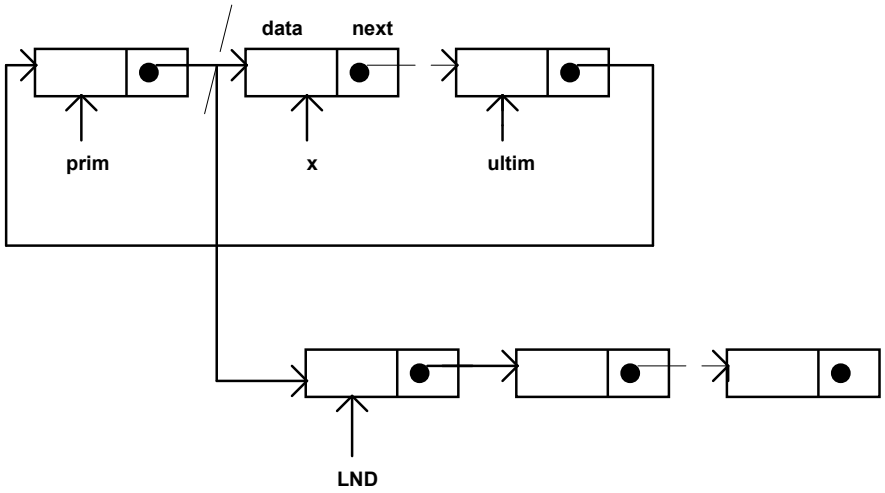


Fig. 15.7. Ştergerea unei liste circulare

Procedura corespunzătoare este:

```
void cerase (LINK prim)
{
        LINK x;
        if (prim!=NULL){
                x=prim->next; (*nodul al doilea*)
                prim->next=LND; (*primul nod se leaga la LND*)
                LND=x; (*al doilea nod devine primul nod al
LND*)
        }
```

```
}
```

15.4. Proceduri suplimentare pentru liste

a) Inversarea elementelor unei liste chain

Se defineşte o procedură care realizează inversarea elementelor unei liste **chain**, utilizând trei pointeri:

```
void  invert (LINK prim)
/* lista spre al carei prim element indica pointerul prim se
inverseaza astfel încât x=(a1, ... ,an) devine dupa inversare x=(an, ...
,a1)*/
{
        LINK x,y,z;
        x=prim;
        y=NULL; (*y urmeaza dupa x*)
        while (x!=NULL) {
                z=y; y=x; (*z urmeaza dupa y*)
                x=x->next; (*se avanseaza la urmatorul element
al listei*)
                y->next=z; (*leaga nodul y la nodul precedent*)
                }
}
```

Justificarea procedurii prezentate anterior rezultă din figura 15.8

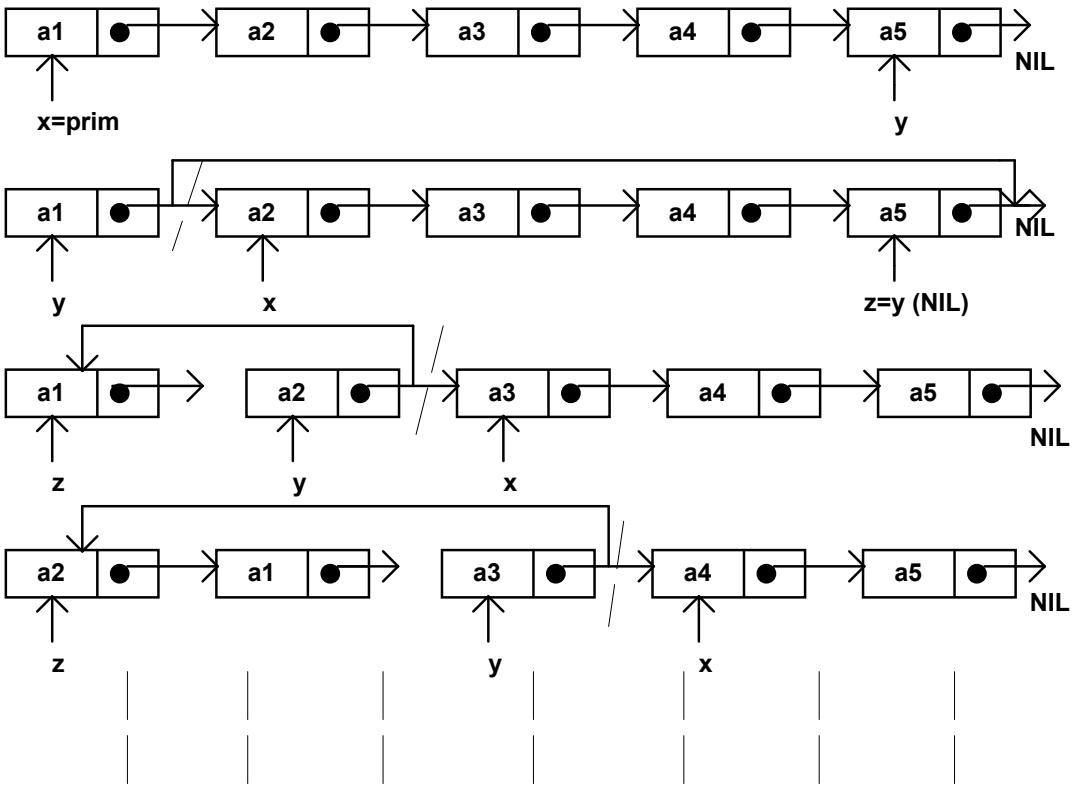


Fig. 15.8 Inversarea elementelor unei liste chain

Pentru o listă având **m** noduri, bucla **while** se execută de **m** ori. Presupunând că timpii de execuţie asociaţi instrucţiunilor sunt practic compatibili, rezultă că timpul total de prelucrare pentru procedura prezentată anterior, este ***O*(m)**.

b) Concatenarea a două liste chain

Se consideră două liste simplu înlănţuite de tip **chain**, având primele noduri referite de pointerii **x** şi respectiv, **y**. Cele două liste sunt:

x=(a1, ... , am) şi respectiv **y=(b1, ... , bn)**, unde **m, n≥0**.

Problema construirii unei liste **z=(a1, ... , am, b1, ... , bn)** este rezolvată prin intermediul procedurii:

```
void concatenate (LINK x,y,z)
{
        LINK p;
        if (x= =NULL)
                        z=y;
                else {
                        if (x!=NULL){
                                p=x;
                                while (p->next!=NULL)
                                                p=p->next;
                                /*se cauta ultimul nod al listei x*/
                                p->next=y;
        /*se leaga ultimul nod al listei x la primul nod al listei y*/
                                }
                        }
}
```

unde: x este pointer spre primul element al listei **x**;
y este pointer spre primul element al listei **y**;
z este pointer spre primul element al listei **z**.

15.5. Stive şi cozi organizate ca liste înlănţuite

În cazul în care este necesară utilizarea simultană a **n cozi** şi **m stive**, reprezentarea secvenţială nu este eficientă; ca urmare, se optează pentru implementarea prin liste simplu înlănţuite.

Direcţia link-urilor este aleasă astfel încât operaţiile de inserare şi ştergere a nodurilor, să se execute cât mai simplu.

În figura 15.9 sunt reprezentate o stivă, respectiv o coadă, organizată ca listă înlănţuită.

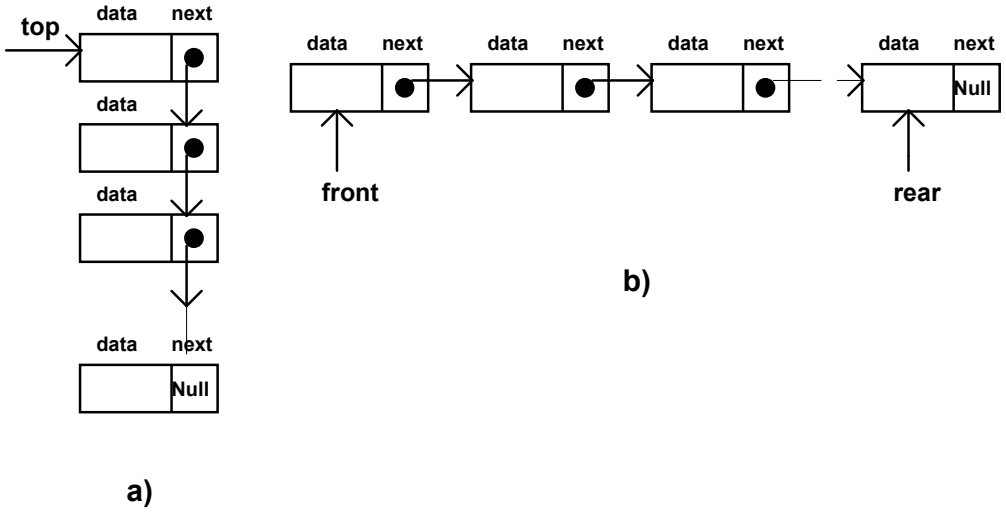


Fig. 15.9. a) Stivă organizată ca listă înlănţuită
b) Coadă organizată ca listă înlănţuită

Definiţiile de tip şi de variabile necesare pentru implementarea simultană a stivelor înlănţuite**,** sunt următoarele:

```
typedef ... DATA;
typedef struct stack{
        DATA data;
        struct stack *next;
        } ELEMENT, *TOP;
```

O stivă se va preciza printr-un pointer de tip **TOP** către elementul din vârful stivei. Un pointer NULL va indica o stivă vidă.

Crearea unui element al stivei se va realiza cu funcţia **new_el()**:

```
TOP new_el (DATA d)
{
        TOP t=(TOP) malloc(sizeof(ELEMENT));
        if (t= =NULL)
                err_exit("new_el:Eroare alocare");
        t->data=d; t->next=NULL;
}
```

Testarea stivei vide se poate face cu funcţia **vid()**:

```
int vid (TOP t) {return t= =NULL;}
```

Furnizarea elementului din vârful stivei (fără extragere) se poate face cu funcţia **vtop()**:

```
DATA vtop (TOP t)
{
        if (t= =NULL)
                err_exit("vtop:Stiva vida")
        return t->data;
}
```

În ceea ce priveşte funcţiile **push()** şi **pop()**, trebuie observat că ambele modifică structura stivei, deci fie se întoarce pointerul de tip **TOP** la stiva modificată prin funcţie, fie se transmite ca parametru formal o referinţă la un pointer de tip **TOP**. De asemenea, funcţia **pop()** trebuie să furnizeze şi elementul din vârful stivei.

```
TOP push (TOP t, DATA d)
{
        TOP p=new_el(d);
        p->next=t;
        return p;
}

TOP pop (TOP t, DATA *pd)
{
        TOP p=t;
        if (t= =NULL)
                err_exit("pop: Stiva vida");
        t=t->next;
        *pd=p->data;
        free(p);
        return(t);
}
```

În programul principal, apelul acestor proceduri se realizează prin:

```
TOP s; DATA d;
```

```
s=push(s,d);
s=pop(s,&d);
```

În cazul implementării cozilor ca liste înlănţuite, se consideră definite tipurile DATA, ELEMENT şi LINK, la fel ca la liste. De asemenea, se consideră funcţia **new_el()** definită similar ca în cazul listelor.

Pointerii **front** şi **rear** către primul, respectiv către ultimul element al cozii sunt de tip LINK. O coadă vidă va fi definită prin **front=rear=NULL**.

Se defineşte:

```
typedef struct queue {LINK front; LINK rear;} *QUEUE;
```

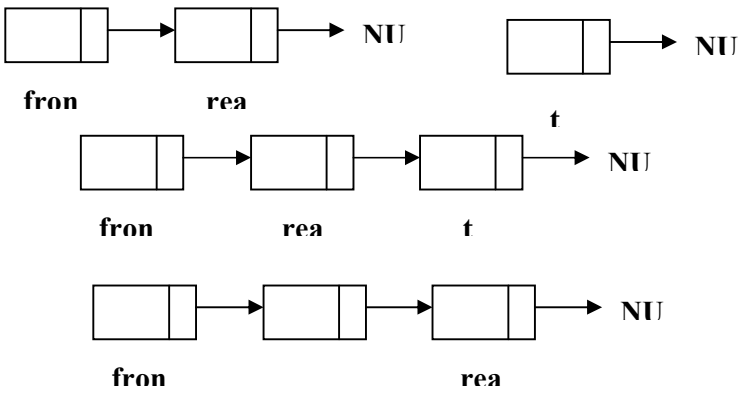
Funcţia de iniţializare a unei cozi înlănţuite este:

```
QUEUE init_queue(void)
{
        QUEUE q=(QUEUE) malloc(sizeof(struct(queue));
        if (q= =NULL)
                err_exit("init_queue:Eroare alocare");
        q->front=q->rear=NULL;
        return q;
}
```

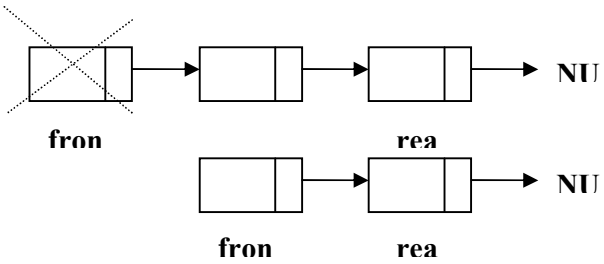
Funcţiile de prelucrare sunt **add(), extr(), front()** şi **rear()** care adaugă, extrag, întorc primul şi respectiv ultimul element fără a-l extrage din coadă.

```
void add (QUEUE q, DATA d)
{
        LINK t=new_el(d);
        if (q->rear= =NULL)
                q->rear=q->front=t;
        else {
                q->rear->next=t;
                q->rear=t;
        }
}
```

```
DATA extr (QUEUE q)
{
        DATA d; LINK t;
        if (q->front= =NULL)
                err_exit("extr: Coada vida");
        d=(t=q->front)->data;
        if (t->next= =NULL)
                q->front=q->rear=NULL;
        else
                q->front=q->front->next;
        free(t);
        return (d);
}
```



1. Inserare dupa



2. Stergere inainte

Figura 15.9 c. Inserarea şi stergerea elementelor dintr-o coadă implementată dinamic

```
DATA front (QUEUE q)
{
        if (q->front= =NULL)
                err_exit("front:Coada vida");
        return q->front->data;
}


DATA rear (QUEUE q)
{
        if (q->rear= =NULL)
                err_exit("rear:Coada vida")
        return q->rear->data;
}
```

Soluţia prezentată anterior pentru implementarea stivelor şi cozilor înlănţuite, este relativ simplu de prelucrat. Nu este necesară deplasarea stivelor sau cozilor în scopul obţinerii de zone de memorie disponibile. Prelucrările se desfăşoară atâta timp cât există noduri libere. În comparaţie cu varianta secvenţială, este necesară o zonă de memorie suplimentară, datorită câmpurilor **next**, care asigură legătura dintre stive şi cozi. Există însă şi cazuri în care câmpurile **next** şi **data** se pot reprezenta în acelaşi cuvânt de memorie; în aceste situaţii, zona de memorie ocupată este identică pentru cele două tipuri de reprezentare a stivelor şi cozilor: secvenţială sau cu liste înlănţuite.

15.6. Liste dublu înlănţuite

Pentru a înlătura dezavantajele listelor simplu înlănţuite determinate de parcurgerea într-un singur sens a listei şi de dificultatea stabilirii predecesorului unui nod, sunt utilizate **listele dublu înlănţuite**.

În cadrul **listelor dublu înlănţuite, fiecare nod are două câmpuri de legătură**: unul spre nodul următor, celălalt spre nodul anterior.

Listele dublu înlănţuite sunt de tip **lanţ (chain)** sau **circulare.**

Definiţia de tip pentru o listă dublu înlănţuită este:

```
typedef struct elem {
        DATA data;
```

```
        struct elem *left, *right;
}ELEMENT, *LINK;
```

Un nod într-o listă dublu înlănţuită are deci trei câmpuri: **data, left** şi **right** Pentru programator, este convenabil ca listele dublu înlănţuite să utilizeze un **nod cap** (**header**) al cărui câmp de date **nu conţine informaţii**. O listă vidă conţine deci nodul cap, după cum s-a reprezentat şi în figura 15.10.

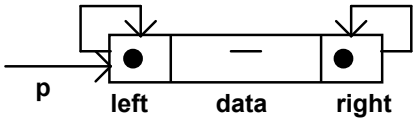


Fig. 15.10 Reprezentarea nodului cap al unei liste dublu înlănţuite

O structură de listă dublu înlănţuită având trei noduri şi un **nod cap**, este reprezentată în figura 15.11.

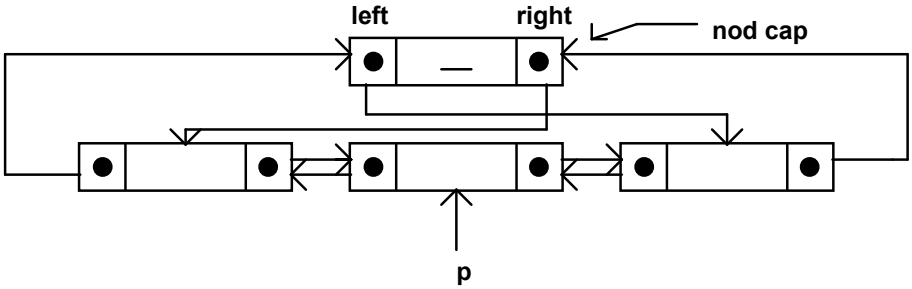


Fig. 15.11. Reprezentarea listelor dublu înlănţuite

Se consideră un nod al unei liste dublu înlănţuite, referit de pointerul **p**. Pentru acest nod, sunt satisfăcute egalităţile:

```
p=p->right->left=p->left->right
```

care evidenţiază posibilităţile de deplasare bidirecţională în listă.

Funcţiile de prelucrare vor primi ca parametru header-ul listei şi vor întoarce headerul către lista modificată.

O primă funcţie este cea care construieşte un header, adică o listă vidă:

```
LINK new_h (void)
```

```
{
        LINK h=(LINK) malloc(sizeof(ELEMENT));
        if (h= =NULL)
                err_exit("new_h:Eroare alocare");
        h->left=h->right=h;
        return h;
}
```

Funcţia care construieşte un element al listei, primind valoarea câmpului DATA, este:

```
LINK new_el (DATA d)
{
        LINK p=(LINK) malloc(sizeof(ELEMENT));
        if (p=  =NULL)
                err_exit("new_el: Eroare alocare");
        p->data=d;
        p->left=p->right=p;
        return p;
}
```

Testarea unei liste vide se poate face cu funcţia:

```
int is_empty (LINK h)
{
        return h= =h->right;
}
```

Formele generale de parcurgere, pentru ambele sensuri, sunt:

```
void parcurge_lr (LINK h)
{
        LINK p;
        for (p=h->right; p!=h; p=p->right)
                prel(p);
}

void parcurge_rl (LINK h)
{
        LINK p;
        for (p=h->left p!=h; p=p->left)
```

```
            prel(p);
}
```

în care s-a presupus că **prel()** este o funcţie care prelucrează elementul curent al listei. De exemplu, o funcţie care tipăreşte toate elementele listei este:

```
void print_list (LINK h)
{
        LINK p;
        for (p=h->right; p!=h; p=p->right)
                print_data(p->data);
        printf("NULL\n");
}
```

a) Inserarea unui element la dreapta unui element dat

```
void ins_right (LINK h, LINK p)
{
        p->right=h->right;
        p->left=h;
        h->right=p;
        p->right->left=p;
}
```

Se inserează **p** la dreapta lui **h** (fig. 15.12). Se observă că **h** poate fi un header de listă sau un element oarecare al listei.

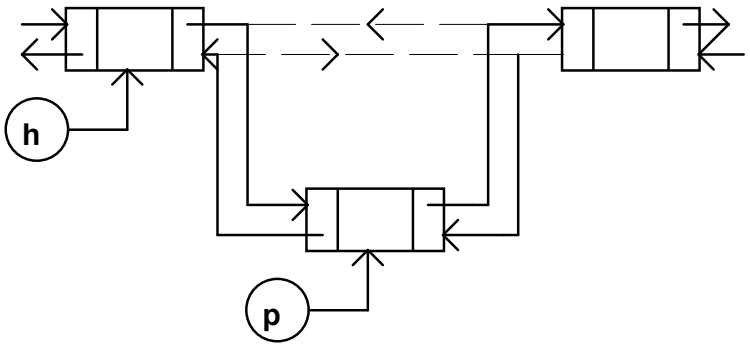


Fig.15.12 Inserare la dreapta unui element dat intr-o lista dublu înlănţuită

b) Inserarea unui element la stânga unui element dat

```
void ins_left (LINK h, LINK p)
{
        p->left=h->left;
        p->right=h;
        h->left=p;
        p->left->right=p;
}
```

c) Ştergerea unui element dat

```
void delete (LINK p)
{
        p->right->left=p->left;
        p->left->right=p->right;
        free(p);
}
```

Atribuirile de pointeri sunt evidente (fig.15.13). Această funcţie nu trebuie folosită cu **p** header de listă, decât dacă acesta reprezintă o listă vidă.

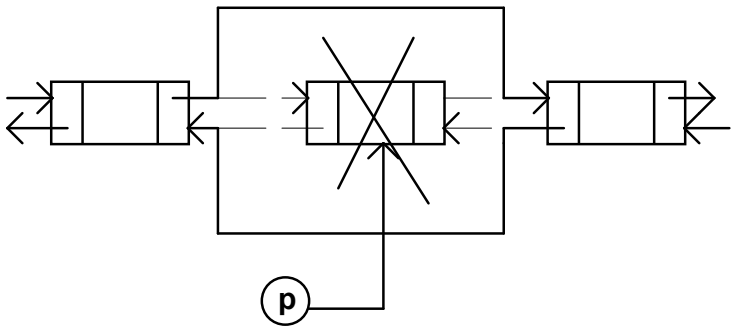


Fig.15.13 Ştergerea unui element într-o listă dublu înlănţuită

d) Ştergerea elementului de la stânga unui element dat

```
void del_left (LINK h)
{
        if (h= =h->left)
                return;
        p=h->left;
        p->left->right=h;
        h->left=p->left;
        free(p);
}
```

Această funcţie se poate folosi şi când **h** este header de listă. Trebuie remarcat că , dacă lista are un singur element propriu-zis, în urma apelului funcţiei, header-ul **h** va reprezenta lista vidă. La o listă cu un element, acesta este atât la dreapta, cât şi la stânga header-ului.

Absolut similar, se scrie şi funcţia de ştergere la dreapta. În practică, funcţiile de ştergere dreapta/stânga se folosesc rar, deoarece se poate realiza ştergerea unui element dat, prin funcţia **delete()**.

e) Inserarea unui element dat prin DATA la stânga unui element dat

```
void insert_l (LINK h, DATA x)
{
        LINK p=new_el(x);
        h->left->right=p;
        p->left=h->left;
        h->left=p;
        p->right=h;
}
```

Funcţia se poate folosi şi când **h** este un header.

16
Arbori

16.1. Definiţii

Un arbore reprezintă o mulţime finită de elemente, având unul sau mai multe noduri cu proprietăţile:

a) există un nod special, denumit **rădăcină**;

b) nodurile rămase după extragerea rădăcinii sunt partiţionate în **n≥0** mulţimi disjuncte: **T1, T2, ... , Tn**, unde fiecare dintre aceste mulţimi reprezintă un arbore.

T1, T2, ... , Tn sunt denumiţi **subarbori ai rădăcinii**.

Definiţia anterioară este recursivă (este asemănătoare cu definiţia listelor generalizate).

O posibilă reprezentare a arborilor este cea din figura 16.1.

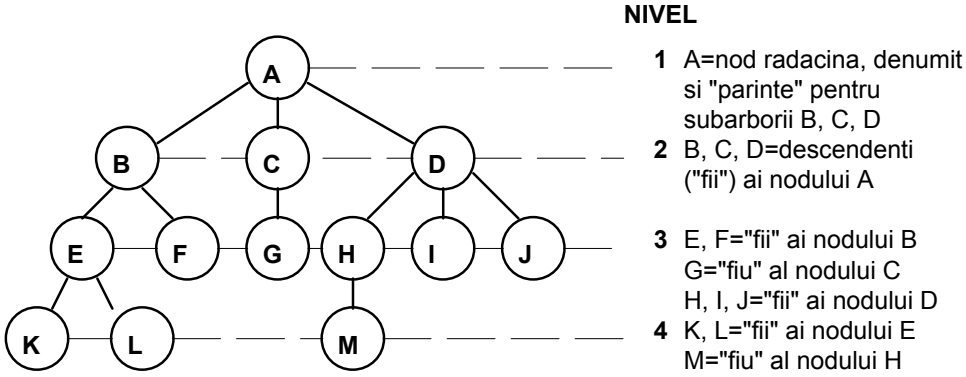


Fig. 16.1. Reprezentarea arborilor

Modul de operare cu arbori impune definirea următoarelor noţiuni:

a) **gradul unui nod** - reprezintă numărul de subarbori ai nodului (pentru arborele din figura 16.1, gradul nodului **A** este 3, al nodului **B** este 2, al nodului **C** este 1, etc.);

b) **nivelul unui nod** - este definit considerându-se nodul rădăcină pe nivelul 1; dacă un nod (**părinte**) se află pe nivelul **p**, atunci descendenţii săi se află pe nivelul **p+1**;

c) **nodul-frunză (nod terminal**) este un nod cu gradul 0 (fără descendenţi); de exemplu, pentru arborele din figura 16.1, nodurile-frunză sunt: **K, L, F, G, M, I, J**;

d) **gradul unui arbore** - este definit ca **maxim{grad(nod)}**, unde **nod** $\in$**arbore** (maximul gradului unui nod din arbore); pentru arborele reprezentat în figura 16.1, gradul este egal cu 3;

e) **adâncimea unui arbore** - reprezintă nivelul maxim al nodurilor-frunză; pentru arborele reprezentat în figura 16.1, adâncimea este egală cu 4;

f) **pădurea** reprezintă o mulţime de arbori disjuncţi. Noţiunea de **pădure** este strâns corelată cu cea de **arbore**, deoarece îndepărtarea rădăcinii unui arbore determină obţinerea unei păduri; de exemplu, îndepărtarea nodului **A** din arborele reprezentat în figura 16.1, determină obţinerea unei păduri cu trei arbori;

g) **nodul interior** - reprezintă orice nod neterminal;

h) **lungimea drumului până la nodul x** - reprezintă numărul de noduri (**arce**) care trebuie traversate pentru a se ajunge de la rădăcină, la nodul **x**;

Observaţie: rădăcina are lungimea drumului egală cu 1, iar descendenţii săi direcţi au lungimea drumului egală cu 2. În general, un nod aflat pe un nivel **i**, are lungimea drumului egală cu **i**;

i) **lungimea traiectoriei interne** - reprezintă suma lungimilor drumurilor de la rădăcină până la toate nodurile unui arbore.

Lungimea medie a traiectoriei interne este:

$P_i = \frac{1}{n} \sum_i n_i i$, unde $\mathbf{n_i}$ reprezintă numărul de noduri de pe nivelul **i**.

Pentru arborele reprezentat în figura 16.1, lungimea medie a traiectoriei interne este:

Pi = (1+3x2+6x3+3x4)/13

j) În cele ce urmează se defineşte **lungimea traiectoriei externe**. În acest scop, într-un arbore de ordin **m**, se completează fiecare nod având un număr **n<m** subarbori, cu **m-n noduri suplimentare (terminale)**, ca în figura 16.2. Astfel, toate nodurile interne ale arborelui au în final, ordinul **m.**

Lungimea traiectoriei externe reprezintă suma lungimilor traiectoriilor de la rădăcină până la nodurile suplimentare introduse.

Lungimea medie a traiectoriei externe este:

$P_e = \frac{1}{m}\sum_i m_i i$, unde $\mathbf{m_i}$ reprezintă numărul de noduri suplimentare de pe nivelul **i**.

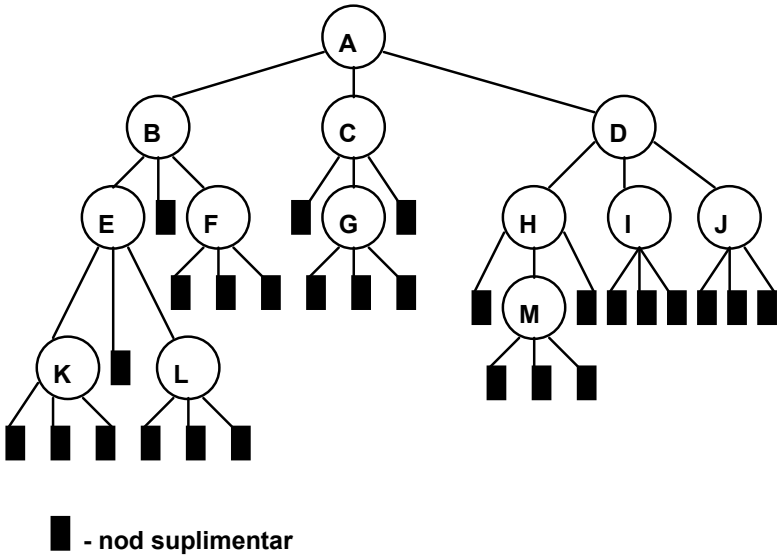


Fig. 16.2. Reprezentarea arborilor pentru calculul lungimii traiectoriei externe

16.2. Arbori binari

a) Definiţie: un arbore binar reprezintă o mulţime finită de noduri, care poate fi:

1) vidă;

2) nevidă, constând **dintr-o rădăcină şi doi arbori binari disjuncţi**, denumiţi **subarborele stâng** şi **subarborele drept**.

Se observă că un arbore binar este un arbore **de gradul 2**.

b) Reprezentarea arborilor binari

În figura 16.3 sunt prezentate două exemple de arbori binari.

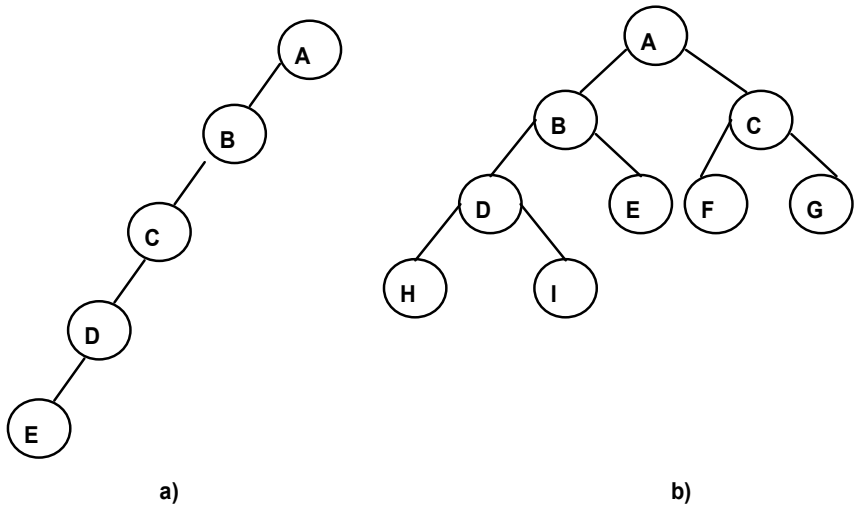


Fig. 16.3. Exemple de arbori binari

Pentru arborii binari sunt posibile următoarele reprezentări:

1) **Reprezentarea secvenţială**

Reprezentarea secvenţială se realizează alocând pentru nodurile de pe **nivelul k, 2^{k-1} locaţii** în tabloul asociat arborelui respectiv. În figura 16.4 sunt ilustrate reprezentările secvenţiale pentru arborii din figura 16.3. Se observă utilizarea ineficientă a memoriei în cazul arborilor incompleţi (figura 16.3.a). De asemenea, inserarea sau ştergerea de noduri implică deplasarea unui număr mare de elemente.

2) **Reprezentarea cu liste înlănţuite**

Definiţiile de tip corespunzătoare acestui tip de reprezentare sunt următoarele:

```
struct node {
        DATA d;
        struct node *left;
        struct node *right; };
typedef struct node NODE, *BTREE;
```

unde **left** şi **right** reprezintă pointeri spre subarborii stâng/ drept ai unui nod.

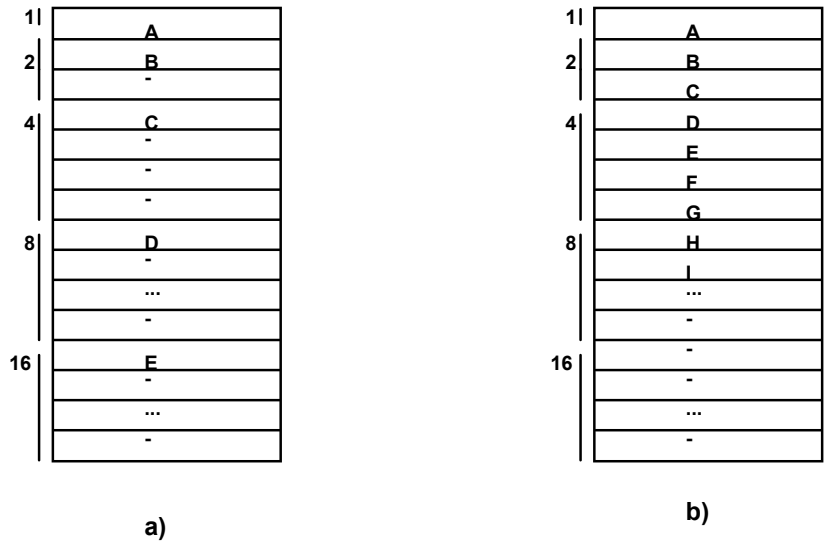


Fig. 16.4. Exemple de reprezentare secvenţială a arborilor binari

Deci un nod al listei înlănţuite asociate arborelui binar poate fi reprezentat ca în figura 16.5.

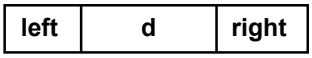


Fig. 16.5. Structura nodurilor listelor înlănţuite asociate arborilor binari

Pentru arborii binari din figura 16.3, reprezentările cu liste înlănţuite sunt ilustrate în figura 16.6.

Un arbore este referit printr-un pointer spre rădăcina sa (**root**).

Definiţie: un **arbore perfect echilibrat** este un arbore la care pentru orice nod, numărul de noduri ai subarborilor stâng şi drept diferă cel mult cu o unitate.

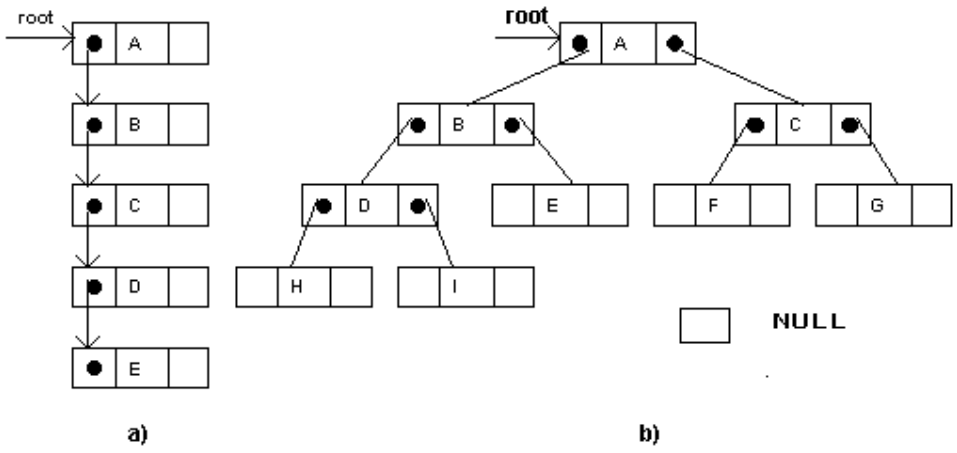


Fig. 16.6. Exemple de reprezentare cu liste înlănţuite a arborilor binari

Definiţie: un **arbore ordonat** este un arbore la care ramurile fiecărui nod sunt ordonate. Astfel, arborii ordonaţi (reprezentaţi în figura 16.7a şi b) sunt diferiţi.

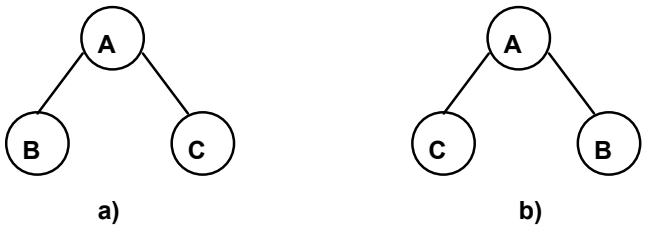


Fig. 16.7. Exemple de arbori ordonaţi

16.3. Operaţii de bază asupra arborilor binari

a) Ca şi la structurile de date anterioare, o primă funcţie este cea care construieşte un nod:

```
BTREE new_node(DATA d)
{
        BTREE t = (BTREE) malloc(sizeof(NODE));
        if (t= =NULL)
                err_exit("new_node:Eroare alocare");
```

```
        t->d=d;
        t->left=t->right=NULL;
        return t;
}
```

Uneori este utilă o funcţie constructor care primeşte atât câmpul de date cât şi cele două câmpuri de legătură:

```
BTREE init_node(DATA d, BTREE left, BTREE right)
{
        BTREE t=new_node(d);
        t->left=left;
        t->right=right;
        return t;
}
```

b) Generarea unui arbore binar perfect echilibrat

Procedura de generare a unui arbore binar perfect echilibrat se bazează pe definiţiile anterioare şi pe următorul algoritm care se desfăşoară în trei etape:

- specificarea nodului rădăcină;
- generarea subarborelui stâng, având **nl=n div 2** noduri (**n** reprezintă numărul de noduri ale arborelui);
- generarea subarborelui drept, având **nr=n-nl-1** noduri.

Se observă că algoritmul descris anterior este recursiv. Implementarea sa se va realiza prin intermediul unei funcţii care întoarce un pointer spre rădăcina subarborelui nou creat:

```
BTREE buildtree (int n)
{
        BTREE x;
        int a, nl, nr;
        if (n= =0)
                        buildtree=NULL;
                else {
                        nl=n/2;
                        nr=n-nl-1;
                        a=getch();
                BTREE x=(BTREE) malloc(sizeof(NODE));
                        x->d=a;
```

```
                x->left=buildtree(nl);
                x->right=buildtree(nr);
            }
    }
```

Exemplu: utilizând **function buildtree**, să se construiască arborele binar perfect echilibrat, cu nodurile: 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18.

Arborele rezultat ca urmare a apelului funcţiei **buildtree** este prezentat în figura 16.8.

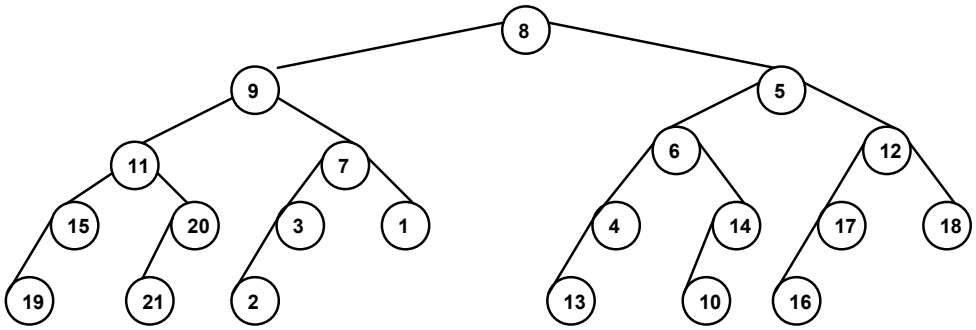


Fig. 16.8. Exemplu de arbore binar perfect echilibrat

c) Traversarea arborilor binari

În mod similar parcurgerii listelor simplu înlănţuite, **operaţia de traversare a arborilor binari** presupune **executarea aceleiaşi secvenţe de instrucţiuni** (în mod uzual, tipărirea informaţiei din nod) **pentru toate nodurile arborelui**. Se menţionează că fiecare nod trebuie parcurs o singură dată. Notând cele trei câmpuri ale unui nod cu **L, D, R** (unde **D** este rădăcina arborelui, iar **L** şi **R** reprezintă subarborii stâng şi drept), se stabileşte prin convenţie că **traversarea arborilor parcurge întâi L şi apoi R**. Ca urmare, rezultă trei posibilităţi diferite de traversare a arborilor:

- **D-L-R - traversarea în preordine**: se "vizitează" întâi nodul curent (se tipăreşte informaţia din nod), se traversează subarborele stâng şi apoi subarborele drept;
- **L-D-R - traversarea în ordine;**

- L-R-D - traversarea în postordine.

Exemplu: se consideră arborele binar din figura 16.9.

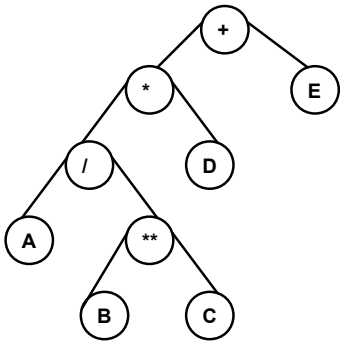


Fig. 16.9. Traversarea arborilor binari

Ca urmare a **traversării în ordine**, se obţine următoarea expresie aritmetică:

A/BC*D+E**, corespunzătoare **formei infixate de scriere a expresiilor aritmetice.**

Traversarea în postordine conduce la expresia aritmetică:

ABC/D*E+**,

care corespunde **formei postfixate de scriere a expresiilor aritmetice**.

Din cele prezentate anterior, rezultă o nouă **metodă de conversie a expresiilor aritmetice din forma infixată în forma postfixată**:

- se construieşte arborele binar asociat expresiei infixate;
- se parcurge arborele respectiv în postordine.

Traversarea în preordine conduce la următoarea expresie aritmetică:

+*/ABCDE**,

care corespunde **formei prefixate de scriere a expresiilor aritmetice** (în care operatorii se scriu înaintea operanzilor).

Se observă că definiţiile celor trei metode sunt recursive, ceea ce sugerează imediat o imlementare recursivă. Dacă presupunem o funcţie **visit** care prelucrează un nod, cu prototipul:

```
void visit (BTREE);
```

cele trei metode se implementează recursiv astfel:

```
void postordine (BTREE t)
/*t = pointer catre un nod oarecare din arbore; pentru
parcurgerea întregului arbore, se apelează procedura cu
parametrul root*/
{
        if (t!=NULL) {
        postordine(t->left); /*traversare subarbore stâng*/
        postordine(t->right); /*traversare subarbore drept*/
        visit(t); /*"vizitare" nod curent*/

        }
}

void inordine (BTREE t)
{
if (t!=NULL) {
        inordine(t->left);
        visit(t);
        inordine(t->right);
        }
}



void preordine (BTREE t)
{
if (t!=NULL) {
        visit(t);
        preordine(t->left);
        preordine(t->right);
        }
}
```

Este posibilă construcţia şi a unor **algoritmi nerecursivi pentru traversarea arborilor binari**. În acest scop, sunt definite următoarele metode:

1) scrierea procedurii recursive de traversare a arborilor binari, urmată de aplicarea regulilor de conversie de la algoritmul recursiv la cel nerecursiv;

2) implementarea directă a algoritmului nerecursiv. Se consideră **traversarea în inordine**. Se folosesc funcţiile **init_stack(), push(), pop()** şi **is_empty()** de iniţializare a unei stive, de introducere a unui element în stivă, de extragere a unui element în stivă şi de test a stării "stivă vidă". Tipul de date folosit pentru stivă (datele care se introduc în stivă) este **BTREE**, deci se vor stoca în stivă referinţe la nodurile arborelui. Funcţiile **push()** şi **pop()** vor avea deci prototipurile.

```
void push (STACK, BTREE);
BTREE pop(STACK);
```

unde tipul **STACK** este definit prin:

```
typedef struct stack{
        int sp;
        BTREE cont[MAX];
} *STACK;
```

Rezultă următoarea funcţie de parcurgere iterativă, în inordine, a unui arbore binar:

```
void i_inordine(BTREE t)
{
        STACK s=init_stack();
        while (1) {
                while (t!=NULL) {
                        push(s,t);
                        t=t->left;
                }
                if (is_empty(s))
                        break;
                        else{
```

```
                    t=pop(s);
                    visit(t);
                    t=t->right;
                    }
        }
        free(s);
}
```

Se iniţializează stiva **s**, după care se intră într-un ciclu infinit. Cât timp **t** este diferit de NULL, se "împinge" **t** în stivă şi se atribuie lui **t** adresa descendentului său stâng. În felul acesta, în stivă vor exista descendenţii stângi succesivi ai rădăcinii. Dacă în acest moment stiva este vidă, algoritmul s-a terminat (instrucţiunea **break** provoacă ieşirea din ciclul infinit şi se revine în programul apelant). Dacă stiva nu e vidă, se extrage un element, se prelucrează cu funcţia **visit** şi se atribuie lui **t** adresa descendentului său drept, reluându-se algoritmul cu această valoare a lui **t**.

Funcţia următoare implementează algoritmul de parcurgere în **preordine**, într-o manieră nerecursivă.

```
void i_preordine(BTREE t)
{
STACK s=init_stack();
while (1) {
        while (t!=NULL) {
                visit(t)
                push(s,t);
                t=t->left;
        }
        if (is_empty(s))
                break;
        else{
                t=pop(s);
                t=t->right;
        }
}
        free(s);
}
```

Realizarea parcurgerii în **postordine** pe cale nerecursivă este mai complicată, deoarece, conform definiţiei, trebuie parcurşi atât subarborele stâng, cât şi cel drept, înainte de vizitarea nodului. Aceasta presupune două

salvări/restaurări în stivă ale fiecărui nod, una pentru a avea acces la legătura din dreapta şi una pentru a vizita nodul, după ce s-au parcurs ambii subarbori. De asemenea, este necesar să se realizeze o distincţie între aceste două salvări/restaurări ale nodului în stivă, deoarece vizitarea se face la a doua restaurare.

Ca urmare, se adaugă o variabilă logică **d** la fiecare BTREE salvat în stivă, **d= =0**
semnificând prima salvare, iar **d= =1** pe a doua. Deci, în stivă va trebui să se memoreze o structură formată dintr-o variabilă BTREE şi o variabilă întreagă. Definiţiile de tipuri şi prototipuri pentru funcţiile de lucru cu stiva sunt următoarele:

```
typedef  struct elem {BTREE t; int d} ELEM;
typedef struct stack {
        int sp;
        ELEM cont[MAX];
} *stack;

void push (STACK, ELEM);
ELEM pop(STACK);

void i_postordine(BTREE t)
{
        STACK s=init_stack();
        ELEM e;

        while (1) {
                while (t!=NULL) {
                        e.t=t; e.d=0;
                        push(s,t);
                        t=t->left;
                }
                if (is_empty(s))
                        break;
                while (!is_empty(s)) {
                        e=pop(s);
                        if (e.d= =0) {
                                e.d=1;
                                push(s,e); t=e.t; t=t->right;
                                break;
                                }
```

```
                else
                        visit (e.t);
                }
        }
        free(s);
}
```

Algoritmul constă dintr-o buclă **while(1)** care se termină în momentul în care stiva **s** este vidă. Ca şi în cazurile precedente, se parcurge arborele pe ramurile din stânga şi salvând în stivă toate nodurile vizitate, cu **d** egal cu 0. Dacă stiva este vidă, algoritmul se termină. Dacă nu, se intră într-o buclă **while** care se execută atâta timp cât **s** nu este vidă, în care se extrage câte un element din stivă. Dacă variabila **d** asociată este 0, atunci i se atribuie valoarea 1 şi se inserează elementul respectiv în stivă. Totodată se trece la legătura către subarborele drept şi se reia parcurgerea pe ramura din stânga (prima buclă **while**). Această operaţie se implementează printr-un **break** care forţează ieşirea din bucla **while** curentă, dar nu şi din cea exterioară (se trece la **while(t!=NULL)**). Dacă **d** este 1, atunci se vizitează nodul curent şi se continuă extragerea elementelor din stivă.

Se observă că algoritmul se încheie atunci când **t** este NULL (nu se face nici un **push** în prima buclă **while**) şi **s** este vidă (s-au extras toate elementele). Se observă, de asemenea, că la vizitare, nu se mai atribuie lui **t** valoarea lui **e.t** (extrase din stivă), ceea ce lasă în final pe **t** la valoarea NULL, asigurând terminarea algoritmului.

d) Tipărirea arborilor binari

Se va prezenta în continuare, o funcţie de afişare a unui arbore binar sub formă indentată, în preordine. Se vor tipări un număr variabil de spaţii libere înainte de afişarea unui nod, în funcţie de nivelul nodului respectiv. Se va prezenta mai întâi o funcţie auxiliară, care primeşte un pointer la un nod şi un întreg reprezentând nivelul nodului respectiv. Funcţia **print_node()** afişează efectiv câmpul de tip DATA al nodului, fiind evident dependentă de acest tip de date.

```
void print_tree_1 (BTREE t, int n)
{
        int i;
        for (i=0; i<n; i++)
                printf(" ");
        if (t!=NULL) {
```

```
                print_node (t->d);
                print_tree_1 (t->left, n+1);
                print_tree_1 (t->right, n+1);
        }

        else
                printf("NULL\n");
}
```

Se observă apelul recursiv al funcţiei, pentru subarborii stâng şi drept. Se poate scrie acum, funcţia de tipărire a unui arbore binar, astfel:

```
void print_tree (BTREE t)
{
        print_tree_1 (t, 0);
        putchar('\n');
}
```

e) Căutarea/inserarea unui element în arborii de căutare

Definiţie: **arborii de căutare** sunt arbori binari în care există o relaţie de ordine pe mulţimea elementelor DATA, deci a câmpurilor de date conţinute în noduri şi care verifică următoarele proprietăţi:

- câmpurile de date ale nodurilor conţin valori distincte;
- prin convenţie, un arbore vid este arbore de căutare;
- rădăcina conţine o valoare mai mare decât toate nodurile subarborelui stâng şi mai mică decât toate valorile din nodurile subarborelui drept;
- subarborii stâng şi drept sunt arbori de căutare.

Prima proprietate importantă a arborilor de căutare, care rezultă chiar din definiţie, este că parcurgerea **în inordine** a unui asemenea arbore va ordona elementele în sens crescător. Această proprietate este folosită în metodele de sortare internă bazate pe arbori de căutare.

A doua proprietate constă în posibilitatea implementării unui algoritm de căutare, asemănător căutării binare în vectorii ordonaţi. Elementul căutat se compară cu rădăcina arborelui. Dacă este egal, atunci algoritmul se încheie; dacă este mai mic, atunci se va căuta în subarborele stâng, iar dacă este mai mare, în cel drept.

Se consideră o funcţie de comparaţie cu prototipul:

```
int cmp_data (DATA a, DATA b);
```

care întoarce o valoare negativă, zero sau pozitivă, după cum **a** este mai mic, egal sau mai mare decât **b**.

Funcţia de căutare, în **forma recursivă**, este:

```
BTREE cauta(BTREE t, DATA x)
{
        int c;
        if  (t= =NULL||(c=cmp_data(x, t->d)= =0)
                return t;
        t=(c<0)?t->left:t->right;
        return cauta(t,x);
}
```

Funcţia primeşte un pointer **t** la rădăcina arborelui şi un element **x**, returnând pointerul la nodul găsit, sau NULL dacă **x** nu este în **t**.

Forma iterativă echivalentă este:

```
BTREE cauta(BTREE t, DATA x)
{
        int c;
        while (t!=NULL&&(c=cmp_data(x, t->d)!=0)
                        t=(c<0)?t->left:t->right;
        return t;
}
```

Dacă arborele şi toţi subarborii săi sunt **echilibraţi**, numărul de comparaţii este de ordinul **log(n)**, unde **n** este numărul de noduri. Dacă arborele este neeechilibrat, atunci căutarea se reduce la o căutare asemănătoare celei dintr-o listă înlănţuită, în care numărul de comparaţii este de ordinul **n**.

Exemplu: să se construiască **arborele de căutare/ inserare** cu următoarele elemente: 14, 7, 19, 4, 18, 3, 6, 8, 17, 21, 20, 10, 5, 1, 9, 16, 15, 11, 13, 2, 12.

Se obţine arborele de căutare/ inserare din figura 16.10.

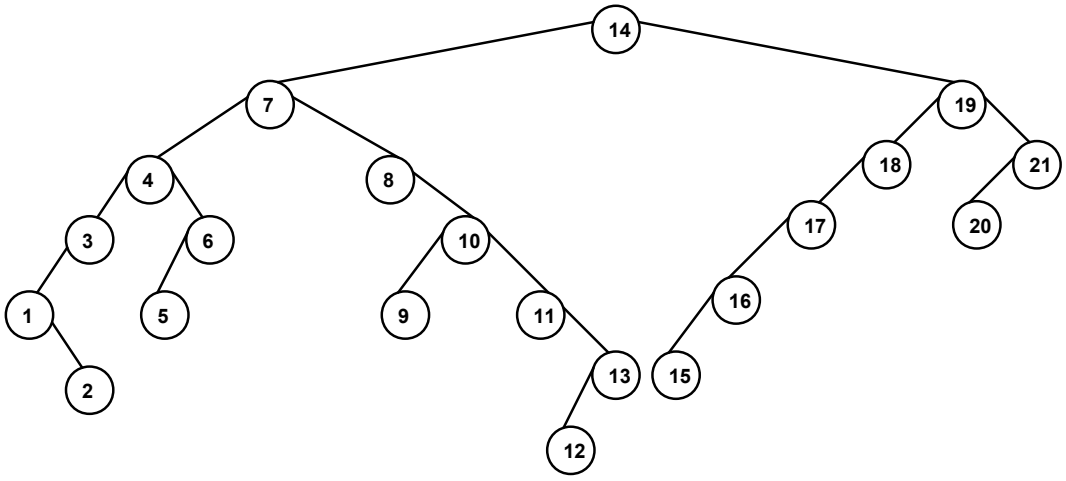


Fig. 16.10. Exemplu de arbore de căutare/ inserare

Inserarea unui element într-un arbore de căutare

Pentru a elimina restricţia de noduri cu elemente distincte, se modifică tipul DATA, introducând pe lângă valoarea propriu-zisă, un întreg (**count**) care precizează de câte ori apare valoarea respectivă. Se va defini tipul informaţiei din nod ca fiind KEY, iar tipul DATA va fi:

```
typedef struct {KEY key; int count;} DATA;
```

Definiţia tipurilor NODE şi BTREE nu se schimbă. Funcţia **new_node()** se va modifica în mod corespunzător, primind o valoare de tip KEY şi iniţializând cu 1 câmpul **count** al nodului creat.

Algoritmul de bază cu care se crează arbori de căutare este cel numit **de căutare şi inserare**. Se dă arborele **t** şi un element **x**. Dacă **x** este în **t**, atunci se incrementează contorul corespunzător; dacă nu, **x** este inserat în **t**, într-un nod terminal, în aşa fel încât arborele să rămână arbore de căutare. Presupunem o funcţie **cmp_key()** care compară două elemente de tip **KEY**, întorcând o valoare negativă, 0 sau pozitivă:

```
int cmp_key (KEY a, KEY b);
```

Implementarea **recursivă** a acestui algoritm este următoarea:

```
BTREE search_and_insert (BTREE t, KEY x)
{
        int c;
        if (t= =NULL)
                return new_node(x);
        if (c=cmp_key(x, (t->d).key))<0
                t->left=search_and_insert(t->left,x);
        else
                if (c>0)
                        t->right=search_and_insert(t->right,x);
                else
                        (t->d).count++;
        return t;
}
```

Funcţia primeşte pointer-ul la rădăcina arborelui **t** şi cheia **x**. Dacă s-a ajuns la un nod terminal, rezultă că **x** nu este în arbore şi se crează un nou nod, iniţializat cu **x**. Dacă nu, se aplică algoritmul de căutare, apelând recursiv funcţia cu subarborele stâng, respectiv drept. Dacă se identifică **x** în arbore, atunci se incrementează câmpul **count** corespunzător. Se întoarce pointerul la arborele modificat.

Implementarea **iterativă** este următoarea:

```
BTREE search_and_insert(BTREE t, KEY x);
{
        int c, gata=0;
        BTREE p=t;
        if (p= =NULL)
                        return new_node(x);
        while (!gata) {
                        if (c=cmp_key(x, (p->d).key))= =0) {
                                (p->d).count++;
                                gata=1;
        }
                        else if (c<0)
                                if (p->left= =NULL) {
                                        p->left=new_node(x);
                                        gata=1;
                                }
                                        else
                                        p=p->left;
```

```
                                else if(p->right= =NULL) {
                                        p->right=new_node(x);
                                        gata=1;
                                }
                                        else
                                        p=p->right;
        }
        return t;
}
```

Dacă arborele nu este vid, se parcurg nodurile într-o buclă **while**, selectând la fiecare nod, descendentul stâng sau drept, în funcţie de rezultatul comparaţiei valorii **x** cu cheia nodului curent. Bucla se termină fie când s-a găsit un nod cu cheie identică cu cea dată, caz în care se incrementează contorul, fie când se ajunge la un nod fără descendent pe ramura pe care trebuie căutat elementul, caz în care se crează un nou nod şi se ataşează nodului curent.

f) Ştergerea unui element dintr-un arbore de căutare

Se consideră în continuare, problema ştergerii unui element dintr-un arbore de căutare, implementând o funcţie care primeşte adresa rădăcinii şi o cheie dată şi care realizează ştergerea cheii respective din arbore, întorcând adresa rădăcinii arborelui modificat. Implementarea este realizată de funcţia **del_search_tree()**.

Dacă cheia apare de mai multe ori, se decrementează contorul respectiv, deci modificarea structurii arborelui este necesară doar în cazul în care cheia apare o singură dată.

Prima operaţie constă în găsirea nodului respectiv în arbore şi este realizată printr-un algoritm similar cu cel de la funcţia de căutare: se compară cheia dată cu cheia rădăcinii, apelându-se recursiv funcţia de ştergere pentru subarborele stâng sau drept. Dacă s-a găsit un nod care conţine cheia dată, iar contorul corespunzător este mai mare ca 1, se decrementează acest contor. Dacă valoarea contorului este 1, apar două subcazuri: nodul are cel mult un descendent şi, respectiv, nodul are doi descendenţi.

Cazul în care există un singur descendent se rezolvă imediat: se copiază în variabila **t** (care conţine adresa nodului care trebuie şters) adresa descendentului, după care se eliberează spaţiul alocat pentru nod, folosind o variabilă intermediară **v**. Dacă lipsesc ambii descendenţi, atunci în **t** se va copia NULL. Funcţia va întoarce variabila **t**.

În cazul în care nodul care trebuie şters are doi descendenţi, apare problema restructurării arborelui de căutare. În câmpul de date al nodului care trebuie şters se copiază o valoare adecvată (care să păstreze caracterul de arbore de căutare), dintr-un nod terminal, după care se va şterge acel nod terminal. Există două variante de alegere a nodului terminal:

- cel mai mare element mai mic decât nodul care urmează a fi şters;
- cel mai mic element mai mare decât nodul care urmează a fi şters.

Aceste două variante se reduc la alegerea:

- celui mai din dreapta nod al subarborelui stâng al nodului care urmează a fi şters;
- celui mai din stânga nod al subarborelui drept al nodului care urmează a fi şters.

În implementarea prezentată în continuare, s-a ales prima variantă, în care se apelează funcţia auxiliară **del_search_node()**. Parametri săi sunt: adresa descendentului stâng şi adresa nodului curent, întorcând adresa descendentului stâng (care a fost modificat). Pentru a se identifica cel mai din dreapta nod, se apelează recursiv funcţia **del_search_node()**. Se remarcă utilizarea variabilei **q** ca variabilă temporară pentru eliberarea spaţiului de memorie ocupat de nodul terminal care se şterge fizic.

Implementările celor două funcţii sunt următoarele:

```
BTREE del_search_node(BTREE r, BTREE q)
        {
                if (r->right!=NULL)
                        r->right=del_search_node(r->right, q);
                else if (q->d).count= =1 {
                        q->d=r->d;
                        q=r;
                        r=r->left;
                        free(q);
                }
                else
                        (q->d).count--;
                return r;
        }

BTREE del_search_tree(BTREE t, KEY x)
        {
                int diff; BTREE v;
```

```
        if (t= =NULL)
                return NULL;
        if (diff=cmp_key(x, (t->d).key))<0
                t->left=del_search_tree(t->left, x);
        else if (diff>0)
                t->right=del_search_tree(t->right, x);
                else
                if ((t->d).count= =1) {
                        v=t;
                        if (v->right= =NULL) {
                                t=v->left;
                                free(v);
                                }
                                else
                v->left=del_search_node(v->left, v);
        }
        else
                (t->d).count--;
return t;
}
```

Testarea funcţiei de ştergere se poate realiza cu următorul program principal, în care s-a presupus tipul **KEY** ca fiind **char ***. Se citesc şiruri de caractere de la consolă (până când se introduce şirul vid) şi se generează arborele de căutare respectiv. Apoi se citesc din nou şiruri de caractere de la consolă şi se şterg din arbore, afişându-se arborele după fiecare operaţie.

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

typedef char *KEY;
typedef struct {KEY key; int count; } DATA;

struct node {
        DATA d;
        struct node *left;
        struct node *right;);
typedef struct node NODE, *BTREE;
BTREE new_node(KEY);
int cmp_key(KEY, KEY);
```

```
void print_key(KEY);
void print_node (DATA);
void print_tree_1(BTREE t, int n);
void print_tree(BTREE t);
void err_exit(char s);
BTREE del_search_tree(BTREE t, KEY x);
BTREE del_search_node(BTREE r, BTREE q);
BTREE search_and_insert(BTREE, KEY);

void main (void)
{
        static char s[80];
        BTREE t=NULL;
        printf("Generare:\n\n");
        while (strlen(gets(s))>0) {
                t=search_and_insert(t, s);
        }
        printf("Arborele de căutare este:\n\n);
        print_tree(t);
        while (strlen(gets(s))>0) {
                t=del_search_tree(t, s);
                printf ("Arborele de căutare este:\n\n");
                print_tree(t);
                if (t= =NULL)
                        break;
        }       }
```

Observaţie: un tablou declarat în interiorul unei funcţii cu specificatorul **static**, va fi alocat la adrese fixe de memorie, dar va fi accesibil doar în funcţia respectivă.

Funcţiile de prelucrare utilizate sunt următoarele (actualizate conform definiţiei câmpului KEY):

```
BTREE new_node(KEY x)
        {
                BTREE t=(BTREE) malloc(sizeof(NODE));
                if (t= =NULL||(t->d).key=strdup(x))= =NULL
                        err_exit("new_node: Eroare alocare");
                (t->d).count=1; t->right=t->left=NULL;
                return t;
        }
```

Observaţie: funcţia standard **strdup()**, cu prototipul:
char ***strdup**(const char *s);

"salvează" şirul indicat de **s** într-o zonă de memorie disponibilă şi întoarce un pointer către zona respectivă sau NULL dacă salvarea nu s-a putut face corect.

```
int cmp_key(KEY a, KEY b)
        { return (strcmp(a, b)); }

void print_node(DATA d)
        { print_key(d.key); printf(" :%2d\n", d.count); }

void print_key(KEY x)
        { printf("%s", x); }

void err_exit(char s);
        { printf("%s\n", s); exit(1) }
```

Rezultatele procedurii de ştergere sunt ilustrate pentru arborele din figura 16.11, arbore din care se elimină succesiv nodurile: 13, 15, 5.

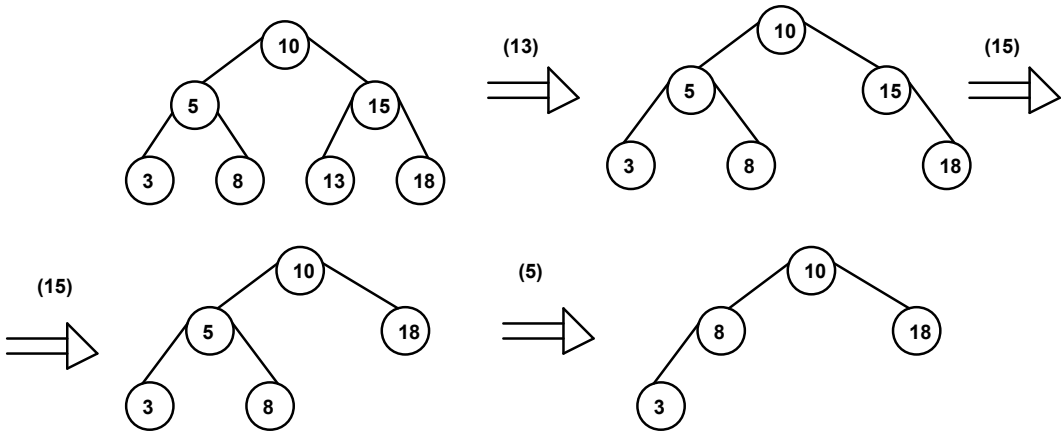


Fig. 16.11. Exemplu de aplicare a procedurii de ştergere pentru un arbore binar

g) Analiza procedurii de căutare/ inserare

Se notează cu **h**, numărul mediu de comparaţii necesare căutării/ inserării unui nou nod într-un arbore binar şi cu **h'** numărul mediu de comparaţii necesare căutării/ inserării unui nou nod într-un arbore binar de căutare.

Pentru un **arbore echilibrat cu n noduri** există relaţia:

$$\mathbf{h \approx \log_2 n}$$

Pentru **arborele de căutare** construit cu procedura de căutare / inserare prezentată anterior, se demonstrează (pentru n mare) că:

$$\mathbf{h' \approx 2\ln(n) - C}$$

unde **C** este o constantă.

Observaţie:

- h reprezintă şi **lungimea medie a traiectoriei interne**. Raportul lungimilor medii ale traiectoriilor **h** şi **h'** (pentru **n** mare), este:

$$\lim_{n->\infty} h'/h = 2\ln(n)/\log_2 n = 2\ln 2 = 1{,}386$$

Deci, pentru arborii echilibraţi se observă o îmbunătăţire a numărului mediu de comparaţii (cu 39%), în raport cu situaţia corespunzătoare unui arbore neechilibrat.

Trebuie remarcat însă faptul că este vorba de numărul **mediu** de comparaţii, ceea ce determină ca **în situaţia cea mai defavorabilă** (**arbore degenerat**, obţinut cu procedura descrisă la punctul d): dacă şirul datelor de intrare este deja ordonat, fiecare nod se inserează imediat la dreapta sau la stânga predecesorului şi ca urmare, arborele se transformă într-o **listă liniară**), numărul comparaţiilor să depăşească această valoare.

Valorile prezentate anterior (raportul egal cu 1,386) au fost deduse în ipoteza datelor de intrare **de probabilităţi egale**.

16.4. Arbori echilibraţi AVL

Deoarece efortul de inserare/ eliminare de noduri dintr-un arbore echilibrat, **cu păstrarea echilibrării**, este relativ mare, s-au adoptat şi alte definiţii (care acoperă un domeniu mai extins) pentru arborii echilibraţi.

Aceste definiţii determină simplificarea procedurilor de inserare/ eliminare de noduri, cu dezavantajul unei creşteri (minime) a timpului mediu de căutare.

O asemenea definiţie corespunde **arborilor echilibraţi AVL (arbori AVL)**, sau **arbori echilibraţi după înălţime**. Denumirea de **AVL** se datorează faptului că definiţia acestor arbori a fost dată de matematicienii **Adelson-Velskii** şi **Landis**.

Definiţie: un arbore este echilibrat **AVL** dacă şi numai dacă pentru orice nod al arborelui, înălţimile subarborilor stâng şi drept diferă cel mult cu o unitate.

Observaţii:

1. Definiţia precedentă determină o valoare a lungimii medii pentru traiectoria internă aproximativ egală cu cea a arborilor echilibraţi.
2. Următoarele operaţii se pot efectua, pentru arborii **AVL,** în $O(\log_2 n)$ unităţi de timp (şi în cazurile cele mai defavorabile):

- găsirea unui nod cu o valoare dată;
- inserarea unui nod cu o valoare dată;
- eliminarea unui nod cu o valoare dată.

Inserarea în arborii AVL

Definiţie: se numeşte **factor de balans** al unui nod **T:**

$$BF(T)=h_L-h_R$$

unde h_L şi h_R sunt înălţimile subarborilor stâng şi drept ai nodului **T.**

Observaţie: pentru un arbore **AVL**, $BF(T)\in\{-1, 0, 1\}$, pentru orice nod **T** al arborelui.

Pentru operarea cu arbori **AVL** se utilizează următoarea definiţie:

```
typedef struct (KEY key; int bf;} DATA;
struct node {
        DATA d;
        struct node *left;
```

```
        struct node *right;}
    typedef struct node NODE, *BTREE;
```

în care **bf** reprezintă factorul de balans al nodului.

Ca urmare a inserării unui nod într-un arbore **AVL**, sunt posibile două situaţii:

1. arborele rezultat este dezechilibrat;
2. arborele rezultat a rămas echilibrat.

1. **Dezechilibrarea arborelui** determină iniţierea procesului de reechilibrare, care se realizează printr-o rotaţie a nodurilor. Există patru situaţii distincte în care inserarea unui nod determină dezechilibrarea arborelui:

a) nodul **Y** se inserează în subarborele **stâng** al subarborelui **stâng** al nodului **A** = **dezechilibrare de tip LL**;
b) nodul **Y** se inserează în subarborele **drept** al subarborelui **stâng** al nodului **A** = **dezechilibrare de tip LR**;
c) nodul **Y** se inserează în subarborele **stâng** al subarborelui **drept** al nodului **A** = **dezechilibrare de tip RL**;
d) nodul **Y** se inserează în subarborele **drept** al subarborelui **drept** al nodului **A** = **dezechilibrare de tip RR**,

unde nodul **A** reprezintă cel mai apropiat predecesor al nodului **Y**, având **BF(A)=±2**.

Operaţiile necesare reechilibrărilor **RR** şi **RL** sunt simetrice cu cele corespunzătoare reechilibrărilor de tip **LL** şi respectiv **LR**. Ca urmare, se vor descrie numai operaţiile de reechilibrare **LR** şi **LL**. În acest scop, se consideră structurile de arbori echilibraţi din figurile 16.12, 16.13, 16.14, 16.15 (se poate verifica faptul că acestea reprezintă singurele cazuri de dezechilibrare a subarborelui stâng).

a) Dezechilibrarea de tip LL

Se notează cu **t** pointerul spre rădăcina subarborelui considerat, cu **a** pointerul către nodul rădăcină (definit ca nodul cel mai apropiat de **Y**, având **BF(A)=±2**), iar cu **b** pointerul către descendentul său stâng: b=a->left;

Se constată că este necesară reechilibrarea de tip **LL** dacă:

```
((a->d).bf=2)) && ((b->d).bf=1))
```

Secvenţa de program care realizează reechilibrarea de tip **LL** este următoarea:

```
a->left=b->right;
b->right=a;
(a->d).bf=0;
(b->d).bf=0;
t=b; /*reactualizeaza radacina*/
```

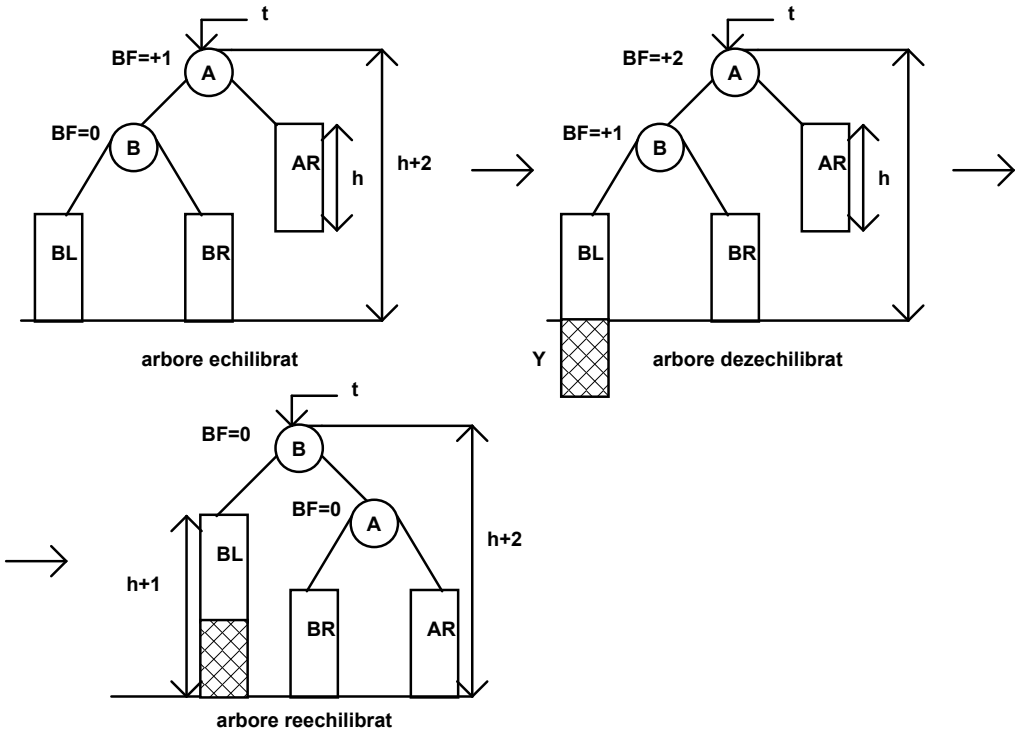


Fig. 16.12. Dezechilibrare de tip LL

b) Dezechilibrarea de tip LR

Se disting trei cazuri:

b1) Dezechilibrare de tip LRa

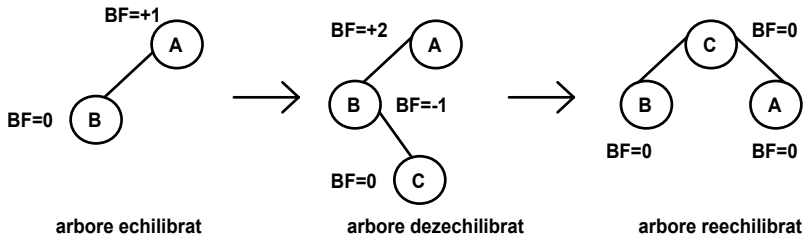


Fig. 16.13. Dezechilibrare de tip LRa

Reechilibrarea descrisă în figura 16.13 este de tipul **LRa** şi este necesară dacă:

((a->d).bf=2)) && ((b->d).bf=1)) && ((c->d).bf=0),

unde: c=b->right.

Se observă că în cazul reechilibrării **LRa** are loc o deplasare în sens trigonometric a nodurilor **A, B, C**.

b2) Dezechilibrare de tip LRb

Reechilibrarea prezentată în figura 16.14 este de tip **LRb** şi este necesară în cazul în care:

((a->d).bf=2)) && ((b->d).bf=1)) && ((c->d).bf=1))

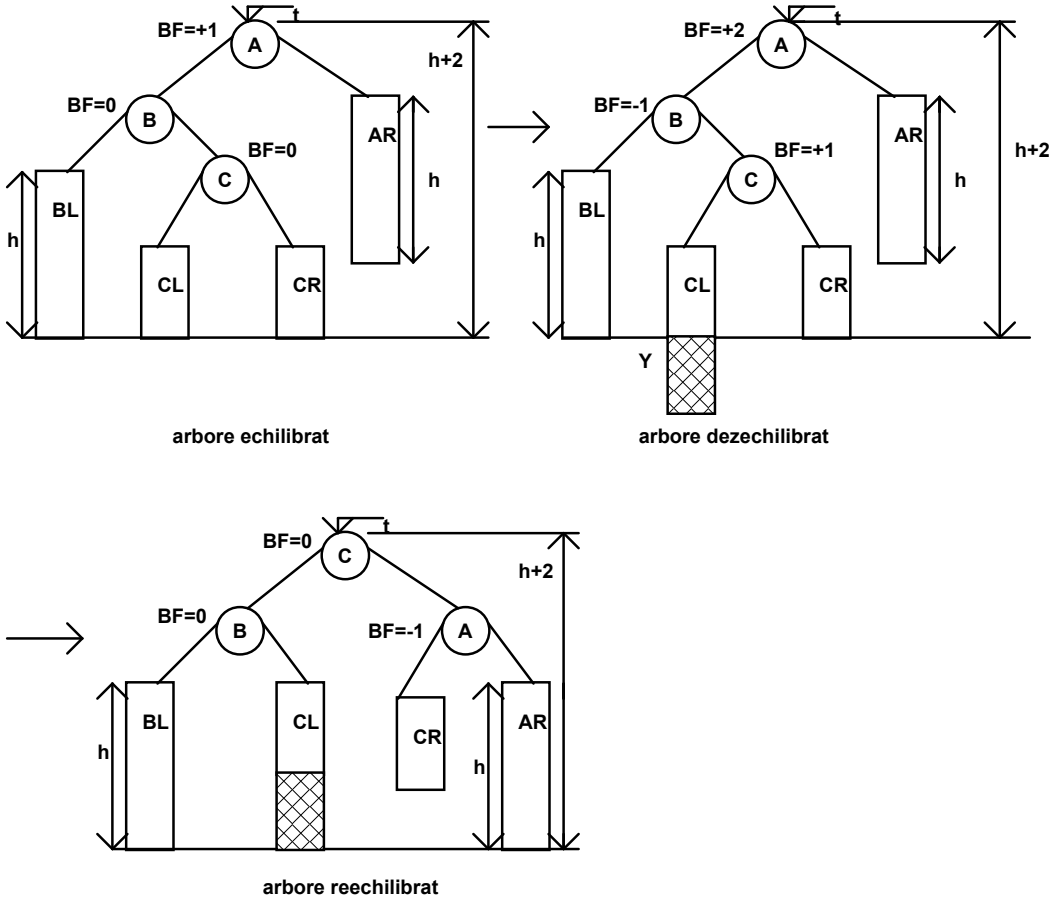


Fig. 16.14. Dezechilibrare de tip LRb

b3) Dezechilibrare de tip LRc

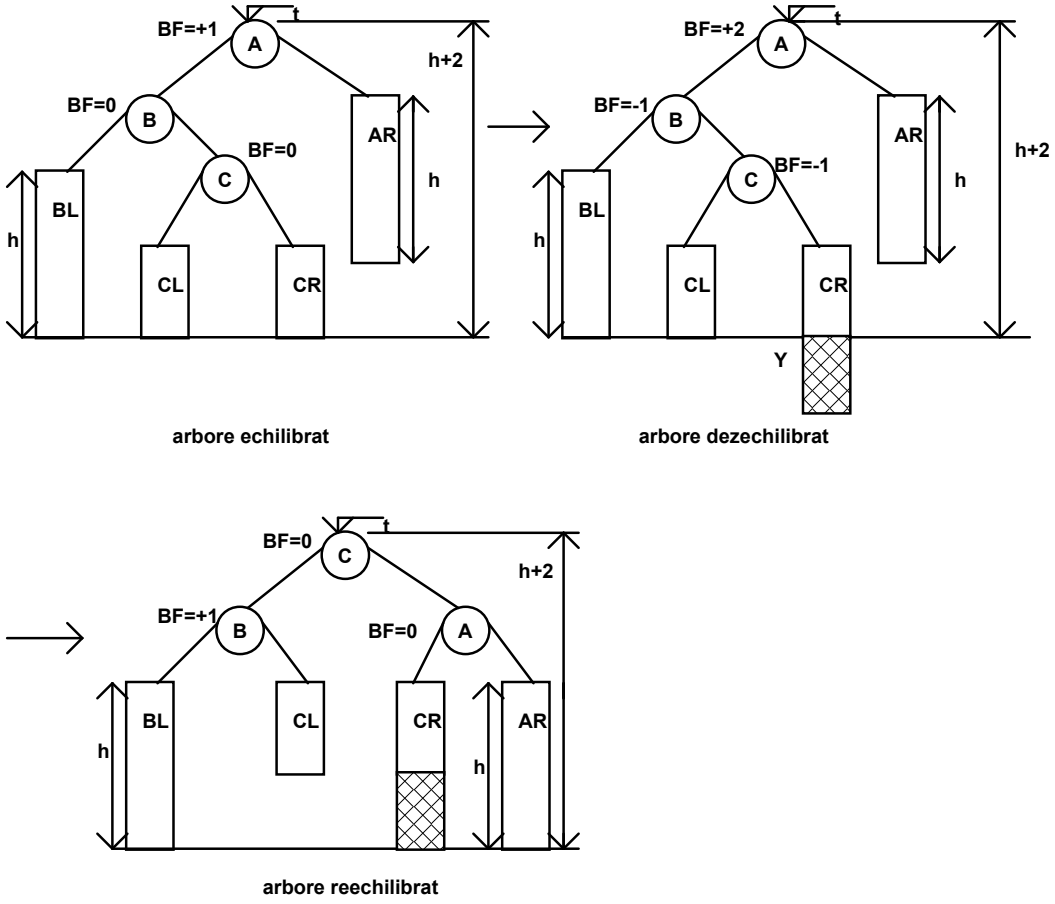


Fig. 16.15. Dezechilibrare de tip LRc

Reechilibrarea de tip **LRc** (figura 16.15) este necesară dacă:

```
((a->d).bf=2)) &&  ((b->d).bf=1)) &&  ((c->d).bf=-1))
```

Ca urmare, prelucrările necesare reechilibrării de tip **LR** sunt următoarele:

```
a->left=c->right;
```

```
b->right=c->left;
c->right=a;
c->left=b;
switch (c->d).bf
        {
        case 0:
                /*LRa*/
                (a->d).bf=0; (b->d).bf=0;
                break;
        case 1:
                /*LRb*/
                (a->d).bf=1; (b->d).bf=0;
                break;
                case -1:
                /*LRc*/
                (a->d).bf=0; (b->d).bf=1;
        }
(c->d).bf=0; t=c;
```

Observaţii:

1. Dezechilibrările şi reechilibrările **RR** şi **RL** sunt simetrice cu **LL** şi **LR** şi se tratează în mod similar.

2. Se constată că după reechilibrarea unui arbore dezechilibrat ca urmare a introducerii unui nod, înălţimea subarborelui obţinut (cu rădăcina **A**) este identică cu cea dinaintea introducerii nodului respectiv. Rezultă că inserarea/ reechilibrarea nu afectează factorii de balans ai nodurilor care nu aparţin subarborelui de rădăcină **A**.

3. Nodul **A**, în jurul căruia se desfăşoară rotaţia de reechilibrare, este cel mai apropiat nod (faţă de **Y**) având **BF(A)=±2**. Rezultă că înainte de inserare, **A** reprezenta cel mai apropiat nod (faţă de **Y**) având **BF(A)=±1**. Această constatare constituie baza metodei de determinare a nodului **A**. Înaintea operaţiei de inserare, orice nod **X** aflat pe traiectoria dintre nodurile **A** şi **Y**, avea **BF(X)=0**. Aceste observaţii permit recalcularea factorilor de balans pentru nodurile subarborelui de rădăcină **A**, după inserarea noului nod.

4. În cazul în care inserarea noului nod **nu determină dezechilibrarea arborelui**, nu se mai desfăşoară nici un proces de reechilibrare. Singura operaţie care trebuie executată este recalcularea factorilor de balans pentru nodurile arborelui respectiv. Fie **A** cel mai

apropiat predecesor al nodului inserat, având **BF(A)=±1** (în cazul în care nu există nici un nod cu **BF(A)=±1**, **A** reprezintă rădăcina arborelui). După inserarea noului nod, rezultă **BF(A)=0**, iar factorii de balans ai nodurilor de pe traiectoria dintre **A** şi **Y** îşi modifică valorile, de la **0**, la **±1**. Modalitatea de determinare a nodului **A**, precum şi cea de calcul pentru factorii de balans ai nodurilor dintre **A** şi **Y,** este identică cu cea prezentată la cazul 1.

Ca urmare a acestor observaţii şi a modului de implementare a reechilibrărilor **LL, LRa, LRb, LRc** (precum şi a condiţiilor care impun efectuarea acestor reechilibrări), rezultă următorul **algoritm de inserare în arborii AVL**.

```
void avlinsert (BTREE root, y, int nk)

/*
root - pointer catre radacina arborelui;
y - pointer catre nodul care se insereaza;
nk - valoarea din câmpul de date al nodului y
*/

BTREE t,a,b,c,f;
int d;
/*
a, b, c - pointeri catre nodurile A, B, C;
f - pointer catre "parintele" nodului A;
d - indicator al sensului dezechilibrului generat (stâng/drept);
t - pointer catre radacina subarborelui care trebuie reechilibrat
*/
{
 /*
determinarea pozitiei de inserare a nodului Y si a pozitiei
nodurilor   referite de pointerii a si f
*/
if (t!=NULL) {
        (insereaza nodul Y ca descendent al nodului curent)
        if (nk>(a->d).key)  d=1;
        /*inserare în subarborele drept al nodului A*/
        else d=-1; /*inserare în subarborele stâng al nodului A*/
        (recalculeaza BF pentru nodurile dintre A si Y)
        if (arbore dezechilibrat) {
                if (d= = -1) { /*dezechilibru stâng*/
```

```
                    b=a->left;
                    if ((b->d).bf= =1)  (reechilibrare de tip LL);
                        else{
                                                c=b->right;
                (reechilibrare de tip LRa, sau LRb, sau  LRc)
                        }
                    }
                    else{
                (prelucrari similare pentru dezechilibru drept)
                    }
        if (f= =NULL)
                    root=t;
                    else if (f->left= =a)  f->left=t;
                            else f->right=t;
}
}
```

16.5. Reprezentarea arborilor prin intermediul arborilor binari

În paragraful 16.1. a fost descrisă metoda de reprezentare a arborilor de ordin **k** (**k≥3**), prin intermediul listelor generalizate. O altă metodă de reprezentare a arborilor utilizează noduri având structura din figura 16.16.

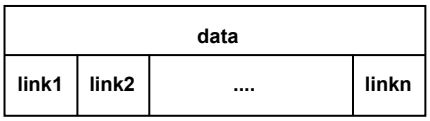


Fig. 16.16. Structura nodurilor unui arbore de ordin n

Această metodă de reprezentare prezintă dezavantajul existenţei a **m[(n-1)+1]** legături libere (NULL), pentru un arbore cu **m** noduri, ceea ce determină o utilizare ineficientă a spaţiului de memorie.

Datorită simplităţii modului de reprezentare a arborilor binari şi datorită variatelor proceduri existente pentru arborii binari în programare, se adoptă în general **reprezentarea arborilor (de ordin k≥3) prin intermediul arborilor binari**.

Se consideră arborele din figura 16.17.

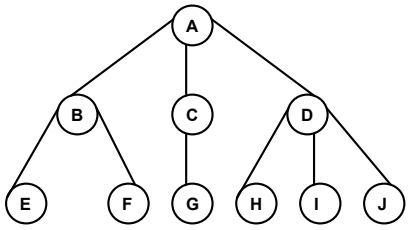


Fig. 16.17. Exemplu de arbore de ordin k=3, care trebuie reprezentat prin intermediul arborilor binari

În scopul utilizării arborilor binari pentru reprezentarea arborilor de un ordin oarecare, trebuie definită o relaţie între noduri care să utilizeze numai două câmpuri de legătură. O asemenea relaţie poate fi, de exemplu:

- descendentul cel mai din stânga;
- primul "frate" din dreapta.

Conform acestei relaţii, structura unui nod este cea ilustrată în figura 16.18.

Fig. 16.18. Structura unui nod al arborelui binar, prin intermediul căruia se reprezintă un arbore de ordin k≥3

unde:

- câmpul **descendent stâng** conţine un pointer spre descendentul cel mai din stânga al nodului respectiv;
- câmpul "**frate" dreapta** conţine un pointer spre primul "frate" din dreapta.

Modul de obţinere a arborelui binar asociat unui arbore de ordin **k≥ 3**, este următorul:

- unirea tuturor descendenţilor unui nod ;
- eliminarea tuturor legăturilor unui nod cu descendenţii săi, cu excepţia nodului din poziţia extremă-stânga (figura 16.19a);
- rotaţia figurii obţinute cu 45^o în sens orar (figura 16.19b).

Rezultă, pentru arborele de ordin **k=3** din figura 16.17, structura prezentată în figura 16.19b.

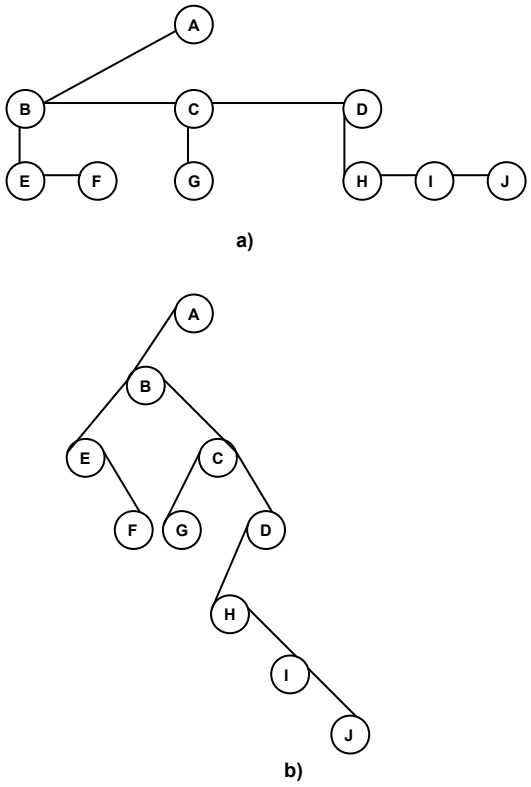


Fig. 16.20. Etapele reprezentării arborelui binar asociat arborelui de ordin k=3, din figura 16.17

Se observă că rădăcina arborelui binar echivalent are un singur descendent (stâng), deci câmpul **"frate" dreapta** are valoarea NULL. Aceasta se datorează faptului că nodul rădăcină al arborelui iniţial nu are "fraţi".

17
Grafuri

O structură de date frecvent utilizată în aplicaţii este reprezentată de **grafuri**.

În electronică, grafurile sunt utilizate în studiul circuitelor electrice (grafuri de semnal, topologia circuitelor), precum şi în rezolvarea problemelor referitoare la reţelele de comunicaţii de date între calculatoare.

17.1. Definiţii

Un **graf G** reprezintă reuniunea a două mulţimi: **N** şi **E (G=(N,E)).**

N reprezintă **o mulţime nevidă de noduri**, iar **E** este **o mulţime de perechi de noduri**, denumite **arce**.

Se numeşte **graf neorientat**, un graf pentru care perechile de noduri care definesc un arc sunt **neordonate**. Astfel, perechile **(x_1, x_2)** şi **(x_2, x_1)** desemnează acelaşi arc.

Se numeşte **graf orientat**, un graf pentru care perechile de noduri care definesc un arc sunt **ordonate**. Pentru arcul **<x_1, x_2>**, x_1 este **vârf iniţial**, iar x_2 este **vârf final**. Perechile **<x_1, x_2>** şi **<x_2, x_1>** reprezintă arce diferite.

În figura 17.1 sunt reprezentate două exemple de grafuri.

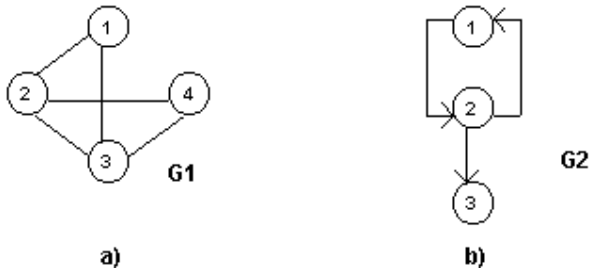


Fig. 17.1. Exemple de grafuri neorientate şi orientate
a) G1=graf neorientat;b) G2=graf orientat

Pentru cele două grafuri ilustrate în figura 17.1, mulţimile de noduri şi arce sunt:

a) **N(G1)={1, 2, 3, 4}**
E(G1)={(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)}

şi respectiv:

b) **N(G2)={1, 2, 3}**
E(G2)={<1, 2>, <2, 1>, <2, 3>}

Pentru un **graf neorientat având n noduri**, numărul maxim de arce este:
n(n-1)/2, iar pentru un **graf orientat**: **n(n-1)**.

Se numeşte **graf complet**, un graf cu **n noduri** şi **n(n-1)/2 arce** (pentru **grafurile neorientate**) sau cu **n(n-1) arce** (pentru **grafurile orientate**).

Nodurile adiacente se definesc în modul următor:

- dacă arcul **(x_1, x_2)** aparţine **grafului neorientat G ((x_1, x_2) $\in$E(G))**, atunci **nodurile x_1, x_2** sunt denumite **adiacente**, iar **arcul (x_1, x_2)** este **incident în nodurile x_1, x_2;**

- dacă arcul **<x_1, x_2>** aparţine **grafului orientat G (<x_1, x_2> $\in$E(G))**, atunci **x_1** este denumit **adiacent către x_2**, iar **x_2**, **adiacent de la x_1**.

Se numeşte **subgraf al lui G**, un graf **G'** pentru care sunt definite relaţiile:

N(G')$\subset$N(G);
E(G')$\subset$E(G).

Se defineşte **drumul între nodurile x_n şi x_m $\in$N(G) (**G este **graf neorientat**), ca o secvenţă de arce: **(x_n, x_{k1}), (x_{k1}, x_{k2}), ... ,(x_{ki},**

$\mathbf{x_m) \in E(G)}$. Dacă graful **G este orientat**, atunci **drumul între nodurile** $\mathbf{x_n}$ **şi** $\mathbf{x_m}$ este reprezentat de secvenţa de arce:

$$\mathbf{<x_n, x_{k1}>, <x_{k1}, x_{k2}>, ... ,<x_{ki}, x_m> \in E(G)}.$$

Se defineşte **lungimea unui drum**, ca numărul de arce din care este constituit drumul respectiv.

Un **drum simplu** reprezintă un drum pentru care toate nodurile din care alcătuit (cu excepţia posibilă a primului sau a ultimului nod), sunt distincte.

Bucla reprezintă un drum simplu pentru care nodul iniţial este identic cu nodul final.

Dacă un graf **G este neorientat**, două noduri $\mathbf{x_1, x_2 \in N(G)}$ sunt **conexe**, dacă există un drum de la $\mathbf{x_1}$ la $\mathbf{x_2}$ (în acest caz, există drum şi de la $\mathbf{x_2}$ la $\mathbf{x_1}$).

Dacă graful **G este neorientat**, atunci **G** se numeşte **conex**, dacă $\forall$ $\mathbf{x_i, x_j \in N(G), i \# j}$, există un drum simplu între nodurile $\mathbf{x_i}$ şi $\mathbf{x_j}$. De exemplu, în figura 17.2, **G1 este graf conex**, iar **G2 este un graf neconex.**

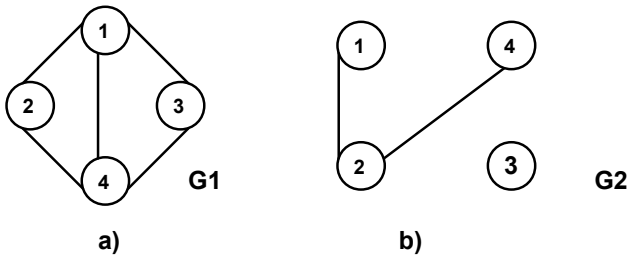


Fig. 17.2. Exemple de grafuri conexe şi neconexe

a) G1=graf conex b) G2=graf neconex

Componenta conexă a unui **graf neorientat** se defineşte ca **subgraful conex maximal** al grafului respectiv.

Un **graf orientat G** se numeşte **strâns conex**, dacă pentru $\forall \mathbf{x_i, x_j \in N(G)}$, există un drum orientat de la $\mathbf{x_i}$ la $\mathbf{x_j}$ şi un drum orientat de la $\mathbf{x_j}$ la $\mathbf{x_i}$.

Gradul unui nod reprezintă numărul de arce incidente în nod (pentru **grafuri neorientate**).

Pentru **grafuri orientate** se definesc:

- **gradul la ieşire** al unui nod: numărul de arce care **pleacă** din nod (**arce divergente**);
- **gradul la intrare** al unui nod: numărul de arce care **converg** în nod (**arce convergente**).

17.2. Reprezentarea grafurilor

Există trei moduri de reprezentare a grafurilor, în funcţie de aplicaţia dată:

a) cu matrici de adiacenţă;
b) cu liste de adiacenţă;
c) cu multiliste de adiacenţă.

a) Reprezentarea grafurilor cu matrici de adiacenţă

Fie **G∈(N, E)** un **graf neorientat cu n noduri, n≥1**.
Se numeşte **matrice de adiacenţă A,** o matrice **nxn**, cu proprietăţile:

- **A[i, j]=1**, dacă şi numai dacă arcul **$(x_i, x_j) \in E(G)$**;
- **A[i, j]]=0**, dacă şi numai dacă arcul **$(x_i, x_j) \notin E(G)$.**

Observaţie: pentru un **graf orientat, A[i, j]=1**, dacă şi numai dacă **$<x_i, x_j> \in E(G)$.**

De exemplu, pentru **graful G1 din figura 17.1a**, matricea de adiacenţă este:

$$\mathbf{A}=\begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{matrix}$$

iar pentru **graful G2 din figura 17.1b**, matricea de adiacenţă este:

$$\mathbf{A}=\begin{matrix}0 & 1 & 0\\ 1 & 0 & 1\\ 0 & 0 & 0\end{matrix}$$

Pentru **grafurile neorientate, matricea de adiacenţă este simetrică**, iar pentru **grafurile orientate, este nesimetrică**.

Spaţiul de memorie necesar stocării elementelor matricii de adiacenţă asociate unui graf cu **n noduri**, este de $\mathbf{n^2}$ **biţi**; ca urmare, rezultă posibilitatea realizării unei economii de memorie pentru **grafurile neorientate**, prin stocarea numai a $\mathbf{n^2/2}$ **elemente**, matricea fiind simetrică.

Pentru un **graf neorientat**, se defineşte **gradul unui nod** $\mathbf{k_i}$:

$$\mathbf{gr(k_i)}=\sum_{i=1}^{n}\mathbf{A[i,\ k]}=\sum_{i=1}^{n}\mathbf{A[k,\ i]}$$

Pentru un **graf orientat**, se definesc:

- gradul la ieşire al nodului $\mathbf{k_i}$:

$$\mathbf{gr_{ieş}(k_i)}=\sum_{i=1}^{n}\mathbf{A[k,\ i]}$$

- gradul la intrare al nodului $\mathbf{k_i}$:

$$\mathbf{gr_{int}(k_i)}=\sum_{i=1}^{n}\mathbf{A[i,\ k]}$$

Utilizând reprezentarea grafurilor cu matrici de adiacenţă, operaţiile de determinare a numărului de arce ale grafului şi de testare a proprietăţii de graf conex au timpul de execuţie egal cu $\boldsymbol{O}(\mathbf{n^2})$.

b) Reprezentarea grafurilor cu liste de adiacenţă

În acest caz, cele **n** linii ale matricii de adiacenţă se reprezintă sub forma a **n liste simplu înlănţuite**. Fiecărui nod al grafului îi corespunde o listă simplu înlănţuită. Structura elementelor unei astfel de liste înlănţuite este ilustrată în figura 17.3.

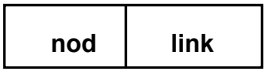


Fig. 17.3. Structura elementelor listelor înlănţuite asociate nodurilor unui graf

Declaraţiile tipurilor şi variabilelor necesare reprezentării grafurilor prin liste de adiacenţă, sunt următoarele:

```
struct gnode {
        int nod;
        struct gnode *link;}
typedef struct gnode NODE, *GRF;

GRF prim[n];
```

Utilizând aceste definiţii, grafurile **G1** şi **G2** din figura 17.1 sunt reprezentate prin intermediul listelor de adiacenţă ca în figura 17.4.

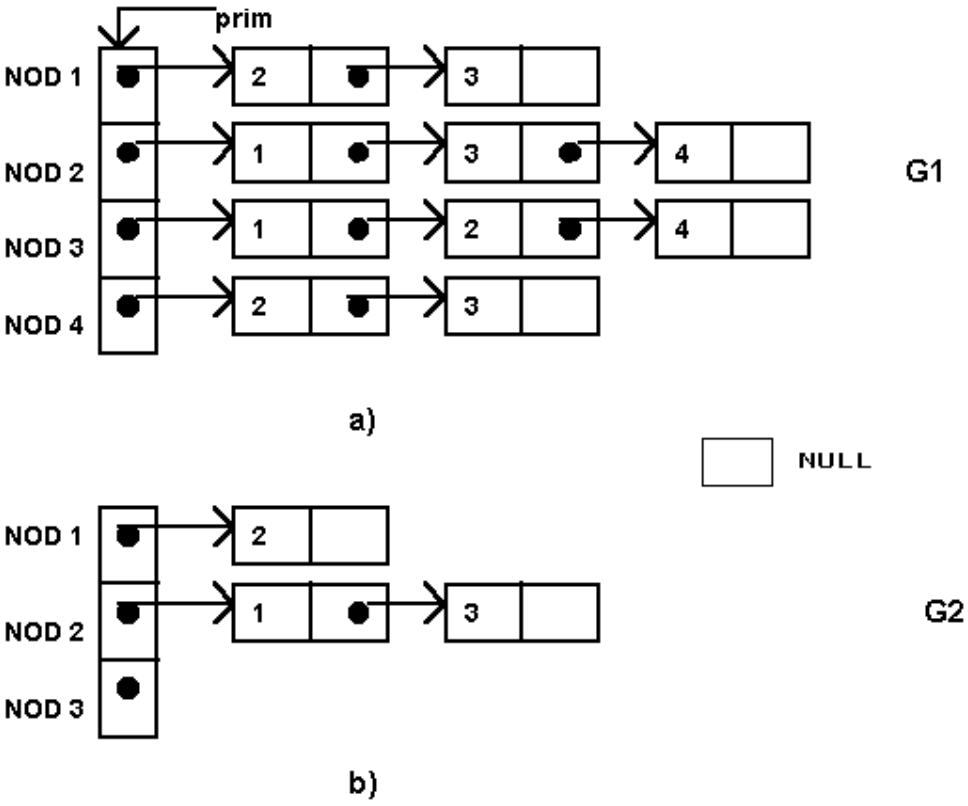


Fig. 17.4. Exemple de reprezentare a grafurilor cu liste de adiacenţă
a) G1=graf neorientat
b) G2=graf orientat

Observaţie: ordinea nodurilor din listele de adiacenţă nu este importantă.

Dacă **G este un graf neorientat** având **n noduri** şi **e arce**, reprezentarea prin liste de adiacenţă necesită **n noduri cap** şi **2e noduri în liste** (respectiv, **e noduri** în liste, pentru **grafurile orientate**).

Gradul unui **nod ∈G (G=graf neorientat)** este egal cu numărul de noduri din listele de adiacenţă. Timpul de execuţie asociat operaţiilor necesare determinării numărului total de arce dintr-un **graf neorientat** este: ***O*(n+e)**.

Pentru un graf **G orientat**, **gradul la ieşire al unui nod ∈G** este egal cu numărul de noduri din listele de adiacenţă.

Pentru determinarea **gradului la intrare al unui nod ∈G (G orientat)**, se utilizează **listele de adaiacenţă inverse** (care conţin nodurile adiacente la **nodurile cap**). De exemplu, pentru graful **G2 din figura 17.1**, listele de adiacenţă inverse sunt prezentate în figura 17.5.

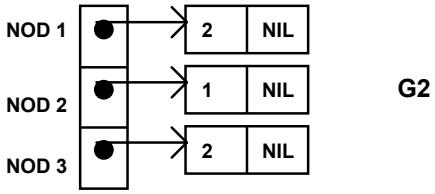


Fig. 17.5. Exemplu de liste de adiacenţă inverse asociate unui graf orientat

c) Reprezentarea grafurilor cu multiliste de adiacenţă

Multilistele de adiacenţă sunt structuri de date care pot avea în comun anumite elemente (un arc al unui graf fiind incident la două noduri, cele două liste de adiacenţă corespunzătoare au un nod comun). Reprezentarea grafurilor prin multiliste de adiacenţă este utilizată în situaţiile în care trebuie determinată şi marcată fiecare parcurgere a arcurilor unui graf. Deoarece este posibilă existenţa aceluiaşi nod în două multiliste de adiacenţă, în structura nodurilor sunt prevăzute câmpuri pentru determinarea succesorului nodului curent în cele două liste specificate.

Structura nodurilor unei multiliste de adiacenţă este prezentată în figura 17.6.

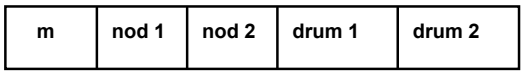


Fig. 17.6. Structura nodurilor unei multiliste de adiacenţă

unde:

- **drum 1** este arcul următor arcului **(nod 1, nod 2)** în **lista 1;**
- **drum 2** este arcul următor arcului **(nod 1, nod 2)** în **lista 2;**
- **m** reprezintă o variabilă booleană, care semnalizează dacă arcul respectiv a mai fost parcurs.

Definiţiile de tip şi declaraţiile de variabile corespunzătoare, sunt următoarele:

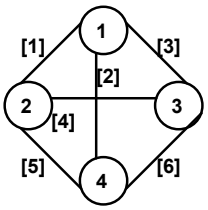
```
struct arc {
        int m;
        int nod1, nod2;
        struct arc *drum1, *drum2;}
typedef struct GNODE, *ARCPTR;

ARCPTR prim[n];
```

Spaţiul necesar stocării în memorie a multilistelor de adiacenţă este diferit de cel corespunzător listelor de adiacenţă, datorită introducerii câmpului suplimentar, **m**.

Exemplu: se consideră graful neorientat din figura 6.7a. Se numerotează arcele grafului şi se scriu, pentru fiecare nod, listele care conţin numerele arcelor incidente nodului respectiv. Se obţin astfel, pentru nodurile grafului din figura 6.7a, următoarele liste:

nod 1: [1], [2], [3];
nod 2: [1], [4], [5];
nod 3: [3], [4], [6];
nod 4: [2], [5], [6].



a)

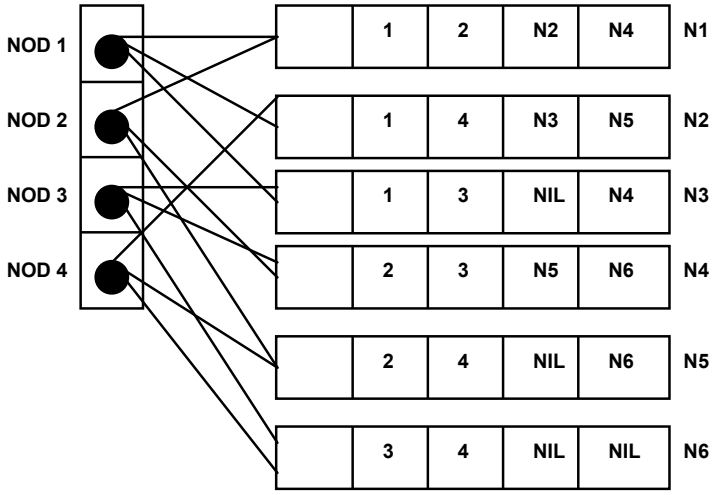


Fig. 17.7. Reprezentarea grafurilor cu multiliste de adiacenţă

a) Graful considerat
b) Multilistele de adiacenţă asociate

Din aceste liste, se pot deduce, pentru un arc oarecare **[j]**, arcele următoare din cele două multiliste de adiacenţă, cărora arcul **[j]** le aparţine.

Au fost utilizate următoarele notaţii:

[1] = arc(1, 2) = N1;
[2] = arc(1, 4) = N2;
[3] = arc(1, 3) = N3;
[4] = arc(2, 3) = N4;
[5] = arc(2, 4) = N5;
[6] = arc(3, 4) = N6.

Rezultă că pentru graful din figura 17.7a, multilistele de adiacenţa corespunzătoare sunt:

nod 1: N1->N2->N3;
nod 2: N1->N4->N5;
nod 3: N3->N4->N6;
nod 4: N2->N5->N6.

17.3. Traversarea grafurilor. Subgrafuri conexe. Arbori de acoperire.

Se consideră **un graf neorientat G, G=(N, E)**. Traversarea grafului **G** constă în parcurgerea tuturor nodurilor conexe (o singură parcurgere pentru fiecare nod), pornind de la un nod **x** $\in$**N(G)**. Există două modalităţi de traversare a grafurilor:

a) traversarea în adâncime = "Depth First Search" (DFS);
b) traversarea în lăţime = "Breadth First Search" (BFS).

a) Traversarea în adâncime

Algoritmul de traversare în adâncime este următorul: se începe parcurgerea grafului **G** cu un nod **x** $\in$**N(G)** (care se marchează ca **vizitat**). Apoi, se selectează un nod **y nevizitat**, adiacent nodului **x** şi se iniţiază un nou proces de traversare în adâncime, începând cu nodul **y**. În momentul în care parcurgerea grafului a ajuns la un nod **z** pentru care s-au **vizitat** toate nodurile adiacente, se revine la ultimul nod **vizitat** care are cel puţin un nod adiacent **nevizitat, u**, care va reprezenta punctul de iniţiere al unui nou proces de traversare în adâncime.

Traversarea în adâncime se încheie în momentul în care nici un nod **nevizitat** nu mai poate fi parcurs pornindu-se de la nodurile **vizitate**.

Pentru descrierea algoritmului, se declară o variabilă globală:

int **visited**[n];

iniţializată în programul principal, cu 0.

Procedura de traversare în adâncime pentru un graf **G=(N, E)**, neorientat, este următoarea:

```
void dfs (int x)
        {
                int y;
                visited[x]=1;
                for (orice nod y adiacent lui x)
                        if (!visited[y] )
                                dfs(y);
        }
```

Începând parcurgerea grafului cu nodul **x**, procedura **dfs** determină toate nodurile conexe nodului **x**.

Dacă graful **G** având **e** arce şi **n** noduri este reprezentat prin **liste de adiacenţă**, atunci durata execuţiei procedurii de traversare în adâncime, este ***O*(e)**; în cazul reprezentării prin **matrici de adiacenţă**, durata de execuţie a acestei proceduri devine $O(n^2)$.

Exemplu: pentru graful din figura 17.8a, reprezentat prin listele de adiacenţă din figura 17.8b, se obţine următoarea secvenţă a nodurilor, ca rezultat al procedurii de traversare în adâncime:

1 -> 2 -> 4 -> 8 -> 5 -> 6 -> 3 -> 7.

b) Traversarea în lăţime

Algoritmul de traversare în lăţime este următorul: se începe parcurgerea grafului **G** cu un nod $x \in N(G)$ şi se **vizitează** toate nodurile **y** adiacente nodului **x**; apoi, se parcurg nodurile adiacente **nevizitate** corespunzătoare nodului **y**, ş.a.m.d.

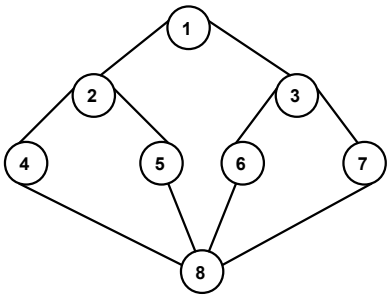
Pentru graful din figura 17.10a, secvenţa nodurilor obţinută prin traversarea în lăţime, este următoarea:

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8.

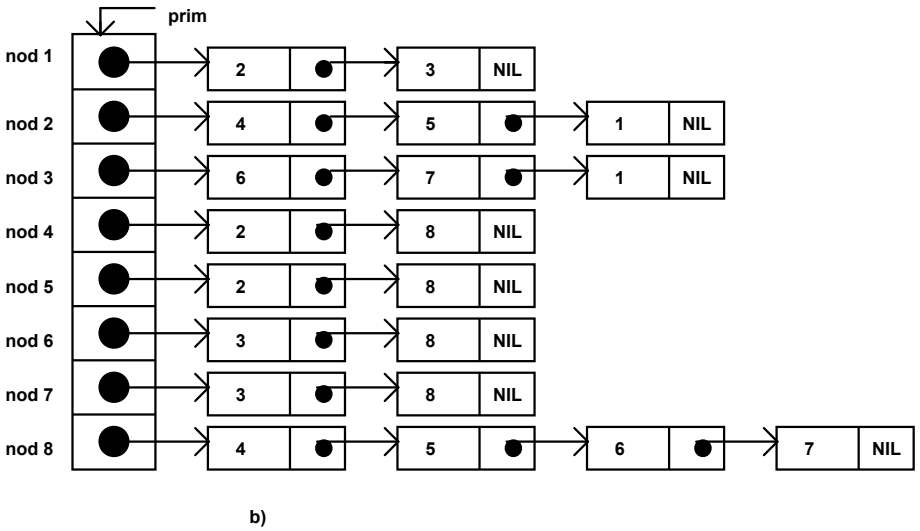
Algoritmul de traversare în lăţime este implementat de procedura **bfs**, care utilizează:

- graful **G** şi tabloul **visited**, ca variabile globale (tabloul **visited** este iniţializat cu valoarea FALSE pentru toate nodurile);

- procedurile **addq, deleteq, initq, emptyq**, care realizează următoarele operaţii asupra cozilor: adăugarea unui element, eliminarea unui element, iniţializarea cozii, test de coadă vidă.



a)



b)

Fig. 17.8. Exemplu de traversare în adâncime a grafurilor
a) Graful considerat b) Listele de adiacenţă asociate

```
void bfs (int x)
        {
                int y;
                queue q;
```

```
                visited[x]=1;
                initq(q); /*initializeaza coada*/
                addq(q, x); /*adauga nodul x în coada*/
                while (!emptyq(q)){
                        deleteq(q, x);
                        for (orice nod y adiacent nodului x) {
                                addq(q, y);
                                visited[y]=1;
                        }
                }
        }
```

c) Subgrafuri conexe

În scopul determinării tuturor subgrafurilor conexe ale unui graf **G**, se apelează procedura **dfs(x)** (sau **bfs(x)**), unde **x** reprezintă un nod **nevizitat** al grafului **G**.

Procedura de determinare a tuturor subgrafurilor conexe ale unui graf G, este următoarea:

```
void sgfconex (graf neorientat g)
        {
                int visited[n];
                int i;
                for (i=1; i=n; i++)
         visited[i]=0; /*initializeazt toate nodurile ca nevizitate*/
                if (!visited[i]){
                        dfs(i); /*determina un subgraf conex*/
                         (tipareste noile noduri determinate,
                         împreuna cu arcele incidente la ele)
                }
        }
```

Timpul de execuţie pentru procedura **sgfconex** este ***O*(e)** în cazul reprezentării grafului **G** (având **n noduri** şi **e arce**) cu **liste de adiacenţă**, respectiv ***O*(n^2)** în cazul reprezentării cu **matrici de adiacenţă**.

d) Arbori de acoperire (arbori liberi)

Fie **G un graf conex, neorientat, având n noduri**.

Definiţie: se numeşte **arbore liber**, un subgraf conex al lui **G**, care conţine toate nodurile grafului, dar nu conţine bucle.

Observaţie: se poate adopta şi următoarea definiţie echivalentă: arborele liber este un subgraf conex al lui **G**, având **n noduri** şi **n-1 arce**.

Arcele arborelui liber se numesc **ramuri** şi constituie elementele mulţimii **R**; arcele subgrafului, diferite de **ramuri**, se numesc **corzi** şi se memorează în mulţimea **C**.

Evident există relaţia **C=E-R**.

Mulţimea **R** se construieşte adăugând pe rând, arcele **vizitate** ca urmare a parcurgerii **dfs** sau **bfs** a subgrafului conex (iniţial, mulţimea **R** este vidă):

R <- R U {(x, y)}, în cazul **notvisited[x]**.

În figura 17.9 sunt prezentate câteva exemple de arbori liberi.

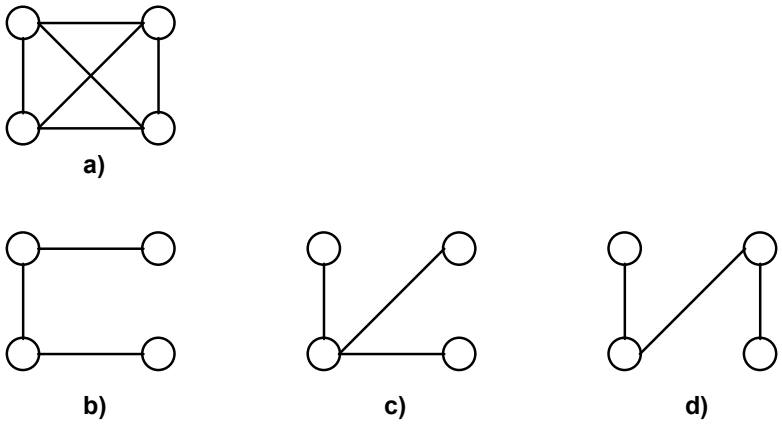


Fig. 17.9. Exemple de arbori liberi
a) Graful iniţial
b), c), d) Arbori liberi asociaţi grafului iniţial

Utilizarea arborilor liberi

a) Obţinerea unui set de ecuaţii liniar independente pentru un circuit electric

În scopul obţinerii unui set de ecuaţii liniar independente pentru un circuit electric, se utilizează următorul algoritm: pentru circuitul electric dat, se determină arborele liber corespunzător şi apoi, se introduc pe rând, în arborele determinat, corzile aparţinând mulţimii **C**. Ca urmare a introducerii unei corzi în subgraf, se crează o buclă, deci se poate scrie o ecuaţie (legea a doua a lui Kirchoff). Procesul descris continuă până în momentul în care mulţimea **C** devine vidă; se obţine un set de ecuaţii liniar independente.

b) Determinarea drumului de cost minim în reţelele de telecomunicaţii

În scopul determinării drumului de cost minim, arcelor grafului li se asociază o transmitanţă (**cost**). În cazul reţelelor de telecomunicaţii, un obiectiv important este interconectarea tuturor nodurilor, astfel încât **costul reţelei respective să fie minim** (se defineşte **costul arborelui liber, ca suma costurilor tuturor ramurilor**). Pentru determinarea **arborelui liber de cost minim**, se utilizează următorul algoritm: se construieşte mulţimea **R**, adăugând pe rând, arcele subgrafului asociat reţelei considerate (arcele sunt ordonate crescător după cost). Un arc se introduce în arborele liber numai în condiţiile în care nu formează o buclă cu arcele existente deja în arbore.

Pentru implementarea acestui algoritm, sunt necesare următoarele funcţii:

- determinarea arcului de cost minim din mulţimea **E(G)**;
- eliminarea unui arc din mulţimea **E(G).**

Iniţial, mulţimea **E(G)** conţine toate arcele grafului. Cele două funcţii specificate anterior se pot executa într-un interval de timp optim, dacă arcele mulţimii **E(G)** sunt memorate într-o listă simplu înlănţuită, ordonată după **cost**.

Algoritmul de determinare a arborelui liber de cost minim este următorul:

```
R=∅; /*initial, arborele nu are nici o ramura*/
while ((card(R)<n-1) && (E nevida)){
                (determina arcul (x, y)∈E(G) de cost minim);
                (sterge (x, y) din E);
        if ((x, y) nu creaza o bucla în R)  (add (x, y) la R);
                        else discard ((x, y)) /*elimina arcul (x, y)*/
}
if (card(R)<n-1) printf ("Nu exista arbore liber");
```

18
Sortarea internă

18.1. Introducere

Definiţie: sortarea reprezintă procesul aranjării unei mulţimi de elemente într-o ordine oarecare.

Fiind date elementele: $\mathbf{a_1, ... ,a_n}$, se impune determinarea unei permutări:
$\mathbf{a_{k1}, ... ,a_{kn}}$, astfel încât fiind dată o funcţie de ordonare, **f**, să existe relaţiile:

$$\mathbf{f(a_{k1}) \le f(a_{k2}) \le ... \le f(a_{kn})}.$$

Scopul sortării este de a facilita căutarea unui element într-o mulţime dată.

Metodele de sortare sunt următoarele:

a) **sortarea vectorilor**; b) **sortarea fişierelor secvenţiale**,

şi au denumirile corespunzătoare: a) **sortare internă** şi respectiv, b) **sortare externă**.

Elementele mulţimii care urmează a fi sortată se pot afla:

- în memoria internă a calculatorului (viteza de acces fiind mare);
- în dispozitivele externe (disc, bandă, etc.) de memorare (viteza de acces fiind redusă).

Definiţie: o metodă de sortare se numeşte **stabilă**, dacă ordinea relativă a elementelor cu chei egale rămâne neschimbată în momentul utilizării unui alt algoritm de sortare.

Stabilitatea sortării este necesară în cazul în care elementele $\mathbf{a_1, ... ,a_n}$ au fost deja sortate după o **cheie secundară** şi urmează a fi sortate după **cheia principală, "key"**.

În continuare, se va utiliza sortarea după o singură cheie.

Evaluarea performanţelor algoritmilor de sortare:

a) **după spaţiul de memorie necesar sortării**: se utilizează numai algoritmi care necesită **n locaţii de memorie** pentru sortarea unei mulţimi de **n** elemente (nu se adoptă algoritmi de sortare care ordonează vectorul iniţial $\mathbf{a_1, ... ,a_n}$ şi depun rezultatul în vectorul final $\mathbf{b_1, ... ,b_n}$);

b) **după timpul de execuţie**: acest timp poate fi evaluat prin determinarea numărului de comparaţii necesare ordonării şi a numărului de permutări de elemente.

18.2. Principiul DIVIDE-ŞI-CUCERE^aTE (DIVIDE-AND-CONQUER)

Fiind date **n** intrări care trebuie prelucrate, tehnica **divide-şi-cucereşte** presupune împărţirea intrărilor în **k** submulţimi, $\mathbf{1 \le k \le n}$, obţinându-se astfel **k "subprobleme"** ale problemei iniţiale. Aceste **k "subprobleme"** trebuie rezolvate, urmând ca ulterior, să se definească o metodă de combinare a rezultatelor într-o soluţie globală a problemei.

În general, **"subproblemele"** sunt de acelaşi tip cu problema iniţială; în aceste situaţii, principiul **divide-şi-cucereşte** se exprimă prin proceduri recursive.

Ca exemplificare, se consideră o aplicaţie a principiului **divide-şi-cucereşte**, în care intrările sunt împărţite în două **"subprobleme"** de acelaşi tip cu problema iniţială. Cele **n** intrări sunt memorate în vectorul **a**, definit prin:

```
item a[n];
int p;
int q;
```

Algoritmul divide-şi-cucereşte este următorul:

```
void d & c (int p, q)
        {
```

```
int m, n;
if SMALL(p, q) PREL(p, q);
        else
                {
                m=DIVIDE(p, q);
                COMBINE(d & c(p, m), d & c(m+1, q));
                }
}
```

În algoritmul prezentat anterior au fost apelate următoarele proceduri şi funcţii:

- funcţia **SMALL(p, q)** - funcţie care întoarce valoarea 1 dacă variabilele **p** şi **q** nu mai permit o divizare în continuare a problemei în **"subprobleme"**;
- funcţia **DIVIDE(p, q)** - funcţie de tip **int**, care întoarce valoarea **m**, indicând modul de grupare a intrărilor (intrările sunt împărţite în două grupuri: până la **m** şi de la **m** în continuare);
- procedura **PREL(p, q)** - realizează prelucrările necesare asupra intrărilor:
a[p], ... ,a[q];
- procedura **COMBINE(d & c(p, m), d & c(m+1, q))** - determină rezultatul prelucrărilor asupra şirului **a[p], ... ,a[q]**, pe baza rezultatelor prelucrărilor asupra intrărilor **a[p], ... ,a[m]** şi respectiv, **a[m+1], ... ,a[q]**.

Dacă se notează timpii de execuţie (pentru **n intrări)** cu:
- **T(n)** - pentru procedura **d and c**;
- **f(n)** - pentru funcţia **DIVIDE** şi procedura **COMBINE**;
- **g(n)** - pentru procedura **PREL**,

se obţine următoarea relaţie de recurenţă:

$$T(n)=\begin{cases} g(n), & \text{pentru n mic} \\ 2T(n/2)+f(n), & \text{în rest} \end{cases}$$

18.3. Sortarea prin inserţie

Metodele de sortare prin inserţie se bazează pe calculul poziţiei finale a unui element al tabloului şi inserarea lui în această poziţie. Inserarea presupune deplasarea unor înregistrări prin copiere succesivă.

Se va sorta şirul stocat în tabloul **numbers[array_size]**;

În această metodă, calculul poziţiei finale a unui element **numbers[i]** se face prin parcurgerea în sens descrescător a elementelor aflate la stânga sa, adică **numbers[i-1], numbers[i-2],** etc., procesul oprindu-se când se ajunge la un element care nu este mai mic decât **numbers[i]**. Simultan, se deplasează la dreapta elementele parcurse, creându-se astfel, spaţiu în tablou pentru poziţia finală a lui **numbers[i]**. Procesul se repetă pentru toate elementele **numbers[i]**.

Elementul curent trebuie copiat iniţial într-o variabilă **index**, şi apoi, la terminarea numărării elementelor succesive mai mici decât **index**, trebuie copiat în poziţia eliberată prin deplasările succesive. Se crează astfel, două cicluri:

- unul care se execută pentru toate elementele tabloului;
- unul care identifică poziţia finală a elementului curent, realizând şi deplasările necesare.

Implementarea este următoarea:

```
void insertionSort(int numbers[], int array_size)
{
  int i, j, index;

  for (i=1; i < array_size; i++)
  {
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {
      numbers[j] = numbers[j-1];
      j = j - 1;
    }
    numbers[j] = index;
  }
}
```

Acest algoritm este de ordinul ***O*(array_size^2)**

În figura 18.1 este ilustrată eficienta algoritmului de sortare prin inserţie (timp de sortare în funcţie de numărul de elemente ale sirului de sortat). Elementele sirului iniţial sînt generate aleator.

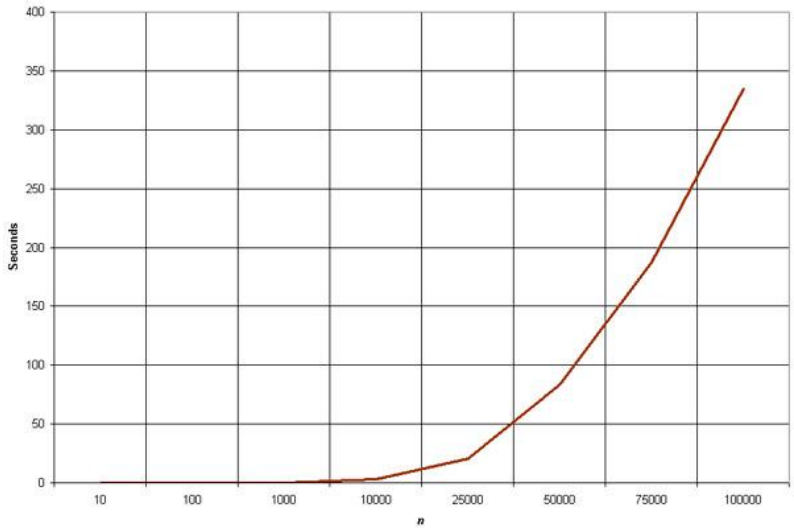


Figura 18.1 Eficienţa algoritmului de sortare prin inserţie.

18.4. Sortarea prin interschimbare (Bubblesort)

Cea mai simplă metodă este cea de **interschimbare directă,** care se mai numeşte şi **Bubblesort** sau **metoda bulelor**.

Prima înregistrare **numbers[0]** se compară cu toate celelalte, realizându-se interschimbări acolo unde este cazul. Se repetă pentru următoarele înregistrări: **numbers[1]**, etc.Programul se va implementa printr-un ciclu inclus în alt ciclu, fiecare cu **array_size-1** paşi, realizându-se deci **(array_size-1)^2**
comparaţii. Sunt însă posibile unele îmbunătăţiri: astfel, la a doua parcurgere a şirului de sortat, se cunoaşte că **numbers[1]** a fost comparat şi eventual interschimbat cu **numbers[0]**; ca urmare, se poate începe comparaţia cu **numbers[2]**. În general, când se prelucrează **numbers[i]**, acesta trebuie comparat doar cu **numbers[i+1] ... numbers[array_size-1]**.

Se va scrie deci un ciclu interior după un indice **j** care parcurge elementele de la indicele **array_size-1** la **i** (**i** fiind indicele curent din ciclul exterior).

Deoarece în ciclul interior se vor compara **numbers[j]** şi **numbers[j-1]**, ciclul exterior începe cu indicele **1** nu cu **0**. Această îmbunătăţire reduce numărul de comparaţii la:

(array_size-1)+(array_size-2)+(array_size-3)+...+1=
array_size(array_size-1)/2

Numărul de interschimbări depinde de starea fişierului, cel mai defavorabil caz fiind un fişier iniţial cu chei distincte, sortat invers, caz în care sunt realizate **array_size(array_size-1)/2** interschimbări. În cazul în care fişierul este sortat, numărul de interschimbări este **0**.

Dacă la o parcurgere a buclei interioare nu s-a realizat nici o schimbare, atunci tabloul este sortat şi algoritmul se încheie. Acest lucru se poate implementa printr-o variabilă **sortat** care este iniţializată cu **1** în fiecare ciclu exterior şi cu **0,** dacă s-a realizat o interschimbare. În acest mod, numărul de parcurgeri ale buclei exterioare se reduce la 1, în cazul unui fişier iniţial sortat (se vor realiza doar **array_size-1** comparaţii) şi se îmbunătăţesc performanţele în cazul unui fişier aleator.

Denumirea metodei provine de la deplasarea elementelor **numbers[i]** către poziţia lor finală, asemănătoare cu mişcarea unor bule de gaz care se ridică într-un lichid.

Implementarea este următoarea:

```
void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
```

```
    }
  }
}
```

Acest algoritm este de ordinul ***O*(array_size^2).**

În figura 18.2 este ilustrată eficienta algoritmului de sortare "bubble sort" (timp de sortare în funcţie de numărul de elemente ale sirului de sortat). Elementele sirului iniţial sînt generate aleator.

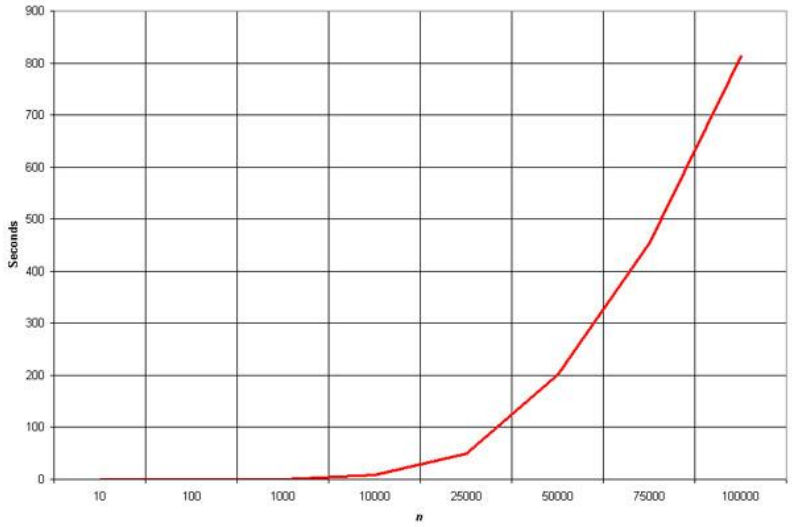


Figura 18.2 Eficienţa algoritmului de sortare bubble sort.

18.5. Sortarea prin selecţie

Metoda de sortare prin selecţie se bazează pe următorul algoritm general:

- se determină **numbers[k0]** - cel mai mic element dintre **numbers[0] ... numbers[array_size-1]**; se interschimbă acest element cu **numbers[0]**;

-se determină **numbers[k1]** - cel mai mic element dintre **numbers[1] ... numbers[n-1]**; se interschimbă acest element cu **numbers[1]**, etc.

Aceste operaţii se execută pentru toate elementele tabloului.

În această metodă, minimul se determină prin parcurgerea tuturor elementelor aflate la dreapta lui **numbers[i]** şi reţinerea atât a valorii minime, cât şi a indicelui elementului minim.

Numărul de comparaţii este acelaşi, independent de starea fişierului, fiind egal cu: (array_size-1)+(array_size-2)+...+2+1, adică **array_size(array_size-1)/2**. O îmbunătăţire considerabilă a performanţelor acestui algoritm se poate obţine prin memorarea adresei elementului minim şi nu a valorii sale (în special în cazul unor înregistrări de dimensiuni mari).

Interschimbarea se realizează după determinarea minimului. Această metodă conduce şi la un număr de interschimbări independent de starea fişierului: **array_size-1**.

Implementarea este următoarea:

```
void selectionSort(int numbers[], int array_size)
{
  int i, j;
  int min, temp;

  for (i = 0; i < array_size-1; i++)
  {
    min = i;
    for (j = i+1; j < array_size; j++)
    {
      if (numbers[j] < numbers[min])
        min = j;
    }
    temp = numbers[i];
    numbers[i] = numbers[min];
    numbers[min] = temp;
  }
}
```

Acest algoritm este de ordinul ***O*(array_size^2).**

În figura 18.3 este ilustrată eficienta algoritmului de sortare prin selecţie (timp de sortare în funcţie de numărul de elemente ale sirului de sortat). Elementele sirului iniţial sînt generate aleator.

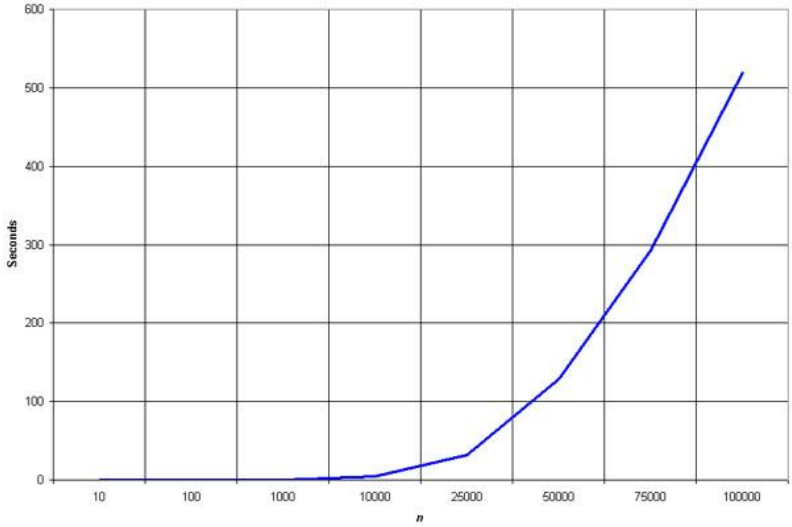


Figura 18.3 Eficienţa algoritmului de sortare prin selecţie.

18.6. Sortarea prin partiţionare şi interschimbare (QUICKSORT)

Algoritmul **Quicksort**, descoperit de C.A.R.Hoare, este un exemplu de algoritm performant de sortare, fiind de ordinul **nlog(n)**., unde **n** = **array_size**.

Funcţia care implementează algoritmul primeşte ca date de intrare: tabloul care trebuie sortat-**numbers**, prin adresa de început şi doi indici **left** şi **right**.

Se aleg:

- un element arbitrar al tabloului **numbers**, denumit **pivot** (variantele uzuale sunt **numbers[left]** sau **numbers[(left+right)/2]**;
- indicii **i** şi **j**, iniţializaţi cu **left**, respectiv **right**.

Cât timp **numbers[i] < pivot**, se incrementează **i**, apoi, cât timp **numbers[j] > pivot**, se decrementează **j**. Dacă **i <= j**, se interschimbă **numbers[i]** cu **numbers[j]**, actualizându-se indicii **i** şi **j**. Procesul continuă până când **i>j**.

În continuare, se apelează aceeaşi funcţie, cu indicii **left** şi **j-1**, respectiv **j+1** şi **right**.

Implementarea este următoarea:

```
void quickSort(int numbers[], int array_size)
{
  q_sort(numbers, 0, array_size - 1);
}

void q_sort(int numbers[], int left, int right)
{
  int pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
```

```
  if (left < pivot)
    q_sort(numbers, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, pivot+1, right);
}
```

În etapa de partiţionare, se efectuează comparaţii şi interschimbări de ordinul ***O*(n)**. Dacă se presupune că după fiecare partiţionare rezultă două subtablouri de aproximativ aceeaşi lungime, numărul de apeluri recursive va fi de ordinul ***O*(log(n))**. Astfel, ordinul total al algoritmului este: ***O*(nlog(n))**.

Exemplu: s-au listat elementele unui tablou de întregi de dimensiune 10, cu etapele corespunzătoare fiecărei iteraţii.

Tabelul 18.1.

[26	5	37	1	61	11	59	15	48	19]
[26	5	19	1	61	11	59	15	48	37]
[11	5	19	1	15	11	59	61	48	37]
[11	5	19	1	15]	26	[59	61	48	37]
[1	5]	1	19	15]	26	[59	61	48	37]
1	5	11	[19	15]	26	[59	61	48	37]
1	5	11	[19	15]	26	[59	61	48	37]
1	5	11	15	19	26	[59	61	48	37]
1	5	11	15	19	26	[59	37	48	61]
1	5	11	15	19	26	[48	37]	59	[61]
1	5	11	15	19	26	37	48	59	[61]
1	5	11	15	19	26	37	48	59	61

În figura 18.4 este ilustrată eficienta algoritmului de sortare “quick sort” (timp de sortare în funcţie de numărul de elemente ale sirului de sortat). Elementele sirului iniţial sînt generate aleator.

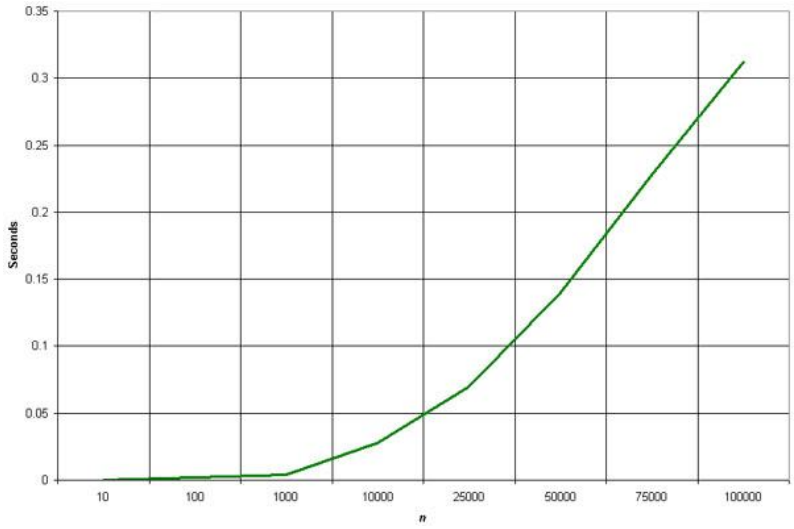


Figura 18.4 Eficienţa algoritmului de sortare "quik sort".

18.7. Sortarea HEAPSORT

Algoritmul **Heapsort** se bazează pe o structură de date numită **heap** sau **ansamblu**.

Un **heap descendent** este un arbore în care fiecare nod are asociată o cheie de valoare mai mică sau egală decât cea a nodului-părinte, proprietatea fiind valabilă pentru toţi subarborii. Similar se poate defini un **heap ascendent**, în care nodurile au asociate chei mai mari decât nodul părinte. În implementarea secvenţială a unui **heap descendent**, condiţia specificată anterior devine:

$$key[j] \le key[(j-1)/2], \quad \text{pentru } 0 \le (j-1)/2 < j \le n-1$$

key[j] fiind cheia (informaţia) asociată nodului de indice **j**.

În continuare, se va prezenta algoritmul **heapsort** pentr un **heap descendent** (pe scurt, **heap**).

Structura de tip **heap** este importantă, deoarece operaţiile de inserare, respectiv de ştergere a unui element în/din **heap**, cu păstrarea

tipului structurii se poate efectua eficient, cu un număr de operaţii de ordinul ***O*(log(n))**.

In figura 18.5 este ilustrat un arbore heap construit daca secvenţa initiala de numere este : 5,4,1,3,2.

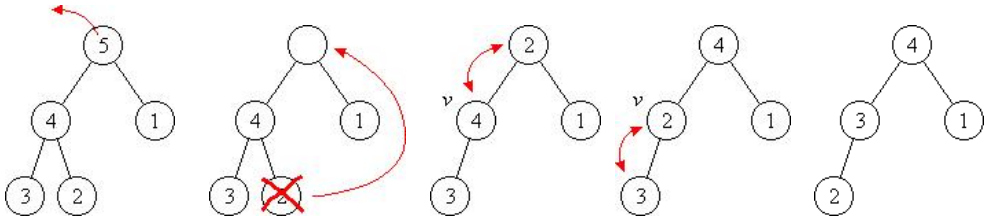


Figura 18.5. Arbore heap

Din arborele heap se extrage radacina, care se va înlocui cu ultimul nod din arbore. Arborele rezultat nu mai este arbore heap. Prin comparatii si

Din arborele heap se extrage radacina, care se va înlocui cu ultimu

radacină al arborelui rezultat dupa extragerea rădacinii, refacerea arborelui heap se va efectua conform algoritmului (s- a notat a(x) – valoarea cîmpului de informaţie a nodului x):

While **v** nu respecta inegalitatea specifica arborelui heap **do**
Alege descendentul direct al lui **v** , **w**, de informatie maxima a(**w**)
Interschimba a(**v**) cu a(**w**)
v = **w**

Implementarea algoritmului de sortare heap este următoarea:

```
void heapSort(int numbers[], int array_size)
{
  int i, temp;

  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array_size);

  for (i = array_size-1; i >= 1; i--)
  {
```

```
    temp = numbers[0];
    numbers[0] = numbers[i];
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
  }
}

void siftDown(int numbers[], int root, int bottom)
{
  int done, maxChild, temp;
  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;

    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
}
```

În figura 18.6 este ilustrată eficienta algoritmului de sortare “heap sort” (timp de sortare în funcţie de numărul de elemente ale sirului de sortat). Elementele sirului iniţial sînt generate aleator.

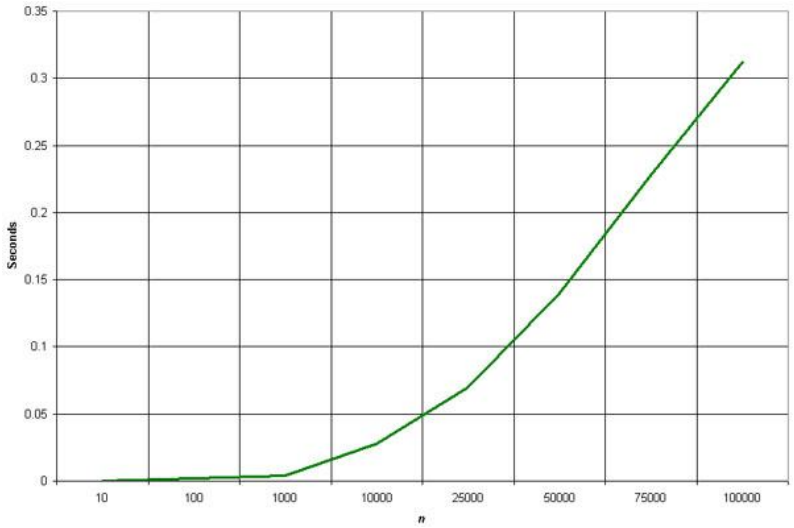


Figura 18.6 Eficienţa algoritmului de sortare "heap sort".

***Exemplu*:** se consideră şirul: 26, 5, 77, 1, 61, 11, 59, 15, 48, 19.

Etapele procesului de sortare **HEAP** sunt prezentate în figura 18.7.

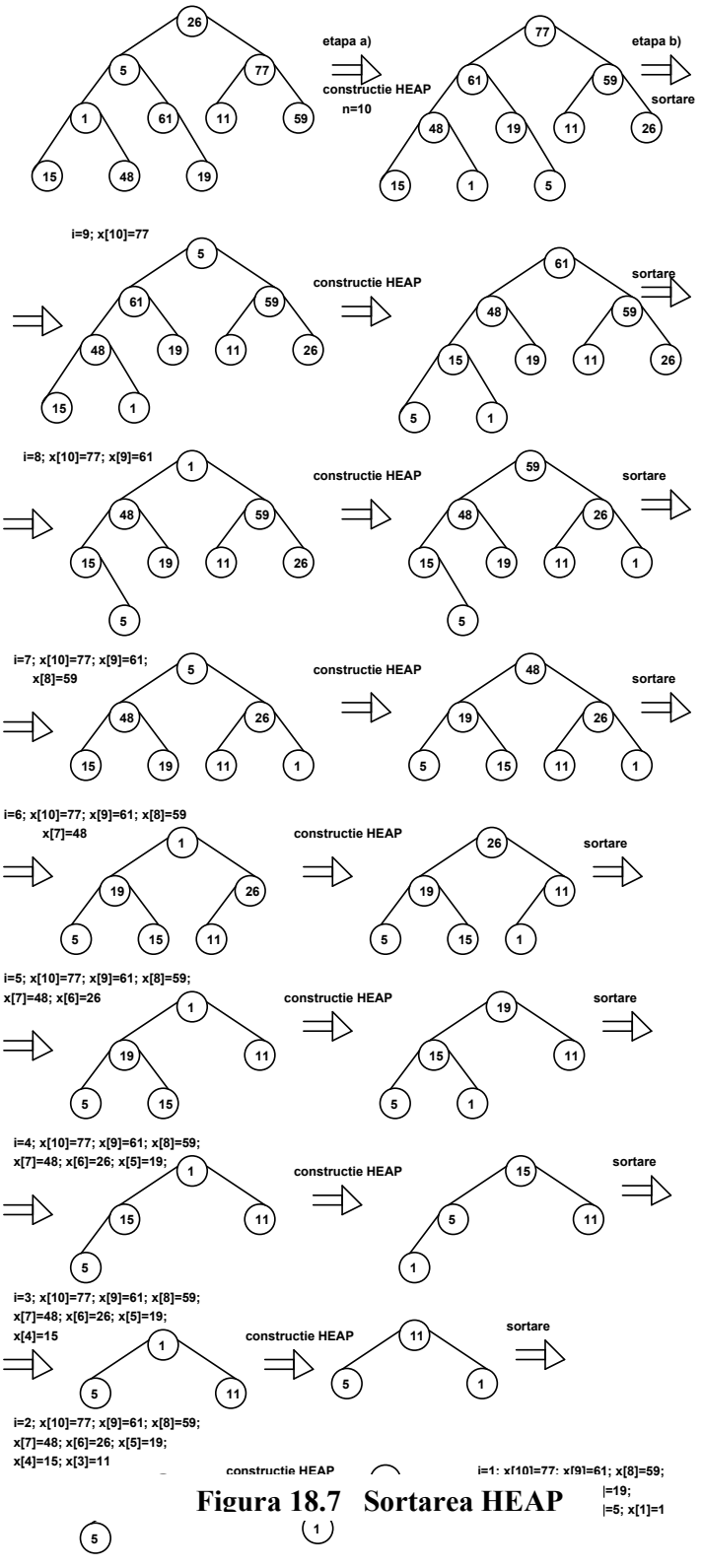


Figura 18.7 Sortarea HEAP

Fig. 7.4. Exemplu de sortare HEAP

19
Sortarea externă

Obiectul sortării externe constă în sortarea mulţimilor de date stocate în **memoria externă a calculatorului** (sortare de fişiere sau de benzi magnetice).

În continuare, se prezintă modul de sortare a **fişierelor secvenţiale**. Operaţiile uzuale necesare sortării fişierelor, sunt următoarele:

- comparaţia a două valori;
- citirea/ scrierea elementelor din/ în fişier;
- copierea de fişiere.

Toate cele trei operaţii necesită un timp mare de execuţie; ca urmare, pentru optimizarea algoritmilor de sortare externă, trebuie minimizate operaţiile de acces la disc.

19.1. Interclasarea a două fişiere

Se va ilustra algoritmul de interclasare, aplicându-l pentru două **tablouri** sortate de întregi **a** şi **b**, de dimensiuni **na** şi **nb**, în scopul obţinerii unui tablou sortat **c**, de dimensiune **na**+**nb**.

```
void interclasare(int a[], int b[], int c[], int na, int nb)
{
        int ia=0, ib=0, ic;
        for (ic=0;ia<na && ib<nb; ic++)
                if (a[ia]<b[ib])
                        c[ic]=a[ia++];
                else
                        c[ic]=b[ib++];
        while (ia<na)
                c[ic++]=a[ia++];
```

```
        while (ib<nb)
                c[ic++]=b[ib++];
}
```

Se compară câte un element din cele două tablouri **a** şi **b**, selectându-se elementul minim; acest element este copiat în tabloul **c**, avansând corespunzător indicele tabloului din care s-a copiat şi indicele tabloului **c**. Această operaţie continuă până când se epuizează cel puţin unul dintre tablourile **a** sau **b**, după care, dacă mai sunt elemente în tabloul care nu s-a epuizat, acestea se copiază în tabloul **c**.

Algoritmul se poate extinde la un număr oarecare **N** de tablouri care se interclasează. Problema este de a selecta elementul minim din **N** elemente date. Dacă **N** este mic, se vor efectua comparaţii directe, în caz contrar, utilizându-se un arbore de selecţie (**heap**) ascendent, în care se inserează elementele curente din cele **N** tablouri; rădăcina arborelui va fi elementul minim.

Interclasarea **fişierelor disc** presupune înlocuirea accesului la elementele tablourilor prin accesul la înregistrări de fişiere. Indicii tablourilor vor fi înlocuiţi cu indicatori de fişier. În faza de alegere a elementului minim, trebuie citite valori din ambele fişiere, dar trebuie avansat numai indicatorul din fişierul corespunzător valorii minime (ceea ce este mai complicat, deoarece funcţiile de citire avansează automat indicatorul fişierului).

O secvenţă de înregistrări aflate în ordine crescătoare într-un fişier va fi numită **monotonie**.

19.2. Interclasarea naturală cu două căi

În această metodă sunt utilizate:

- fişierul iniţial care trebuie sortat şi
- patru fişiere de lucru: **in[0], in[1], out[0]** şi **out[1]**.

Fie **file** fişierul care trebuie sortat; capacitatea memoriei interne este de **n** înregistrări, dispuse în tabloul **vec**. Se defineşte o variabilă globală **size**, care precizează dimensiunea unei înregistrări şi o funcţie globală **cmp()**, care compară două înregistrări.

Prima operaţie ce trebuie efectuată în **interclasarea naturală cu două căi**, se numeşte **distribuire** şi constă din următorul algoritm:

```
void distribute (FILE *file, FILE *out[2], size_t n)
{
        size_t i, m;
        i=0;
        do{
                m=fread(vec, size, n, file);
                if (m= =0)
                        break;
                qsort(vec, m, size, cmp);
                fwrite (vec, size, m, out[i]);
                i=(i+1)%2;
        } while(1);
}
```

Funcţia **distribute** distribuie monotonii de lungime **n** în fişierele **out[0]** şi **out[1]**, succesiv în modul următor: se citesc **n** înregistrări, fiecare de dimensiune **size**, în tabloul extern **vec**, funcţia de citire raportând numărul de înregistrări efectiv citite. Se sortează cu funcţia **qsort**() tabloul **vec**, utilizând o funcţie de comparaţie adecvată **cmp()**. Apoi se copiază tabloul **vec** în **out[i]** şi se incrementează **i** modulo 2. Operaţia continuă până se ajunge la sfârşitul fişierului **file** (**m= =0**). Se obţine astfel o distribuţie succesivă de monotonii de lungime **n** în cele două fişiere **out**. Ultima monotonie dintr-unul din fişierele **out** poate avea mai puţin de **n** înregistrări.

A doua operaţie ce trebuie efectuată în **interclasarea naturală cu două căi**, se numeşte **interclasare** şi constă din interclasarea câte unei monotonii de lungime **n** din două fişiere **xin[0]** şi **xin[1]** într-un fişier **xout.**

Se defineşte următoarea structură (care conţine un pointer la fişier şi numele fişierului):

```
typedef struct tape {FILE *fp; char *name;} TAPE;
```

În funcţia **merge()**, variabilele **a** şi **b** reprezintă buffere pentru câte o înregistrare citită din cele două fişiere. Variabilele **posa** şi **posb** memorează poziţiile curente în cele două fişiere, iar **ma** şi **mb** sunt contoare pentru numărul de înregistrări citite. Variabilele **na** şi **nb** memorează numărul de înregistrări efectiv citite din fişiere (ele au valorile 0 sau 1, deoarece se citeşte câte o înregistrare). Dacă **na= =0 (nb= =0)**, înseamnă că s-a ajuns la sfârşitul fişierului. Pentru citirea şi modificarea poziţiei indicatorului în fişier, se utilizează funcţiile **fgetpos** şi **fsetpos**. Funcţia **merge()** va întoarce 0 sau 1 (dacă mai sunt sau nu monotonii de interclasat în cel puţin unul din fişierele de intrare.

Algoritmul constă dintr-o buclă **do**, care se execută cât timp nu s-a ajuns la sfârşitul vreunuia din cele două fişere de intrare şi nu s-au citit **n** înregistrări din fiecare din cele două fişiere. Variabilele **ma** şi **mb** sunt iniţializate cu **n** (lungimea monotoniilor).

În bucla **do** se memorează poziţiile curente în cele două fişiere, după care se citeşte câte o înregistrare din fiecare fişier. Dacă s-a ajuns la sfârşitul unui fişier (**na= =0** sau **nb= =0**), se poziţionează indicatorul din celălalt fişier la valoarea memorată şi se iese din buclă.

Dacă s-au putut citi înregistrări din ambele fişiere, acestea se compară cu funcţia **cmp()** şi înregistrarea cu valoare mai mică este scrisă în fişierul **out**. Se decrementează contorul corespunzător fişerului din care s-a citit valoarea mai mică şi se repoziţionează indicatorul celuilalt fişier la vechea valoare.

Condiţia de execuţie a ciclului **do** este:

```
na= =1 && nb = =1 && ma>0 && mb>0
```

La ieşirea din ciclul **do** sunt posibile următoarele situaţii: dacă **ma>0** sau **mb>0**, atunci se copiază înregistrările rămase din monotonia respectivă în fişierul de ieşire, decrementând corespunzător **ma** sau **mb**.

Observaţie: trebuie efectuate câte o citire fictivă din fiecare fişier, fără a modifica poziţia actuală, pentru a interpreta corect atingerea sfârşitului de fişier. Acest fapt derivă din particularităţile funcţiilor de intrare/ieşire, care semnalează sfârşit de fişier numai în urma unei citiri fără succes.

Se întoarce programului apelant condiţia logică **na= =0 && nb= =0**, cu semnificaţia: cel puţin unul dintre fişiere mai conţine monotonii.

Implementarea este următoarea:

```
int merge (TAPE xin[2], TAPE xout, long n)
{
        FILE *in[2], *out;
        void *a=malloc(size);
        void b*=malloc(size);
        size_t na, nb;
        long ma, mb;
        fpos_t posa, posb;
```

```
in[0]=xin[0].fp; in[1]=xin[1].fp;
out=xout.fp;
ma=mb=n;
na=nb=0;
do {
        fgetpos(in[0], &posa);
        fgetpos(in[1], &posb);
        na=fread(a, size, 1, in[0]);
        if (na= =0) {
                fsetpos(in[1], &posb); break;
        }
        nb=fread(b, size, 1, in[1]);
        if (nb= = 0) {
                fsetpos(in[0], &posa); break;
        }
        if (cmp(a, b)>0) {
                mb--;
                fsetpos(in[0], &posa);
                fwrite(b, size, 1, out);
        }
        else {
                ma--;
                fsetpos(in[1], &posb);
                fwrite(a, size, 1, out);
        }
} while (na= =1 && nb= =1 && ma>0 && mb>0);

while (mb--) {
        nb=fread(b, size, 1, in[1]);
        if (nb= =0)
                break;
        fwrite(b, size, 1, out);
}

while (ma--) {
        na=fread(a, size, 1, in[0]);
        if (na= =0)
                break;
        fwrite(a, size, 1, out);
}
```

```
        fgetpos(in[0], &posa); fgetpos(in[1], &posb);
        na=fread(a, size, 1, in[0]);
        nb=fread(b, size, 1, in[1]);
        fsetpos(in[0], &posa); fsetpos(in[1], &posb);

        free(a); free(b);
        return na= =0 && nb= =0;
}
```

Metoda de sortare constă într-o distribuire iniţială şi apoi interclasări succesive de monotonii de lungime **n, 2n,** etc., până când această dimensiune este mai mare sau egală cu lungimea fişierului iniţial.

Cele patru fişiere temporare vor fi exploatate în modul "**wb**+", adică actualizare binară, cu ştergerea vechiului conţinut la deschidere. Deoarece aceste fişiere sunt succesiv fişiere de ieşire şi intrare, sunt necesare următoarele operaţii:

- poziţionarea indicatorului unui fişier deschis înaintea primei înregistrări, care se realizează cu funcţia standard **rewind()**;
- resetarea unui fişier (ştergerea vechiului conţinut şi poziţionarea indicatorului înaintea primei înregistrări); implentarea corespunzătoare este realizată cu următoarea funcţie (care închide fişierul şi îl deschide în acelaşi mod):

```
void reset (TAPE *x)
{
        fclose(x->fp);
        x->fp=fopen(x->name, "wb+);
}
```

Funcţia **rewind()** se va aplica fişierelor de intrare, iar **reset()** fişierului de ieşire, înaintea unei interclasări.

Funcţia prezentată în cele ce urmează realizează operaţia de sortare externă. Această funcţie primeşte numele **sursa** al fişierului de sortat şi dimensiunea **nn** a monotoniei iniţiale. În funcţie se declară cele patru structuri TAPE temporare, grupate în tablourile **in** şi **out**, o structură temporară **temp** care va fi utilizată la interschimbarea lui **in** cu **out** şi o structură **final**, care va memora fişierul de ieşire curent. La sfârşitul algoritmului, **final** va conţine fişierul cu date sortate.

Funcţia începe prin a deschide pentru citire fişierul **sursa** şi a-i calcula lungimea (numărul de înregistrări). Acest lucru este realizat cu funcţiile standard **fseek()** şi **ftell()**.

Se crează apoi fişierele temporare. Numele lor se obţin cu funcţia standard **tmpnam()**, care furnizează nume de fişiere, garantând că acestea nu se repetă. Aceste nume se copiază în structurile de tip TAPE şi se folosesc şi la deschiderea propriu-zisă cu **fopen()**. Copierea se realizează cu **strdup()**.

Monotoniile iniţiale sunt distribuite în fişierele **in[0]** şi **in[1]**.

Se execută un ciclu după lungimea monotoniilor **n**, din care se iese în momentul în care **n** este mai mare sau egal cu lungimea fişierului de sortat.

Bucla de interclasare se continuă până când funcţia **merge** indică sfârşitul ambelor fişiere de intrare. Ca fişiere de ieşire se utilizează succesiv **out[0]** şi **out[1]**. Dacă lungimea monotoniilor curente nu a depăşit lungimea fişierului, se dublează lungimea monotoniei, se interschimbă fişierele de intrare cu cele de ieşire şi se continuă bucla exterioară.

Se redeschide fişierul iniţial **sursa**, pentru scriere şi se copiază **final** în acest fişier. Se şterg fişierele temporare şi se revine în programul apelant.

Implementarea este:

```
void nat_merge(char *sursa, size_t nn)
{
        TAPE in[2], out[2], temp, final;
        FILE *fp=fopen(sursa, "rb"), *xin[2];
        int i, j, stop;
        long n=nn, lng_fis;

        if (fp= =NULL)
        err_exit ("Merge: nu se poate deschide fisierul sursa");
        fseek(fp, 0L, SEEK_END);
        lng_fis=ftell(fp)/size;
        rewind(fp);

        for(i-0; i<2); i++) {
in[i].fp=fopen(in[i].name=strdup(tmpnam(NULL)), "wb+");

out[i].fp=fopen(out[i].name=strdup(tmpnam(NULL)), "wb+");
                if(in[i].fp= =NULL||out[i].fp= =NULL)
```

```
        err_exit("Merge: eroare deschidere fisiere de lucru);
        }

        xin[0]=in[0].fp; xin[1]=in[1].fp;
        distribute(fp, xin, n); fclose(fp);
        while(1) {
                for (i=0; i<2; i++) {
                        rewind(in[i].fp; reset(&out[i]);
        }
        i=stop=0;

        do {
                stop=merge(in, final=out[i], n);
                i=(i+1) % 2;
        } while (!stop);

        if (n>lng_fis)
                break;
        for (j=0; j<2; j++) {
                temp=in[j]; in[j]=out[j]; out[j]=temp;
        }
        n*=2;
        }
        rewind(final.fp);
        fp=fopen(sursa, "wb");
        if (fp= =NULL)
        err_exit ("Merge: nu se poate deschide fisierul sursa");
        do {
                n=fread(vec, size, DIM_MAX/size, final.fp);
                if (n= =0)
                        break;
                fwrite(vec, size, n, fp);
        } while (1);
        fclose(p);

        for (i=0; i<2; i++) {
                fclose(in[i].fp); fclose(out[i].fp);
                remove(in[i].name); remove(out[i].name);
                free(in[i].name); free(out[i].name);
        }
}
```

Se mai defineşte o funcţie de listare fişier, care se va apela înainte şi după sortare. Funcţia are ca parametru numele fişierului şi apelează o funcţie externă **print_rec** pentru a tipări o înregistrare. Se declară o variabilă **n_row**, care precizează câte înregistrări se vor scrie pe o linie la consolă. Drept buffer de citire se vautiliza acelaşi tablou **vec**, ca şi la funcţia de distribuţie. Constanta globală **DIM_MAX** precizează dimensiunea, în octeţi, a tabloului **vec**.

Funcţia de listare este:

```
void list_file(char *file)
{
        size_t i, n, dim;
        long j=0;
        FILE *fp=fopen(file, "rb");
        if (fp= =NULL)
                err_exit("List_file: eroare deschidere citire");
        printf("\n");
        dim=DIM_MAX/size;
        do {
                n=fread(vec, size, dim, fp);
                if (n= =0)
                        break;
                        for (i=0; i<n; i++; j++) {
                                print_rec(vec+i*size);
        printf("%c, ((j%n_row)= =n_row-1) ? '\n': ' ');
                        }
        } while (1);
        printf("\n");
        fclose(fp);
}
```

Funcţia de tratare a erorilor este:

```
void err_exit(s)
{
        fprintf(stderr, "\n%s\n", s);
        exit(1);
}
```

Programul principal este listat mai jos.

Pentru ca programul să fie universal (să poată sorta orice fel de fişiere), funcţia de comparaţie **cmp()**, variabila **size**, funcţia de tipărire a unei înregistrări **print_rec** şi variabila **n_row** sunt definite ca fiind externe. Ele trebuie precizate într-un modul de program separat, care se va lega cu modulul în care sunt definite funcţiile de sortare.

Numele fişierului care se sortează este preluat ca argument al funcţiei **main**.

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

typedef unsigned char BYTE;
typedef int (*FCMP) (const void*, const void*);
typedef struct tape {FILE *fp; char *name; } TAPE;
extern size_t size;
extern int cmp(const void*, const void*);
extern int n_row;
extern void print_rec(void*);
extern void qsort(void*, size_t size_t, FCMP);

#define DIM_MAX 1000
BYTE vec[DIM_MAX];

void main (int argc, char **argv)
{
        if (argc!=2)
                err_exit("Merge: sintaxa este: merge<file>");
        list_file(argv[1]);
        nat_merge(argv[1], DIM_MAX/size);
        list_file(argv[1]);
}
```

20
Programarea în C++ cu clase

20.1. Definirea şi utilizarea unei clase

Un tip **clasă** cuprinde date şi funcţii într-o singură entitate. Funcţiile unei clase, denumite şi **funcţii membru** sau **metode**, implementează operaţiile specifice datelor din clasa respectivă.

Un tip **clasă** este o extindere a unui tip **structură**, prin includerea de funcţii alături de variabilele componenete ale structurii. În C++ un tip **struct** poate include şi funcţii, fiind considerat ca o variantă a tipului **class**.

O variabilă de tip **clasă** se numeşte şi **obiect**, de unde şi expresia "Programare orientată pe obiecte", în loc de "Programare cu clase".

Pentru definirea unei clase, se poate folosi unul din cuvintele cheie **class, struct** sau **union**, dar fiecare din ele are implicaţii asupra accesului la membrii clasei dupa cum urmează:

- o clasă definită cu **struct** sau **union** are implicit toţi membrii publici, deci accesibili pentru funcţii din afara clasei;
- o clasă definită cu **class** are implicit toţi membrii locali, deci inaccesibili din afara clasei.

Este posibilă şi declararea explicită a nivelului de acces la fiecare membru al unei clase, folosind cuvintele cheie **public, privat** şi **protected**.

În practică, este necesar ca datele clasei să fie inaccesibile din afară, dar ca funcţiile clasei să poată fi apelate din afara clasei. Ca urmare, utilizarea unui cuvânt cheie de control al accesului este necesară, oricare ar fi modul în care s-a definit clasa.

Exemplu de definire a unei clase constând dintr-un număr întreg şi din operaţiile necesare pentru a folosi acest întreg:

```
class intobj {
        int i;
public:
```

```
        void set(int  v=0) {i=v;}
        void get() {return i;}
}
```

Deşi ordinea în care se definesc membrii unei clase nu este impusă, se obişnuieşte să se înceapă cu datele (variabilele) şi să se încheie cu metodele clasei.

Funcţiile unei clase, dacă sunt scurte, se pot defini chiar în cadrul clasei.

O altă variantă de a defini clasa **intobj** este următoarea:

```
class intobj {
        int i;
public:
        void set (int v=0);
        int get();
};


void intobj::set (int v)
{ i=v
}
int intobj::get ()
{ return i;
}
```

Observaţie: în C++, este definit operatorul de rezoluţie **::**, care este utilizat înaintea unui nume de variabilă (pentru a deosebi o variabilă globală de o variabilă locală cu acelaşi nume) şi la definirea funcţiilor membru dintr-o clasă, efectuată în afara clasei (pentru a le deosebi de alte funcţii cu acelaşi nume din alte clase, sau de funcţii nemembru).

Practica programării cu clase recomandă ca definiţiile claselor să fie plasate în fişiere antet (de tip **.H**), iar definiţiile funcţiilor membru care au fost doar declarate în clasă să fie plasate într-un fişier sursă (de tip **.CPP**) cu acelaşi nume cu fişierul antet şi care include fişierul antet.

Fişierul antet cu definiţia clasei (sau a unei familii de clase) trebuie inclus şi în fişierele sursă unde se declară şi se folosesc obiecte din clasa respectivă.

O reluare a exemplului anterior, extins cu un program care realizează operaţii cu două obiecte de tipul **intobj** este următoarea:

```
// Fisierul INTOBJ.H
class intobj {
        int i;
public:
        void set (int v=0);
        int get ();
};

// Fisierul INTOBJ.CPP
#include "intobj.h"
void intobj::set (int v) {
        i=v;
}
int intobj::get() {
        return i;
}

// Fisierul TESTOBJ.CPP
#include "intobj.h"
#include <iostream.h>
main() {
        intobj x, y;
        x.set(); y.set(3);
        cout<<"x:"<<x.get()<<"\n";
        cout<<"y:"<<y.get()<<"\n";
        y.set(x.get());
        cout<<"y:"<<y.get()<<"\n";
}
```

Obsevaţie: în C++, în fişierul antet <**iostream.h**>, sunt predefinite variabilele de tip clasă, denumite **cin** şi **cout**, pentru citire şi respectiv scriere de la/la consolă.

Operatorul **<<** (utilizat numai cu obiectul **cout**) are sensul: "trimite listă valori la consolă", iar operatorul >> (utilizat numai cu obiectul **cin**) are sensul: "primeşte una sau mai multe valori de la consolă".

Utilizarea membrilor publici ai unei clase este similară cu cea a componentelor unei structuri, deci numele funcţiei membru trebuie precedat de punct şi de numele variabilei clasă la care se referă.

Ca şi în cazul tipului **struct**, este permisă atribuirea între obiecte dintr-o aceeaşi clasă.

Pentru exemplul prezentat anterior, operaţiile de afişare a obiectelor unei clase se pot simplifica, dacă se adaugă clasei o funcţie de afişare, astfel:

```
class intobj {
        int i;
public:
        void set (int v=0);
        int get ();
        void show (char *msg);
};

void intobj::show (char *msg) {
        cout<<msg<<i<<"\n";
}

main () {
...
intobj x, y;
x.set; x.show("x:");
y.set(7); y.show("y:");
...
}
```

Aşa cum este definită clasa **intobj** este posibil să apară erori de utilizare a unor obiecte neiniţializate (declarate dar nedefinite). Pentru a preveni asemenea erori, se poate recurge la iniţializarea obiectelor simultan cu crearea lor şi nu ulterior.

Iniţializarea simultană cu crearea unui obiect se realizează în C++ prin definirea unei metode speciale în fiecare clasă, denumită funcţie **constructor**.

Un **constructor** este o funcţie fără tip (nici chiar **void**) şi care are acelaşi nume cu clasa din care face parte. Funcţia **constructor** este apelată automat la crearea unui obiect. În cazul obiectelor de date alocate dinamic, funcţia **constructor** realizează şi alocarea dinamică a memoriei.

Funcţia **constructor** poate fi supradefinită (pot exista într-o clasă mai mulţi constructori cu argumente diferite, dar cu acelaşi nume).

Exemplu:

```
class intobj {
        int i;
```

```
public:
        intobj (int vi=0) {i=vi;}       // constructor
        void show () {cout<<i<<"\n";}
};
main() {
        intobj x, y(7);
        x.show(); y.show();
x=3; x.show();
}
```

Aşa cum există o metodă **constructor**, activată la apariţia unui nou obiect, tot aşa există o metodă **destructor**, activată la dispariţia unui obiect. Funcţia **destructor** are acelaşi nume cu clasa şi cu funcţia **constructor**, dar precedat de simbolul ~ (este opusul funcţiei constructor). Ca şi funcţia **constructor**, funcţia **destructor** nu are tip.

20.2. Implementarea de noi tipuri prin clase

O clasă poate reuni în mod natural definiţia noului tip, cu operaţiile specifice acestui tip, oferind posibilitatea verificării de către compilator a utilizării corecte a datelor din noul tip. La fel ca şi pentru tipurile predefinite, nu se admite decât folosirea operaţiilor specifice tipului de date, prin intermediul metodelor clasei.

În cele ce urmează, se va realiza introducerea în C++ a tipului **set** din Pascal (corespunzător unei mulţimi de valori întregi, pozitive).

Exemplu de definire a unei clase **set**:

```
#include <iostream.h>

class set {
        unsigned int m;
public:
set() {m=0;}
set (int elm) {m=1<<elm;}
set&plus (int elm) {m=m|(1<<elm); return *this;}
set&minus (int elm) {m=m&~(1<<elm); return *this;}
int este (int elm) {return m&(1<<elm);}
set&plus_set (set&mul) {m=m|mul.m; return *this;}
set&inters (set&mul) {m=m&mul.m; return *this;}
```

```
set&minus_set (set&mul) {m=m&~(mul.m); return *this;}
void print (char *msg);
};
```

Se observă că în clasa **set** sunt definiţi doi constructori: unul fără argumente, care crează o mulţime vidă şi unul cu un argument întreg, care crează o mulţime cu un element iniţial. În descrierea anterioară **mul** este un obiect de tipo **set** .

Cuvântul cheie "this"

La apelarea unei metode, aceasta trebuie să primească adresa obiectului pentru care se apelează.

Funcţia **print()** sau oricare altă metodă a clasei **set** (din exemplul anterior), se poate referi la variabila pointer primită ca argument implicit prin cuvântul **this**, care ar putea fi tradus prin "pointer către acest obiect". De exemplu, funcţia **constructor**:

```
set() {m=0;}
```

poate fi explicitată prin:

```
set (set *this) {this->m=0;}
```

Cuvântul **this** a fost introdus în limbaj, deoarece există situaţii în care funcţiile unei clase trebuie să folosească explicit acest pointer către obiectul cu care lucrează.

20.3. Crearea de noi operatori utilizând clase

Limbajul C++ permite supradefinirea operatorilor limbajului C (cu excepţia următorilor: **., ::, ?:,** #), deci crearea unor noi operatori folosind simbolurile existente în alfabetul limbajului şi menţinând semnificaţiile anterioare ale acestor simboluri.

Supradefinirea unui operator se realizează considerându-se operatorul ca o funcţie
cu un nume special şi cu o utilizare specială. Numele funcţiei operator constă din cuvântul **operator** urmat de simbolul operatorului supradefinit.

Rezultatul unei funcţii operator poate fi de orice tip diferit de **void**; în cele mai multe cazuri este o referinţă la o clasă.

Pentru ca o funcţie să poată opera cu datele proprii unei clase, există două posibilităţi:

- funcţia este şi ea membru al clasei care conţine datele;
- funcţia este definită ca "prieten" al clasei ce conţine datele.

Fie următoarea definiţie care îi dă operatorului *****, sensul de **intersecţie a două mulţimi**:

```
set & set:: operator * (set & mul) {
        m=m & mul.m;
        return *this;
}
```

Deci, utilizarea operatorului definit anterior este următoarea:

```
set a, b, c;
c=a.operator * (b);
```

Prin convenţie, limbajul C++ admite şi forma:

```
c = a * b;
```

Această formă este similară cu utilizarea operatorilor predefiniţi pentru tipurile de date predefinite ale limbajului.
Este admisă şi utilizarea:

```
d = a * b * c;  // set d;
```

În acest caz, compilatorul preia de la programator sarcina de a reduce o expresie cu mai mulţi operanzi la mai multe expresii cu câte doi operanzi, generând obiecte temporare anonime:

```
t = a * b; d =t * c; // obiectul temporar t este generat automat.
```

20.4. Funcţii prieten (friend)

În cadrul unei clase se pot declara funcţii **prieten** cu acea clasă, folosind cuvântul **friend** înainte de tipul funcţiei.

O funcţie **prieten** primeşte de la clasa care o anunţă dreptul de acces la membrii **private** ai clasei, deci la datele din cadrul acelei clase.

Una dintre aplicaţiile importante ale funcţiilor **prieten** este aceea de supradefinire a unui operator sub forma unei funcţii **prieten**, sau prin mai multe funcţii operator prieten cu acelaşi nume, dar cu argumente diferite.

Exemplu:

```
#include <iostream.h>
class set {
        unsigned int m;
public:
set() {m=0;} //constructor
set(int elm) {m=1<<elm;} //constructor
friend set operator + (set &, set &);
set operator + (set&a, set&b)
                        {
                         set c;
                        c.m=a.m | b.m;
                        return c;
                        }
}
```

20.5. Clase pentru tipuri abstracte de date în C++

Un tip de date abstract (TDA) este un tip de date despre care se ştie cum va fi utilizat (ce operaţii sunt posibile), dar nu este precizată implementarea.

Exemple de tipuri abstracte şi de structuri de date prin care pot fi implementate:

- **mulţime**: vector de octeţi, de biţi, listă înlănţuită;
- **stivă**: vector, listă înlănţuită;
- **arbore**: vector de întregi, de înregistrări, structură arborescentă cu pointeri;
- **graf**: matrice de adiacenţă, vectori de noduri şi de succesori, listă înlănţuită de liste, etc.

Separarea **tip abstract-structură concretă** permite modificarea ulterioară a implementării iniţiale a unui tip abstract (de exemplu, din considerente de performanţă).

Programarea pe obiecte permite "încapsularea" datelor din structura concretă într-o clasă, împreună cu funcţiile de acces la structură, făcând inaccesibile aceste date pentru orice alte instrucţiuni din afara metodelor.

Avantajele utilizării de clase pentru tipuri abstracte de date sunt:
- protejarea datelor la modificări ulterioare;
- posibilitatea de a modifica implementarea tipului abstract, fără consecinţe asupra programelor care utilizează acest tip;
- posibilitatea adaptării la diferite cerinţe, fără modificarea clasei;
- posibilitatea de a combina diferite tipuri abstracte de date în structuri mai complexe.

Ca exemplu, se va considera tipul abstract **stivă de întregi**, cu următoarele operaţii asociate:

- iniţializarea unei stive;
- introducerea unui întreg într-o stivă (cu test de stivă plină);
- extragerea întregului din vârful stivei (cu test de stivă vidă).

```
        // Fisier STIVA1.H

const int SIZE=100;

typedef int tip;        //tip componente stiva

// definitia clasei
class stack {
        tip st[SIZE];           //stiva
        int top;                //top of stack=vârf stiva
public:
        stack() {top=0};
        int push (tip elem);
        int pop (tip &elem);
};

        // Fisier STIVA1.CPP

#include "stiva1.h"
```

```
int stack:: push (tip elem) {
        if (top<SIZE) {
                st[top++]=elem;
                return 0;
        }
        else
                return -1;
}

int stack:: pop(tip&elem) {
        if (top>0) {
                elem=st[--top];
                return 0;
        }
        else
                return -1
}
```

// Fisier CUSTIVA1.CPP

```
#include <iostream.h>
#include <stiva1.h>

        // program test

main () {
        stack s; int x;
        while (cin>>x)
                if (s.push(x)= = -1)
                        cout<<"stiva plina\n";
        while (s.pop(x)!= -1)
                cout <<x<<" ";
}
```

20.6. Familii de clase

Clasele şi obiectele folosite într-un program modelează obiecte reale din aplicaţia pentru care se scrie acel program.

Obiectele reale pot fi complet independente sau pot exista anumite relaţii între obiectele modelate.

Limbajul C++ permite exprimarea următoarelor tipuri de relaţii între obiecte:

- **relaţia de descendenţă**: o clasă D descinde dintr-o clasă B (clasa D este derivată din clasa B), moştenind de la clasa de bază B toate componentele (datele) şi metodele acesteia (la care se pot adăuga date şi metode specifice clasei derivate);
- **relaţia de compunere**: o clasă compusă C poate conţine ca membrii obiecte
dintr-o clasă B sau pointeri către obiecte de alte tipuri;
- **relaţia de prietenie**: o clasă prieten P primeşte de la o clasă B dreptul de acces la datele locale (private) din clasa B, dar nu moşteneşte date sau metode de la B.

Cea mai importantă şi mai des întâlnită este relaţia de descendenţă, care poate da naştere unei **familii de clase** înrudite, cu o structură ierarhică.

Pentru definirea unei clase D derivate dintr-o clasă de bază B, se utilizează următoarea sintaxă:

```
class D: public B {
...// date specifice clasei D
public:
...//metode specifice clasei D
};
```

Cuvântul **public** arată că este o derivare publică, care păstrează pentru componentele moştenite acelaşi nivel de accesibilitate din clasa de bază.

Pentru a fi utlizată drept clasă de bază ("părinte") pentru crearea de clase derivate, o clasă B trebuie să aibă datele protejate (**protected**), adică accesibile pentru clasele derivate din ea, dar inaccesibile pentru clase sau funcţii "străine":

```
class B {
protected:
... // date locale lui B
public:
...// metodele clasei B       };
```