

Cler
Tudor Sorin

Chelba

Informatica

**Manual pentru clasa a XI-a
Varianta C++**

*Bdul Stefan cel Mare nr 11
Sector 1
Bucuresti*

© Toate drepturile sunt rezervate editurii L&S INFOMAT

**Editura L&S INFOMAT
Bucureşti**

Prefată

Lucrarea este organizată pe mai multe niveluri de accesibilitate, aşa cum rezultă din explicațiile ce urmează.

Nivelul 1. La acest nivel sunt incluse cunoștințele minime, cerute de programă. Paragrafele nemarcate și problemele nemarcate aparțin acestui nivel. O excepție de la această regulă o constituie **capitolul 7**.

Nivelul 2. Contine cunoștințe suplimentare, necesare elevilor care studiază intensiv informatică. Toate paragrafele și problemele corespunzătoare acestui nivel sunt marcate cu *.

Nivelul 3. Paragrafele și problemele de la acest nivel se adresează elevilor pasionați de informatică și care doresc să se pregătească pentru diverse concursuri de specialitate. Toate paragrafele și problemele corespunzătoare acestui nivel sunt marcate cu **.

Nivelul 4. Contine cunoștințe care ajută la buna înțelegere a materiei, dar care devin indispensabile atunci când se scrie un program care se utilizează în practică. Mai precis, în **capitolul 7**, veți găsi o inițiere în *Programarea Vizuală*. Studiind acest capitol veți înțelege utilitatea programării orientate pe obiecte și veți putea realiza programe care prezintă o interfață asemănătoare produselor soft existente pe piață. Elevii care au studiat limbajul **Pascal** vor aborda *Programarea vizuală* în **Borland Delphi**, iar cei care au studiat **C++** vor aborda în **Borland C++ Builder**. Pe lângă avantajele enumerate, mai există unul, pe care-l considerăm esențial. Elevii vor învăța să folosească acele componente vizuale pe care, ca utilizatori, le-au întâlnit în orice program. În acest mod ei devin eficienți, învățând să folosească în propriile programe soft consacrat. Menționăm că în acest studiu nu prezintă importanță *limbajele prezentate*, ci maniera de utilizare a componentelor puse la dispoziție de acestea și capacitatea de a folosi și altele, neprezentate în curs, prin *autoinstruire*. Acest capitol este facultativ.

În lucrare veți mai întâlni următoarele notătii:

Lucrare în echipă. Are semnificația de aplicație complexă, care trebuie realizată de un colectiv de elevi. Colectivul va fi constitut din 4-6 elevi.

- Unul dintre ei va prelua funcția de **analist**. Aceasta va proiecta aplicația, prin consultare cu ceilalți colegi. În urma analizei se va obține o documentație care conține: tipurile de date necesare, descrierea lor, antetele subprogramelor care se vor realiza.

- Sarcina scrierii programelor (subprogramelor) revine celorlalți elevi, numiți *programatori*.
- La sfârșitul unei astfel de aplicatii se va elabora documentația finală, care trebuie să fie tehnoredactată.

3. Problemă care trebuie rezolvată cu o aplicație de tip „Lucrare în echipă”, prin consultarea documentației. Elevii vor învăța să consulte documentația aplicației și să utilizeze programele (subprogramele) pe care aceasta le conține. De multe ori, în această etapă se descoperă anumite erori ascunse (bug-uri) ale aplicației. Acestea vor fi comunicate realizatorilor, în scopul eliminării lor.

Lucrare tip referat. Conține cunoștințe care nu au fost incluse în manual. Trebuie elaborată de unul sau mai mulți elevi. Aceștia găsesc informațiile necesare prin accesarea Help-ului atașat mediului de programare, pe Internet sau în literatura de specialitate. Lucrarea trebuie prezentată tuturor elevilor din clasă.

Autoinstruire. Un set de probleme, prin a căror rezolvare elevul este condus la descoperirea unor rezultate teoretice importante, care nu au fost prezentate în manual, sau spre rezolvarea unor probleme considerate dificile.

Capitolul 1

Alocarea dinamică a memoriei

Anumite elemente care privesc alocarea dinamică a memoriei au fost studiate anul trecut. În acest an vom extinde acest studiu.

1.1 Variabile de tip pointer

Am învățat faptul că memoria internă poate fi privită ca o succesiune de octeți. Pentru a-i distinge, aceștia sunt numerotați. *Numărul de ordine al unui octet se numește adresa lui.*

Orice variabilă ocupă un număr de octeți succesivi. De exemplu, o variabilă de tip *int* ocupă doi octeți (în varianta *Borland C++ 3.0*).

Definiție. Adresa primului octet al variabilei se numește adresa variabilei.

Observații:

- Nu trebuie confundată adresa unei variabile cu valoarea pe care aceasta o memorează.
- Uneori, în loc de adresa a unei variabile vom folosi termenul *pointer*.

Memorarea adreselor variabilelor se face cu ajutorul variabilelor de tip *pointer*.

⇒ Variabilele de tip *pointer* se caracterizează prin faptul că valorile pe care le pot memora sunt adrese ale altor variabile.

Limbajul C++ face *discriminare* între natura adreselor care pot fi memorate. Astfel, există adrese ale variabilelor de tip *int*, adrese ale variabilelor de tip *float*, adrese ale variabilelor de tip *char*, etc. Din acest motiv și tipul variabilelor de tip *pointer* este diferit.

⇒ Tipul unei variabile de tip *pointer* se declară ca mai jos:

*tip *nume*

Exemple:

- Variabile de tip *pointer* către variabile de tip *int*. Variabilele *adr1*, *adr2*, pot reține adrese ale variabilelor de tip *int*:

```
int *adr1, *adr2;
```

- Variabilă de tip pointer către variabile de tip float. Variabila **adresa**, poate retine adrese ale variabilelor de tip float.

```
float* adresa;
```

- Variabile de tip pointer către variabile de tip elev, care în sfârșit sunt de tip struct. Variabilele **a** și **b**, pot retine adrese ale variabilelor de tipul elev.

```
struct elev
{
    char nume[20], prenume[20];
    float nota_mate, nota_info;
    int varsta;
};

elev *a,*b;
```

- Caracterul '*' poate fi așezat în mai multe feluri, după cum se observă:

```
int* adr1; int * adr1; int *adr1;
```

- Pentru a declara mai multe variabile de acest tip, caracterul '*' se trece de fiecare dată: int *adr1, *adr2, *adr3;

- O declaratie de genul: int* adr1, adr2; are semnificația că adr1 este de tip pointer către int, în vreme ce adr2 este de tip int. Atentie! Aici se greseste deseori...

- Adresa unei variabile se obține cu ajutorul operatorului de referințiere '&', care trebuie să preceadă numele variabilei:

```
&Nume_variabila;
```

Exemplu: **adr1=&numar** Variabila **adr1** își se atribuie adresa variabilei **numar**.

- Fiind dată o variabilă de tip pointer către variabile de un anume tip, care memorează o adresă a unei variabile de acel tip, pentru a obține continutul variabilei la cărei adresă este memorată se utilizează operatorul unar '*', numit și operator de dereferințiere.

Exemple:

- Variabila **a** este initializată cu 7, iar variabila **adra** este initializată cu adresa lui **a**. Secvența afișează continutul variabilei **a** (7) pornind de la adresa ei, reținută de **adra**:

```
int a=7, *adra=&a;
cout<<*adra;
```

- Variabila **a**, de tip elev, este initializată, iar variabila **adra**, de tip pointer către variabile de tip elev este initializată cu adresa variabilei **a**. Secvența tipărește conținutul variabilei **a**.

```
...
struct elev
{
    char nume[20], prenume[20];
};

elev a, *adra=&a;
strcpy(a.nume, "Bojan");
strcpy(a.prenume, "Andronache");
cout<<(*adra).nume<< " "<<(*adra).prenume<<endl;
```

- Observați modul în care am obținut conținutul unui câmp al variabilei **a**, pornind de la pointerul către **a**: **(*adra).nume**. De ce este nevoie de paranteze?

Operatorul '...' -numit operator de selecție, are prioritatea 1, deci maximă.

Operatorul '*' -unar, numit și operator de dereferințiere, are prioritatea 2, mai mică. Prin urmare, în absența parantezelor rotunde, se încearcă mai întâi evaluarea expresiei **adra.nume**, expresie care n-a sens! Parantezele schimbă ordinea de evaluare, se evaluatează mai întâi ***adra**, expresie care are sens.

⇒ Pentru o astfel de selecție, în loc să folosim trei operatori, se poate utiliza unul singur, operatorul de selecție indirectă: '**->**'. Acesta accesează un câmp al unei structuri pornind de la un pointer (adresă) către acea structură. El are prioritatea maximă -vezi tabelul operatorilor. Tipărirea se poate face și asta:

```
cout<<adra->nume<< " "<<adra->prenume;
```

- Între variabile de tip pointer sunt permise atribuirile doar în cazul în care au același tip pointer (retin adrese către același tip de variabile).

Exemplu:

```
int *adr1, *adr2;
float *adr3;
```

Atribuirea **adr1=adr2** este corectă;
Atribuirea **adr3=adr2** nu este corectă.

În aceste condiții vă puteți întreba: cum putem atribui continutul unei variabile de tip pointer către tipul `x`, altor variabile de tip pointer către tipul `y`? În definitiv, amândouă rețin o adresă... în acest caz se utilizează operatorul de conversie explicită. De această dată, pentru exemplul anterior, atribuirea: `adr3=(float*)adr2` este corectă.

1.2 Alocarea dinamică a memoriei

Anumite variabile pot fi alocate dinamic. Asta înseamnă că:

- Spatiul necesar memorării este rezervat într-un segment special destinat acestui scop, numit **HEAP**.
- În memorie se rezervă spațiu în timpul executării programului, atunci când se utilizează un anumit operator.
- Atunci când variabila respectivă nu mai este utilă, spațul din memorie este eliberat, pentru a fi rezervat, dacă este cazul, pentru alte variabile.
- Mecanismul alocării dinamice este următorul:
 - ⇒ Se declară o variabilă pointer, s-o numim `p`, care permite memorarea unei adrese.
 - ⇒ Se alocă variabila dinamică prin operatorul `new` aplicat asupra unui tip, iar rezultatul este atribuit variabili `p`. În urma acestor operații variabila `p` reține adresa variabili alocată dinamic.
 - ⇒ Orice acces la variabila alocată dinamic se face prin intermediul variabilei `p`.



În C++, pentru alocarea dinamică se utilizează următorii operatori:

- ⇒ Operatorul `new` alocă spațiu în **HEAP** pentru o variabilă dinamică. După alocare, adresa variabili se atribuie lui `p`, unde `p` este o variabilă de tip pointer către tip:

`p=new tip`

- numărul de octeti alocati în **HEAP** este, evident, egal cu numărul de octeti ocupat de o variabilă de tipul respectiv.
- durata de viață a unei variabile alocate în **HEAP** este până la eliberarea spațiului ocupat (cu `delete`) sau până la sfârșitul executării programului.
- ⇒ Operatorul `delete` eliberează spațul rezervat pentru variabila și cărei adresă este reținută în `p`. După eliberare, continutul variabilei `p` este nedefinit.

`delete P`

Exemple:

1. Variabile de tip pointer către variabile de tip `int`. Variabila `adr1` poate reține adrese ale variabilelor de tip `int`.

```

...
int* adr1;
adr1=new int; // aloc spatiu in HEAP pentru o variabila de tip int
*adr1=7; //variabila alocata retine 7
cout<<*adr1; // tiparesc continutul variabilei
delete adr1; // eliberez spatiul
  
```

2. Variabile de tip pointer către variabile de tip `float`. Variabila `adresa`, poate reține adrese ale variabilelor de tip `float`.

```

float* adresa; // sau float* adresa=new float;
adresa=new float;
*adresa=7.65;
cout<<*adresa;
  
```

3. Variabile de tip pointer către variabile de tip `inreg`, care la rândul lor, sunt de tip `struct`. Variabila `adr_inr`, poate reține adrese ale variabilelor de tipul `inreg`.

```

#include <iostream.h>
struct inreg
{
    char nume[20], prenume[20];
    int varsta;
};
main()
{
    inreg* adr;
    adr=new inreg;
    cin>>adr->nume>>adr->prenume>>adr->varsta;
    cout<<adr->nume<<endl<<adr->prenume<<endl<<adr->varsta;
}
  
```

1.3 Alocarea dinamică a masivelor

Pentru început, vom prezenta, pe scurt, legătura între pointeri și masive.

⇒ Un tablou p-dimensional se declară astfel:

```
tip nume[n1] [n2] ... [np]
```

Exemple:

- float a[7][4][2]; am declarat un tablou cu 3 dimensiuni, unde tipul de bază este float.

- long b[9][7][8][5]; am declarat un tablou cu 4 dimensiuni, unde tipul de bază este long.

⇒ Numele tabloului p-dimensional de mai sus este pointer constant către un tablou p-1 dimensional de forma [n₂] ... [n_p], care are componentele de bază de același tip cu cele ale tabloului. Exemplile se referă la tablourile anterior prezentate.

- a este pointer constant către tablouri cu [4][2] componente de tip float.

- b este pointer constant către tablouri cu [7][8][5] componente de tip long.

⇒ Un pointer către un tablou k-dimensional cu [l₁][l₂] ... [l_k] componente de un anumit tip se declară astfel:

```
tip (*nume)[l1] [l2] ... [lk].
```

- variabilă de tip pointer, numită p, care poate reține pe a se declară:

```
float (*p)[4][2];
```

- variabilă de tip pointer, numită q, care poate reține pe b se declară:

```
long (*q)[7][8][5];
```

✓ De ce se folosesc parantezele rotunde? Operatorul de tip array ([]) are cea mai mare prioritate (mai mare decât operatorul *). Declarația:

```
float *p[7][8][5];
```

este interpretată ca masiv cu [7][8][5] componente de tip float*. În concluzie, este masiv cu componente de tip pointer și nu pointer către masive. Atenție! Aici se fac multe confuzii!

⇒ Întrucât numele unui masiv p-dimensional este pointer către un masiv p-1 dimensional, pentru a aloca dinamic un masiv se va utiliza un pointer către masive p-1 dimensionale (ultimele p-1 dimensiuni).

Exemple.

1. Alocăm în HEAP un vector cu 4 componente de tip int. Numele unui astfel de vector are tipul int* (pointer către int). Prin urmare, variabila a are tipul int*. După alocare, ea va conține adresa primului element al vectorului din HEAP. Din acest moment, pornind de la pointer (reținut în a) vectorul se adresează exact cum suntem obișnuiti.

```
int *a = new int[4];  
a[3]=2;
```

✓ Observați modul în care a fost trecut tipul în dreapta operatorului new. Practic, declararea tipului se face întocmai ca declararea masivului, însă numele a fost eliminat.

2. Declarăm în HEAP o matrice (masiv bidimensional) cu 3 linii și 5 coloane, cu elemente de tip double:

```
double (*a)[5]=new double [3][5];  
a[1][2]=7.8;
```

⇒ În cazul masivelor, trebuie atenție la eliberarea memoriei (cu delete). Trebuie să se ia în considerare tipul pointerului, așa cum rezultă din exemplele următoare:

a) Fie vectorul alocat în HEAP

```
int *a = new int[4];
```

Dacă încercăm dezalocarea sa prin:

```
delete a; îl dezalocăm prima componentă, pentru că pointerul este tip int*.
```

Corespond, dezalocarea se poate face prin: delete [4] a - eliberăm spațiul pentru toate componente (în acest caz avem 4). Observează că operatorul delete se poate utiliza și ca mai sus.

b) Fie matricea alocată în **HEAP**:

```
double (*a)[5]=new double [3][5];
```

Eliberarea spațiului ocupat de ea se face prin **delete [3] a;**

□ Programul următor citește și afisează o matrice. Nou este faptul că matricea este alocată în **HEAP**.

```
#include <iostream.h>
main()
{
    int m,n,i,j,(*adr)[10];
    adr=new int[10][10];
    cout<<"m=";cin>>m;
    cout<<"n=";cin>>n;
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            cin>>adr[i][j];
    for(i=0;i<m;i++)
    {
        for (j=0;j<n;j++)
            cout<<adr[i][j]<<" ";
        cout<<endl;
    }
}
```

✓ Alocarea în **HEAP** nu este lipsită de importanță. De obicei, matricele ocupă mult spațiu în memorie. Faptul că, de această dată, ele sunt rezervate în **HEAP**, conduce la economisirea memoriei din segmentul de date sau acela de stivă. În acest fel programul dispune de mai multă memorie, pentru că **HEAP-ul** este oricum rezervat.

□ Este cunoscut faptul că funcțiile nu pot întoarce masive. În schimb, o funcție poate întoarce un pointer către orice tip (**void***). Reamintim că unei variabile de tipul **void*** își poate atribui orice tip de pointer, dar atribuirea inversă se poate face doar prin utilizarea operatorului de conversie explicită. Programul următor citește două matrice și afisează suma lor. Matricele sunt rezervate în **HEAP**.

```
#include <iostream.h>
void* Cit_Mat(int m, int n)
{
    int i,j,(*adr1)[10]=new int[10][10];
    for(i=0;i<m;i++)
        for (j=0;j<n;j++) cin>>adr1[i][j];
    return adr1;
}
```

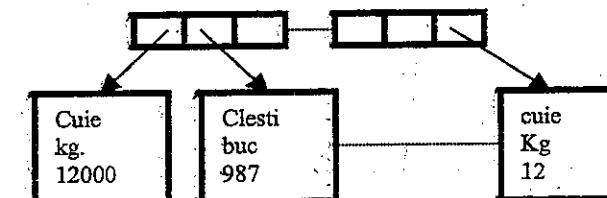
```
void Tip_Mat( int m,int n,int (*adr1)[10])
{
    int i,j;
    for(i=0;i<m;i++)
        for (j=0;j<n;j++)
            cout<<adr1[i][j]<<" ";
    cout<<endl;
}

void* Suma_Mat( int m, int n,int (*adr1)[10],int (*adr2)[10])
{
    int i,j,(*adr)[10]=new int[10][10];
    for (i=0;i<m;i++)
        for (j=0;j<n;j++) adr[i][j]=adr1[i][j]+adr2[i][j];
    return adr;
}

main()
{
    int m,n,i,j,(*adr)[10],(*adr1)[10],(*adr2)[10];
    cout<<"m=";cin>>m;
    cout<<"n=";cin>>n;
    adr1=(int(*)[10])Cit_Mat(m,n);
    adr2=(int(*)[10])Cit_Mat(m,n);
    adr=(int(*)[10])Suma_Mat(m,n,adr1,adr2);
    Tip_Mat(m,n,adr);
}
```

□ Pentru fiecare dintre cele **m** materiale, aflate într-o magazie, se cunoaște: denumirea materialului, unitatea de măsură (bucăți, kg. etc.) și cantitatea care se găseste în magazie, exprimată în unitatea de măsură înscrisă. Programul de mai jos citește informațiile și le organizează astfel:

- segmentul de date conține un vector cu **n** componente, unde fiecare componentă reține un pointer către înregistrarea cu datele citite pentru fiecare material;
- datele citite pentru fiecare material se organizează ca **struct**, alocat în **HEAP**.



```
#include <iostream.h>
struct material
{
    char denumire[20],unit_mas[5];
    float cantitate;
};
```

```

void Cit_Material(int i,material* v[100])
{
    v[i]=new material;
    cout<<"Denumire "; cin>>v[i]->denumire;
    cout<<"Unitate de masura "; cin>>v[i]->unit_mas;
    cout<<"Cantitate "; cin>>v[i]->cantitate;
}

void Tip_Material(int i,material* v[100])
{
    cout<<v[i]->denumire<<endl<< v[i]->unit_mas<<endl<<v[i]->cantitate;
}

main()
{
    int n,i;
    material* v[100];
    cout<<"n="; cin>>n;
    for (i=0;i<n;i++) Cit_Material(i,v);
    for (i=0;i<n;i++) Tip_Material(i,v);
}

```

Probleme propuse

1. O variabilă de tip pointer către tipul `int` poate memoria:

- a) un număr întreg;
- b) continutul unei variabile de tipul `int`;
- c) adresa unei variabile de tipul `int`;

2. Fie declarațiile următoare:

```
int *a1, *a2;
float *a3;
```

Care dintre atribuirile de mai jos este corectă?

- a) `a1=7.35;`
- b) `a3=7.35;`
- c) `a2=a3;`
- d) `a1=a2;`

3. Ce înțelegeți prin `HEAP`?

- a) un segment;
- b) tipul variabilelor care rețin adrese;
- c) o variabilă de sistem în care se pot retine date de orice tip.

4. Rolul operatorului `new` este:

- a) de a crea o adresă;
- b) de a aloca spațiu pentru o variabilă la o adresă dată;
- c) de a aloca spațiu în `HEAP` pentru o variabilă de tip pointer;
- d) de a aloca spațiu în `HEAP` pentru o variabilă de un tip oarecare.

5. Rolul operatorului `delete` este:

- a) de a șterge continutul unei variabile;
- b) de a șterge continutul unei variabile alocate în `HEAP`;
- c) de a elibera spațiu ocupat de o variabilă alocate în `HEAP`;
- d) de a elibera spațiu ocupat de o variabilă oarecare;
- e) de a șterge variabila la cărei adresă se găsește memorată în `HEAP`.

6. Ce se afisează în urma executării programului următor?

```
#include <iostream.h>
main()
{
    float *a1,*a2;
    a1=new float; *a1=3;
    a2=new float; *a2=4;
    a2=a1;
    cout<<*a2;
}
```

- a) 4;
- b) 3;
- c) eroare de sintaxă

7. Ce se afisează în urma executării programului următor, dacă se citesc, în această ordine, valorile 7 și 8?

```
#include <iostream.h>
main()
{
    int *a1,*a2,*man;
    a1=new int; cin>>*a1;
    a2=new int; cin>>a2;
    man=a2;
    a2=a1;
    a1=man;
    cout<<*a1<< " "<<*a2;
}
```

- a) 7 7
- b) 8 8
- c) 7 8
- d) 8 7

8. Identificați variabilele de tip pointer din următoarea secvență de declarări scriind numele fiecăreia:

```
typedef int adr;
adr x;
char v[10]; char* u;
float* m[5][5];
```

9. În condițiile de mai jos, stabiliți care dintre următoarele sevențe de instrucțiuni au ca efect interschimbarea valorilor variabilelor alocate la adresele **p** și **q**:

```
struct inreg
{ char nume[20], prenume[20];
  unsigned char varsta;
};

inreg *p,*q,*r;
```

- a) $x=p$; b) $*x=*p$; c) $x= \text{new } \text{inreg}$; d) $p->\text{nume}=q->\text{nume}$;
 $p=q$; $*p=*q$; $*x=*p$; $p->\text{prenume}=q->\text{prenume}$;
 $q=x$; $*q=*x$; $*p=*q$; $p->\text{varsta}=q->\text{varsta}$;
 $*q=*x$; $*q=*x$; $*q=*q$; $q->\text{nume}=q->\text{nume}$;
 $q->\text{prenume}=p->\text{prenume}$; $q->\text{varsta}=p->\text{varsta}$

10. Dacă **v** reprezintă un tablou care memorează adresele a 20 de vectori cu elemente numere reale memorate în **HEAP**, stabiliți care dintre următoarele instrucțiuni afisează primul element al celui de-al 10-lea vector. Stabiliti litera corespunzătoare răspunsului corect:

- a) `cout<<v[10][1];`
- b) `cout<<v[10,1];`
- c) `cout<<v[10]*[1];`
- d) `cout v[1]*[10];`
- e) `cout<<v[9][0];`
- f) `cout<<v*[9,0])`
- g) `cout<<v[9][1];`
- h) `cout<<v[0][10];`

11. Dacă **ad** reprezintă un tablou care memorează adresele a 20 de tablouri bidimensionale de căte 50×50 de elemente, tablouri alocate în **HEAP**, stabiliți care dintre următoarele instrucțiuni afisează al 30-lea element de pe linia a 5-a a celui de-al 7-lea tablou. Stabiliti litera corespunzătoare răspunsului corect:

- a) `cout<<ad*[7,5,30];`
- b) `cout<<ad*[6,4,29];`
- c) `cout<<ad[6][4][29];`
- d) `cout<<ad[7][30,5];`
- e) `cout<<ad[5,30)*[7];`
- f) `cout<<ad[4,29][6]);`
- g) `cout<<ad->[7]->[5,30];`

12. Se citesc **n**, număr natural și **m** numere reale care se memorează într-un vector alocat în **HEAP**. Se cere să se afișeze media aritmetică a numerelor reținute în vector.

13. Creați un tip numit **My_String**. O variabilă de acest tip conține un pointer (adresă) către un sir de caractere (tipul **String**). Sirul de caractere va fi reținut în **HEAP**. Scrieți, de asemenea, două funcții care permit citirea și scrierea conținutului unei variabile de acest tip. Care este avantajul utilizării tipului **My_String**?

14. Dați exemple de folosire pentru tipul **My_String** a subprogramelor care lucrează cu siruri de caractere.

15. Programul dv. trebuie să citească, să memoreze și să afișeze în siruri de tip **My_String**. Cum puteți realiza aceasta?

16. Să presupunem că avem două variabile de tip **My_String**, numite **S1** și **S2**. Fiecare dintre ele conține adresa unui sir, deja citit, care se află în **HEAP**. Ce credeți că se întâmplă la atribuirea **S1=S2**?

17. Cum credeți că o astfel de atribuire s-ar face în mod corect?

18. Scrieți o funcție prin care atribuirea să se efectueze corect.

19. Lucrare în echipă. Scrieți un modul de program care permite lucrul cu matrice rezervate în **HEAP**.

20. Testați dacă sevența de mai jos este permisă în C++:

```
int n;
cin>>n;
char* a=new char [n];
```

21*. Creați un tip numit **TStrings** care permite lucrul cu mai multe siruri de caractere. O variabilă de acest tip conține un pointer (adresă) către un vector care se găsește în **HEAP**. Fiecare componentă a vectorului reține o valoare de tip **My_String**. Cititi și tipăriți n siruri de caractere prin utilizarea acestui tip.

✓ Știți că în **Borland C++Builder** există tipul **TStrings**?

22*. Un fisier text numit **Lucrare.txt** conține pe fiecare linie cel mult 255 de caractere. Fisierul nu are mai mult de 10 linii. Se cere să se citească și să se afișeze sirurile reținute de fisier, utilizând o variabilă de tipul **TStrings**.

Indicații și răspunsuri

1) c); 2) d); 3) a); 4) d); 5) c); 6) b); 7) d); 8) a și componentele vectorului m; 9) a) și c); 10) a); 11) c).

13. Faptul că sirul poate conține spații impune utilizarea funcției `<cin.get()`. Ultimul caracter rămâne în buffer, motiv pentru care am utilizat `<cin.get()` fără parametri.

```
#include <iostream.h>
typedef char* My_String;

My_String Cit_Sir()
{
    My_String s1;
    s1=new char[256];
    cout<<endl<<"Introduceti sirul ";
    cin.get(s1,256);
    cin.get();
    return s1;
}

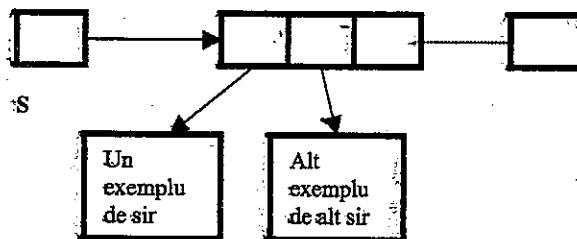
main()
{
    My_String s=Cit_Sir();
    cout<<s;
    My_String s1=Cit_Sir();
    cout<<s1;
}
```

15. Indicație. Se folosește un vector cu componente de tip `My_String`.

16. O astfel de atribuire este total neindicată. În urma ei, `s1` va reține adresa sirului `s2`. Aceasta înseamnă că atât `s1` cât și `s2` "poartează" către același sir. Sirul `s1` este pierdut, dar, și mai rău, rămâne alocat în `HEAP`, ocupând memoria inutil. Multă atenție la astfel de atribuiriri.

17. Chiar dacă cele două siruri sunt după atribuire identice, este corect ca fiecare să fie reținut în `HEAP`, întrucât este posibil ca în urma unor prelucrări ulterioare ele să nu mai coincidă. În concluzie, adresele nu se modifică, ci numai variabila din `Heap`.

21. Fie `s` o variabilă de tip `TString`. Dată schema de adresare cu ajutorul ei:



```
#include <iostream.h>
#include<string.h>
typedef char* My_String;
typedef My_String* TString;

My_String Cit_Sir()
{
    My_String s1;
    s1=new char[256];
    cout<<endl<<"Introduceti sirul ";
    cin.get(s1,256);
    cin.get();
    return s1;
}

TString Cit_TSir(int n)
{
    int i;
    TString s1=new My_String[10];
    for (i=0;i<n;i++) s1[i]=Cit_Sir();
    return s1;
}

main()
{
    int n,i;
    TString s;
    cout<<"n=">>n;
    cin.get();
    s=Cit_TSir(3);
    for (i=0;i<n;i++) cout<<endl<<s[i];
}
```

Capitolul 2

Liste liniare

2.1 Definția listelor

Definție. O listă liniară este o colecție de $n \geq 0$ noduri, x_1, x_2, \dots, x_n , aflate într-o relație de ordine. Astfel, x_1 este primul nod al listei, x_2 este al doilea nod al listei..., x_n este ultimul nod. Operatiile permise sunt:

- Accesul la oricare nod al listei în scopul citirii sau modificării informației continute de acesta.
- Adăugarea unui nod, indiferent de poziția pe care o ocupă în listă;
- Stergerea unui nod, indiferent de poziția pe care o ocupă în listă.
- Schimbarea poziției unui nod în cadrul listei.

2.2 Liste liniare alocate simplu înlăntuit

2.2.1 Prezentare generală

O listă liniară simplu înlăntuită este o structură de formă:



Semnificația notatiilor folosite este următoarea:

- $adr_1, adr_2, adr_3, \dots, adr_n$ reprezintă adresele celor n înregistrări;
- in_1, in_2, \dots, in_n reprezintă informațiile continute de noduri, de altă natură decât cele de adresă.
- 0 -are semnificația "nici o adresă" -elementul este ultimul în listă.

După cum observăm, fiecare nod, cu excepția ultimului, reține adresa nodului următor.

1. Accesul la un nod al listei se face prin parcurgerea nodurilor care îl preced. Aceasta necesită un efort de calcul.

2. Informațiile de adresă sunt prezente în cadrul fiecărui nod, deci ocupă memorie.
3. Avantajele alocării înlăntuite sunt date de faptul că operațiile de adăugare sau eliminare a unui nod se fac rapid.

2.2.2 Crearea și afisarea listelor

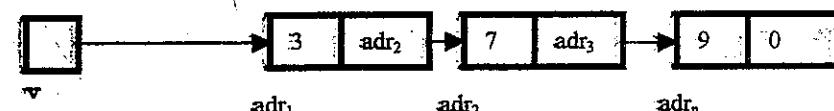
În capitolul precedent am învățat alocarea dinamică. Totuși, întreaga structură memorată în HEAP putea fi gestionată printr-un singur pointer, memorat în segmentul de date.

În cazul listelor, prin acel pointer se poate accesa numai primul element al listei. Apoi, pornind de la acesta se poate accesa al doilea element al listei și.m.d. De aici deducem că în cazul listelor, acest mecanism este mai complicat.

Crearea listelor.

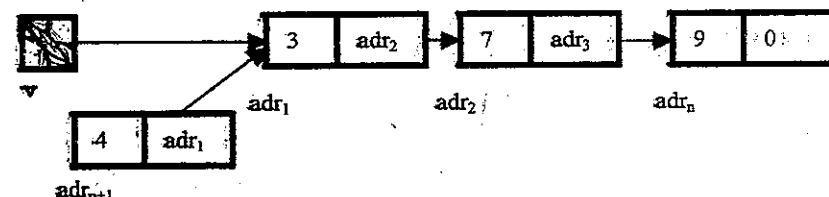
Initial, o variabilă v reține 0 (în C++ 0 are semnificația "nici o adresă")

Presupunem că, la un moment dat, lista este cea de mai jos, iar v reține adresa primului element (adr_1)

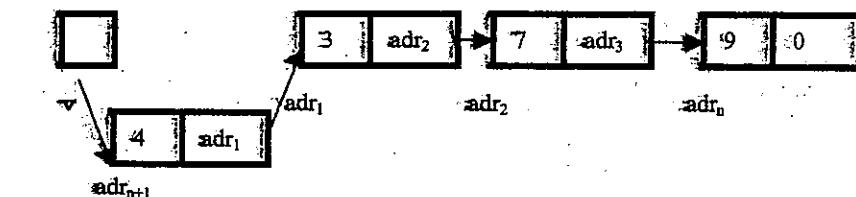


Dacă se citește un nou număr (de exemplu 4), atunci acesta se adaugă într-o înregistrare aflată la începutul listei, în următoarele etape:

- a) Se alocă spațiu pentru noua înregistrare, se completează câmpul numeric, iar adresa următoare este cea din v , deci a primului element al listei.



- b) Variabila v va memora adresa noii înregistrări:



```
#include <iostream.h>

struct Nod
{
    int info;
    Nod* adr_urm;
};

Nod* v;
int nr;

void Adaug(Nod*& v, int nr)
{
    Nod* c=new Nod;
    c->info=nr;
    c->adr_urm=v;
    v=c;
}

void Tip(Nod* v)
{
    Nod* c=v;
    while (c)
    {
        cout<<c->info<<endl;
        c=c->adr_urm;
    }
}

main()
{
    cout<<"numar=";cin>>nr;
    while (nr)
    {
        Adaug(v,nr);
        cout<<"numar=";cin>>nr;
    };
    Tip(v);
}

```

- ✓ Procedând astfel, lista va conține informațiile în ordinea inversă în care au fost introduse. Acest fapt nu prezintă importanță pentru majoritatea aplicatiilor.

Un alt algoritm de creare a listei, recursiv, este prezentat mai jos. De această dată lista cuprinde informațiile în ordinea în care acestea au fost introduse.

```
#include <iostream.h>

struct Nod
{
    int info;
    Nod* adr_urm;
};

Nod* Adaug()
{
    Nod* c;
    int nr;
    cout<<"numar ";cin>>nr;
    if (nr)
    {
        c=new(Nod);c->adr_urm=Adaug();
        c->info=nr; return c;
    }
    else
        return 0;
}

void Tip(Nod* v)
{
    Nod*c=v;
    while (c)
    {
        cout<<c->info<<endl;
        c=c->adr_urm;
    }
}

main()
{ v=Adaug();
  Tip(v);
}
```

Mai jos este prezentată o funcție recursivă care tipărește informațiile în ordine inversă față de modul în care se găsesc în listă.

```
void Tip_inv(Nod* v)
{
    if (v)
    {
        Tip_inv (v->adr_urm);
        cout<<v->info<<endl;
    }
}
```

2.2.3 Operării asupra unei liste liniare

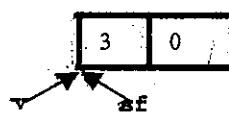
În acest paragraf prezentăm principalele operații care se pot face cu o listă liniară simplu înlățuită. Fiecare operație îi corespunde o funcție. Toate funcțiile se găsesc în modulul `Lista_v1.cpp`, listat la sfârșitul paragrafului.

- ✓ Trebuie să vă obișnuiați ca operațiile uzuale, sub formă de funcții, care lucrează cu structuri de date, să fie "depozitate" într-un modul separat. În acest fel programul care le utilizează va avea un text mai scurt, funcțiile le puteți folosi în mai multe programe și, în ultimul rând, vă apropiați de modul în care se procedează în practică.
 - ✓ În prezentarea operațiilor, vom folosi de multe ori desene. Retețeti: orice operație aveți de efectuat, faceți un mic desen. Vă ajută mult...
 - ✓ Orice listă va fi reținută prin două informații de adresă: a primului nod (`v`) și a ultimului nod (`sf`). Precizăm faptul că, în general, numai prima informație este indispensabilă. Pentru simplitatea funcțiilor și pentru rapiditatea executării vom reține și adresa ultimului nod.
1. Adăugarea Fiind dată o listă liniară se cere să se adauge la sfârșitul ei un nod, cu o anumită informație, în exemplele noastre, un număr întreg. Funcția care realizează această operație are antetul:

```
void Adaugare(Nod*& v, Nod*& sf, int val)
```

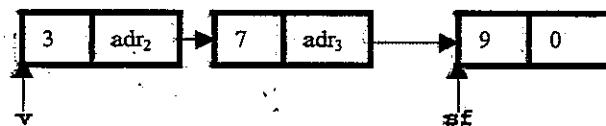
Se disting două cazuri:

- a) Lista este vidă - `v` retine 0. Să presupunem că vrem să adăugăm un nod cu informația 3. Se alocă în HEAP nodul respectiv, adresa sa va fi în `v`, și cum lista are un singur nod, adresa primului nod este și adresa ultimului, deci continutul lui `v` va coincide cu acela al lui `sf`.

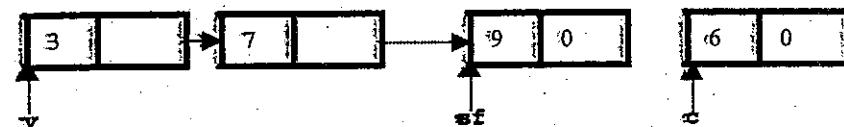


- b) Lista este nevidă.

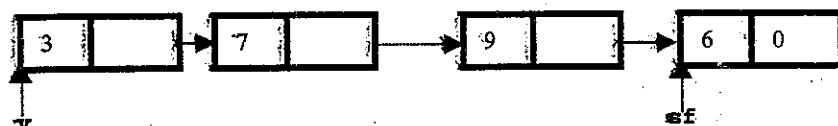
Fie lista:



Se adaugă un nod cu informația 6. Inițial se alocă spațiu pentru nod.



Câmpul de adresă al ultimului nod, cel care are adresa în `sf`, va reține adresa nodului nou creat, după care și `sf` va reține aceeași valoare.



- ✓ Dacă n-am fi utilizat variabila `sf` pentru a reține adresa ultimului nod, ar fi fost necesar să parcurem întreaga listă, pornind de la `v`, pentru a obține adresa ultimului.

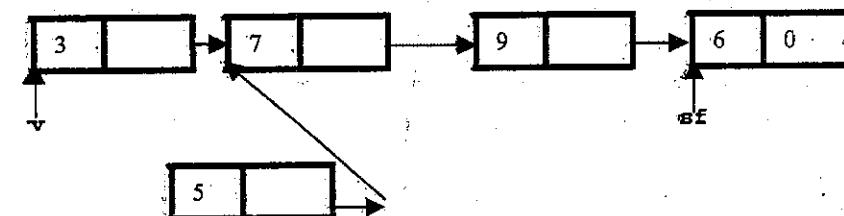
2. Inserarea unui nod, după un altul, de informație cunoscută. Operația este executată de funcția:

```
void Inserare_dupa(Nod* v, Nod*& sf, int val, int val1)
```

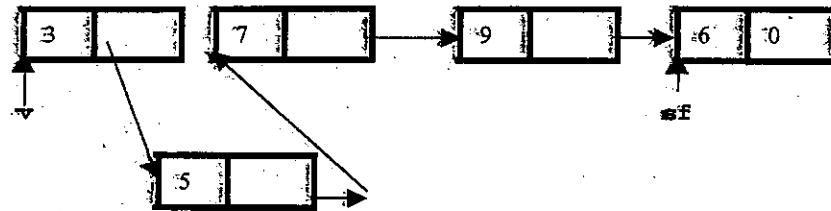
unde, `val` este informația căutată și `val1` este informația care se adaugă.

Exemplu. Fie lista din figura anterioară. Dorim să adăugăm după nodul cu informația 3, un altul, cu informația 5.

Inițial, se identifică nodul după care se face adăugarea. În cazul de față acesta este primul. Se alocă spațiu pentru noul nod. Se completează informația de adresă și anume adresa nodului care urmează după cel de cheie 3 și informația numerică 5.



Apoi, câmpul de adresă al nodului cu informația 3 va reține adresa nodului nou creat.



✓ Un caz aparte apare atunci când nodul de informație **val** este ultimul în distă. În acest caz **sf** va reține adresa nodului nou creat, pentru că acesta va fi ultimul.

3. Înserarea unui nod, înaintea altuia, de informație cunoscută. Întrucât operația este răsemănătoare celei precedente dăm numai funcția care realizează aceasta:

```
void Inserare_inainte(Nod*& v, int val, int val1)
```

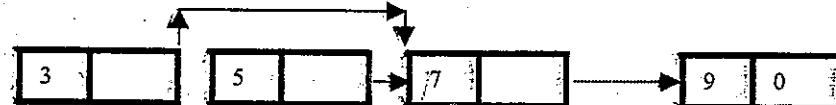
4. Stergerea unui nod de informație dată. Se realizează cu funcția următoare:

```
void Sterg(Nod*& v, Nod*& sf, int val)
```

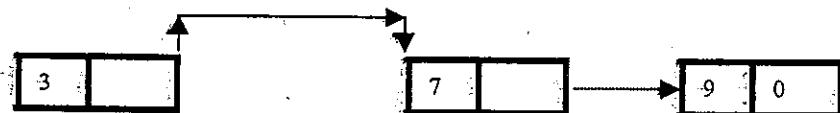
Algoritmul este diferit în funcție de poziția în distă a nodului care va fi sters - dacă este primul sau nu.

a) Nodul nu este primul:

Informația de adresă a nodului care îl precede trebuie să rețină adresa nodului următor:



Memoria ocupată de nodul care urmează să fie sters este eliberată.



b) Nodul este primul. Fie lista:



Variabila **v** va reține adresa celui de-al doilea nod:



Spațiul ocupat de primul nod va fi eliberat:



Mai jos este prezentat modulul care conține funcțiile:

```
#include <iostream.h>

struct Nod
{
    int info;
    Nod* adr_urm;
};

void Adaugare(Nod*& v, Nod*& sf, int val)
{
    Nod* c;
    if (v==0)
    {
        v=new(Nod); v->info=val; v->adr_urm=0;
        sf=v;
    }
    else
    {
        c=new(Nod); sf->adr_urm=c;
        c->info=val; c->adr_urm=0;
        sf=c;
    }
}

void Inserare_dupa(Nod* v, Nod*& sf, int val, int val1)
{
    Nod* c=v, *d;
    while (c->info!=val) c=c->adr_urm;
    d=new Nod; d->info=val1;
    d->adr_urm=c->adr_urm; c->adr_urm=d;
    if (d->adr_urm==0) sf=d;
}
```

```

void Inserare_inainte(Nod*& v, int val, int val1)
{
    Nod* c, *d;
    if (v->info==val)
    {
        d=new Nod; d->info=val1; d->adr_urm=v; v=d;
    }

    else
    {
        c=v;
        while (c->adr_urm->info!=val) c=c->adr_urm;
        d=new Nod; d->info=val1; d->adr_urm=c->adr_urm; c->adr_urm=d;
    }
}

void Sterg(Nod*& v, Nod*& sf, int val)
{
    Nod* c, *man;
    if (v->info==val)
    {
        man=v; v=v->adr_urm;
    }
    else
    {
        c=v;
        while (c->adr_urm->info!=val) c=c->adr_urm;
        man=c->adr_urm; c->adr_urm=man->adr_urm;
        if (man==sf) sf=c;
    }
    delete man;
}

void Listare(Nod* v)
{
    Nod* c=v;
    while (c)
    {
        cout<<c->info<<endl;
        c=c->adr_urm;
    }
    cout<<endl;
}

```

Programul următor are rolul de a testa modulul:

```

#include "lista_u.cpp"
Nod* v,*sf;
int i;
main()
{
    for (i=1;i<=10;i++) Adaugare(v,sf,i);
    Listare(v);
}

```

```

Inserare_dupa(v,sf,7,11);
Inserare_dupa(v,sf,10,12);
Inserare_dupa(v,sf,1,13);
Listare(v);
Inserare_inainte(v,13,14);
Inserare_inainte(v,1,15);
Listare(v);
Sterg(v,sf,15);
Sterg(v,sf,13);
Sterg(v,sf,12);
Listare(v);
}

```

2.2.4 Aplicații ale listelor liniare

2.2.4.1 Sortarea prin inserție

Se citesc de la tastatură n numere naturale. Se cere ca acestea să fie sortate crescător prin utilizarea sortării prin inserție.

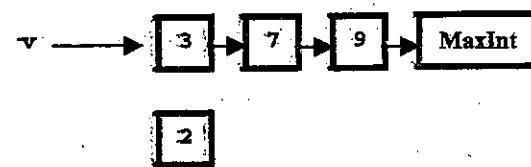
Această metodă de sortare a fost studiată în clasa a 9-a. Ideea de bază a metodei constă în a considera primele k valori sortate, urmând să inserăm valoarea $k+1$ în sirul deja sortat. Prin utilizarea listelor liniare înălțuite, inserția este mai simplă, întrucât nu necesită deplasarea componentelor, ca în cazul vectorilor.

Pentru simplificarea algoritmului, lista va conține valoarea maximă **MaxInt** alocată deja în listă. În acest fel algoritmul se simplifică, pentru că se pomenește deja de la o listă liniară nevidă. Evident, valoarea **MaxInt** nu va fi listată, atunci când se tipăresc numerele sortate.

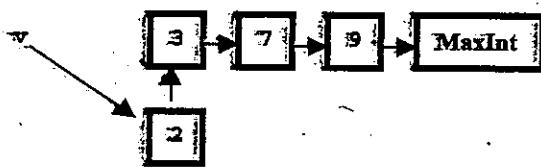
Problema se reduce la inserția unui număr într-o listă deja sortată. Mai întâi se alocă spațiu în **HEAP** pentru o valoare, apoi aceasta este citită. Se disting două cazuri:

1. Valoarea citită este mai mică decât prima valoare a listei. Aceasta înseamnă că ea este cea mai mică din listă și va fi introdusă prima în listă.

Exemplu. Fie lista următoare, și se citește 2.



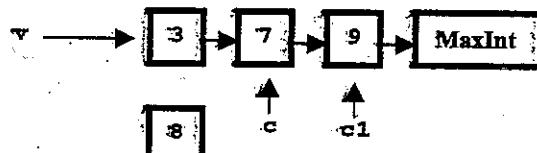
Noua înregistrare va conține adresa nodului 3, iar v va conține adresa noii înregistrări.



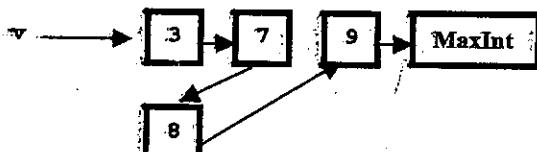
2. Valoarea citită nu este cea mai mică din listă. În mod sigur, nu este cea mai mare, pentru că am introdus MaxInt în listă. Dacă nu a fost îndeplinită condiția de la cazul 1, înseamnă că valoarea nu este nici cea mai mică. Aceasta înseamnă că ea va trebui introdusă în interiorul listei.

Va trebui să identificăm prima valoare mai mare decât valoarea citită. Întrucât până la adresa acestei valori avem nevoie de adresa precedentă (pentru a putea lega în listă nouul nod) vom "merge" cu doi pointeri, c și c1, unde c1 reține adresa înregistrării cu valoare mai mare decât înregistrarea citită, iar c adresa înregistrării precedente.

Exemplu. Fie lista următoare și se citește 18. Se identifică prima înregistrare care reține o valoare mai mare decât cea citită (în exemplu 9).



Noua valoare se inseră în listă.



```

#include <iostream.h>
const MaxInt=32000;
struct Nod
{
    int info;
    Nod* adr_urm;
};

int n,i;
Nod *v,*adr,*c,*c1;

```

```

main()
{
    cout<<"n=";cin>>n;
    v=new Nod; v->info=MaxInt; v->adr_urm=0;
    for (i=1;i<=n;i++)
    {
        adr=new Nod;
        cout<<"numar=";cin>>adr->info;
        if (adr->info<v->info)
            // primul din lista
        {
            adr->adr_urm=v;
            v=adr;
        }
        else // nu e primul din lista
        {
            c=v;
            c1=c->adr_urm;
            while (c1->info<adr->info)
            {
                c=c->adr_urm;
                c1=c->adr_urm;
            }
            c->adr_urm=adr;
            adr->adr_urm=c1;
        }
    }
    //tiparesc
    c=v;
    while (c->info!=MaxInt)
    {
        cout<<c->info<<endl;
        c=c->adr_urm;
    }
}

```

- ✓ Procedeul generării unei valori mai mari sau mai mici decât toate cele posibile este deseori folosit în informatică și vă veți mai întâni cu el în capitolul următor. Realizați cât de mult a simplificat algoritmul?
- ✓ La fiecare adăugare în listă se face o parcurgere a acesteia, deci algoritmul are complexitatea maximă $O(n^2)$.

2.2.4.2 Sortarea topologică*

Presupunem că dorim sortarea numerelor $1, 2, \dots, n$, numere care se găsesc într-o ordine oarecare, altă decât cea naturală. Pentru a afla relația în care se găsesc numerele, introducem un număr finit de perechi (i,j) . O astfel de pereche ne spune faptul că, în relația de ordine considerată, i se află înaintea lui j .

Exemplul 1. Fie $n=3$ și citim perechile $(3,1)$ și $(3,2)$. Numărul 3 se află înaintea lui 1 și 2 se află înaintea lui 2. Apar două soluții posibile: $3,1,2$ și $3,2,1$, întrucât nu avem nici o informație asupra relației dintre 1 și 2.

De aici tragem concluzia că o astfel de problemă poate avea mai multe soluții.

Exemplul 2: Fie $n=3$ și citim $(1,2)$, $(2,3)$, $(3,1)$. În acest caz nu avem soluție. Din primele două relații rezultă că ordinea ar fi $1,2,3$ iar relația $3,1$ contrazice această ordine.

În concluzie, problema poate avea sau nu soluție, iar dacă are poate este unică sau nu.

Algoritmul pe care îl prezentăm în continuare furnizează o singură soluție atunci când problema admite soluții. În caz contrar, specifică faptul că problema nu admite soluție.

Vom exemplifica funcționarea algoritmului pe exemplul următor: $n=4$ și se citesc perechile $(3,4)$, $(4,2)$, $(1,2)$, $(3,1)$.

Pentru fiecare număr între 1 și n trebuie să avem următoarele informații:

- numărul predecesorilor;
- lista succesorilor.

Pentru aceasta folosim doi vectori:

- **contor**, vector care reține numărul predecesorilor fiecarui k , $k \in \{1..n\}$;
- **a**, care reține adresele de început ale listelor de succesiuni a fiecarui element.

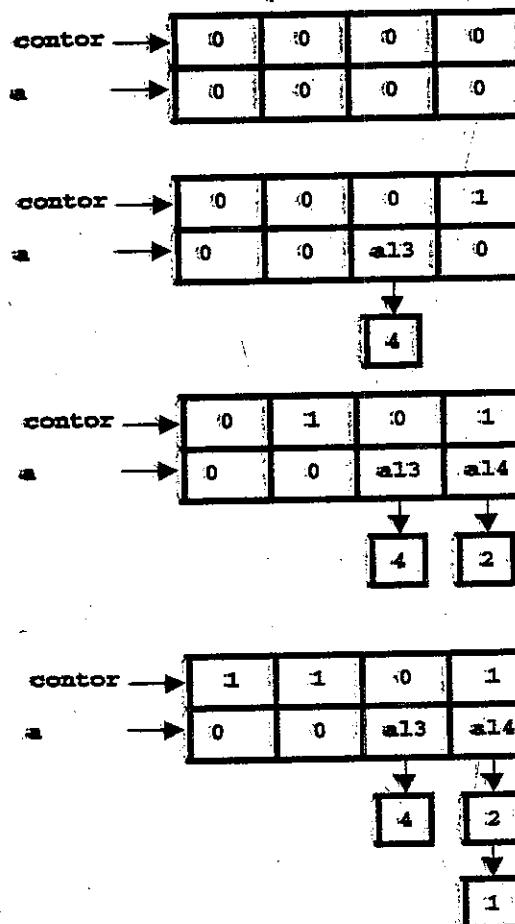
Pentru fiecare element există o listă simplu înălțuită a succesorilor săi.

Initial, în dreptul fiecărui element în vectorul **contor** se trece 0, iar în vectorul **a** se trece 0.

Citirea unei perechi (i,j) înseamnă efectuarea următoarelor operații:

- ⇒ incrementarea variabilei **contor(j)** (j are un predecesor, și anume i);
- ⇒ adăugarea lui j la lista succesorilor lui i .

Pentru exemplul dat, se procedează în felul următor:

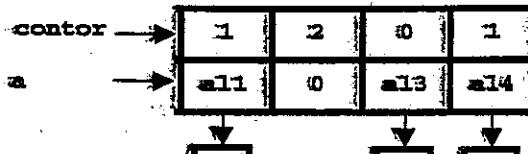


Valorile inițiale ale celor doi vectori.

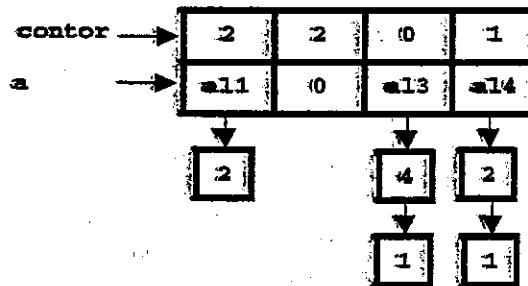
am citit $(3,4)$;
a13 – adresa listei 3;

am citit $(4,2)$;

am citit $(4,1)$;



am citit (1,2);

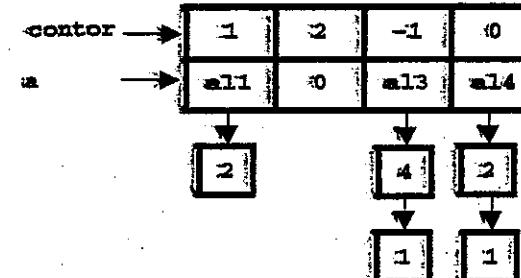


am citit (3,1);

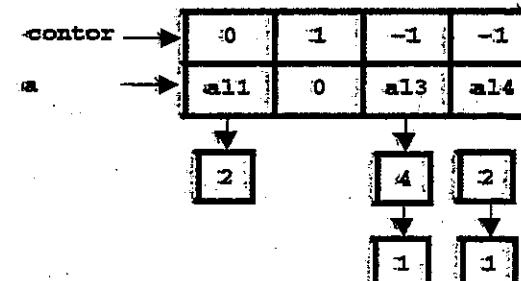
În continuare se procedează astfel:

- ⇒ toate elementele care au 0 în câmpul **contor** se rețin într-un vector **c**;
- ⇒ pentru fiecare element al vectorului **c** se procedează astfel:
 - ⇒ se tipărește;
 - ⇒ se marchează cu -1 câmpul său de contor;
 - ⇒ pentru toți succesorii săi (aflatî în lista succesorilor) se scade 1 din câmpul **contor** (este normal, întrucât aceștia au un predecesor mai puțin);
- ⇒ se reia algoritmul dacă nu este îndeplinită una din condițiile următoare:
 - au fost tipărite toate elementele, caz în care algoritmul se încheie cu succes;
 - nu avem nici un element cu 0 în câmpul **contor**, caz în care relațiile au fost incoerente.

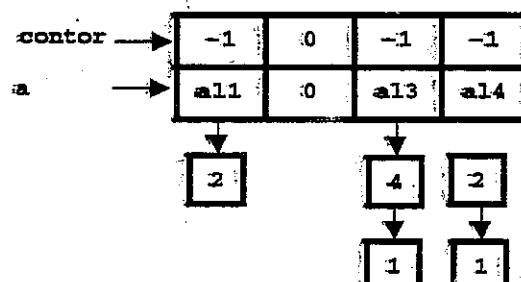
În continuare se procedează astfel:



Tipăresc 3, scad 1 din predecesorii lui 4 și 1, marchez cu -1 contorul lui 3.



Tipăresc 4, scad 1 din predecesorii lui 2 și 1, marchez cu -1 contorul lui 4.



Tipăresc 1, scad 1 din predecesorii lui 2, marchez cu -1 contorul lui 1.

- ✓ Algoritmul are multe aplicații, ca de exemplu:
 - ordonarea unor activități, atunci când ele sunt condiționate una de alta;
 - ordonarea unor termeni care se cer explicati, pentru că-i putea explica prin alții deja prezentați.
- ✓ Algoritmul are complexitatea $O(n^2)$. Odată sunt extrageri și la fiecare extragere se parurge lista succesorilor unui nod.

```

#include <iostream.h>
struct Nod
{
    int succ;
    Nod* urm;
};

int n,m,i,j,k,gasit;
int contor[100],c[100];
Nod* a[100];

void adaug(int i,int j)
{
    Nod *c,*d;
    contor[i]++;
    c=a[i];
    d=new Nod;
    d->urm=0; d->succ=j;
    if (c==0)
        a[i]=d;
    else
        { while (c->urm) c=c->urm;
          c->urm=d;
        }
}

void actual(int i)
{
    Nod* c=a[i];
    while (c)
        {contor[c->succ]--;
         c=c->urm;
        }
}

main()
{
    cout<<"n";cin>>n;
    for (i=1;i<=n;i++)
        {contor[i]=0;
         a[i]=0;
        }
    while (i)
        {cout<<"Tastati i, j="; cin>>i>>j;
         if (i) adaug(i,j);
        }
    m=n;
    do
    {k=1;
     gasit=0;
     for (i=1;i<=n;i++)
        if (contor[i]==0)
            {gasit=1;
             m--;
             c[k]=i;
             k++;
             contor[i]=-1;
            }
    }
}

```

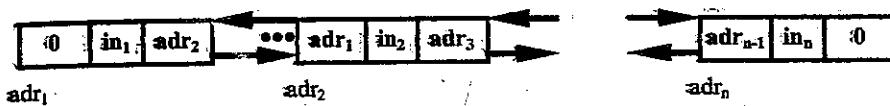
```

for (i=1;i<=k-1;i++)
{
    actual(c[i]);
    cout<<c[i]<<endl;
}
}
while (gasit == 0);
if (m) cout<<"relatii contradictorii";
else cout<<"Totalul e OK";
}

```

2.3 Liste liniare alocate dublu înălțuit

O listă alocată dublu înălțuit este o structură de date de forma:



✓ Avantajul utilizării listei alocate dublu înălțuit este dat de faptul că o astfel de listă poate fi parcursă în ambele sensuri.

Operatiile pe care le facem cu o listă dublu înălțuită sunt următoarele:

- 1) creare;
- 2) adăugare la dreapta;
- 3) adăugare la stânga;
- 4) adăugare în interiorul listei;
- 5) stergere din interiorul listei;
- 6) stergere la stânga listei;
- 7) stergere la dreapta listei;
- 8) listare de la stânga la dreapta;
- 9) listare de la dreapta la stânga.

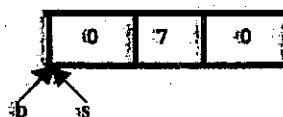
1) Creare

O listă dublu înălțuită se creează cu o singură înregistrare. Pentru a ajunge la numărul de înregistrări dorit, utilizăm funcții de adăugare la stânga sau la dreapta. În programul de față acest lucru este realizat de funcția **creare**. Această funcție realizează operațiile următoare:

- citirea informației numerice;
- alocarea de spațiu pentru înregistrare;
- completarea înregistrării cu informația numerică;
- completarea adreselor de legătură la stânga și la dreapta cu 0;

- variabilele tip referință și s vor căpăta valoarea adresei acestei prime înregistrări (s semnifică adresa înregistrării cea mai din stânga, și adresa ultimei înregistrări din dreapta).

Exemplu. Creăm lista cu un singur element.

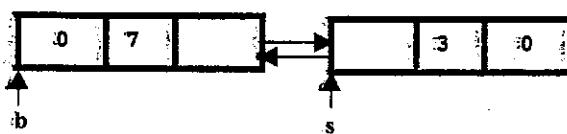


2) Adăugare la dreapta

Această operație este realizată de funcția **Addr**. Pentru adăugarea unei înregistrări se realizează următorii pași:

- citirea informației numerice;
- alocarea spațiului pentru înregistrare;
- completarea adresei la dreapta cu 0;
- completarea adresei din stânga cu adresa celei mai din dreapta înregistrări (retinute în variabila s);
- modificarea câmpului de adresă la dreapta a înregistrării din s cu adresa noii înregistrări;
- va lua valoarea noii înregistrări, deoarece aceasta va fi cea mai din dreapta.

Exemplu. Adăugăm la dreapta înregistrarea 3.



3) Adăugare la stânga

Această operație vă este propusă ca exercițiu.

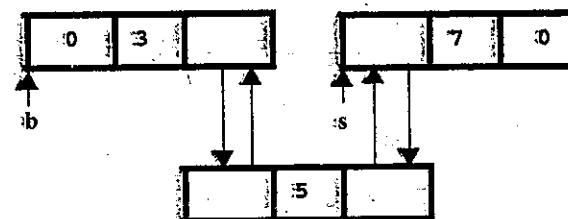
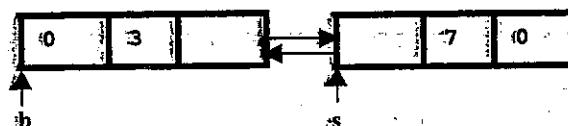
4) Adăugare în interiorul listei

Această operație este realizată de funcția **Includ**:

- parcurge lista de la stânga la dreapta căutând înregistrarea cu informația numerică m, în dreapta căreia urmează să introducem noua înregistrare;
- citește informația numerică;
- alocă spațiu pentru noua înregistrare;

- completează informația utilă;
- adresa stângă a noii înregistrării la valoarea adresei înregistrării de informație utilă m;
- adresa stângă a înregistrării care urma să acest moment înregistrării cu informația numerică m capătă valoarea adresei noii înregistrări;
- adresa dreaptă a noii înregistrării la valoarea adresei dreapta a înregistrării cu informația utilă m;
- adresa dreaptă a înregistrării cu informația numerică m la valoarea noii înregistrări.

Exemplu. În lista următoare adăugăm după înregistrarea 3 înregistrarea 5.



✓ Propunem ca exercițiu realizarea unei funcții de adăugare în interiorul listei la unei înregistrări la stânga înregistrării cu informația numerică m.

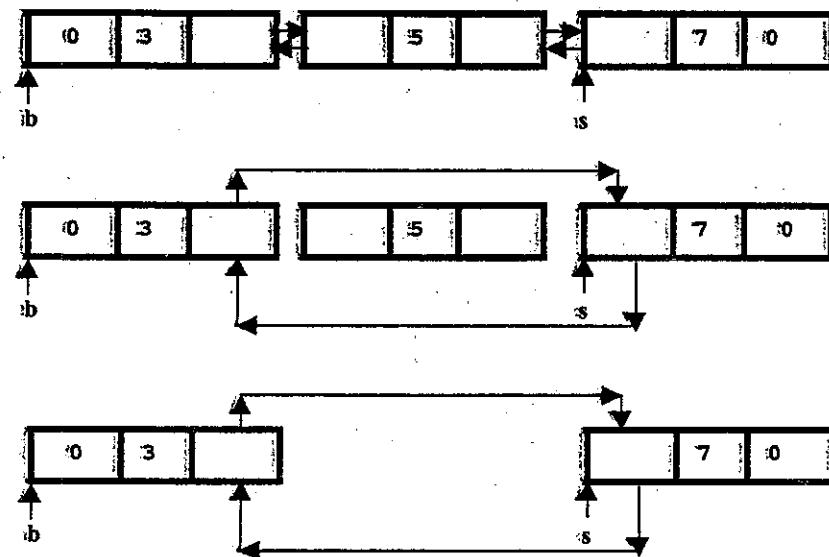
5) Stergere în interiorul listei

Această operație este realizată de funcția **Sterg**. Operațiile efectuate de această funcție sunt următoarele:

- se parcurge lista de la stânga la dreapta pentru a ne pozitiona pe înregistrarea care urmează a fi stearsă;
- câmpul de adresă dreapta al înregistrării care o precede pe această va lua valoarea câmpului de adresă dreapta al înregistrării care va fi stearsă;
- câmpul de adresă stânga al înregistrării care urmează înregistrării care va fi stearsă va lua valoarea câmpului de adresă stânga al înregistrării pe care o stergem;

- se eliberează spațiul de memorie rezervat înregistrării care se sterge.

Exemplu. În lista de mai jos se sterge elementul cu informația numerică 5.



6)-7) Stergere la stânga și la dreapta listei

Acste două operații sunt propuse ca exerciții.

8) Listare de la stânga la dreapta

Această operație este realizată de funcția **Listare** care realizează următoarele operații:

- pomenite din stânga listei;
- întâi timp cât nu s-a ajuns la capătul din dreapta al listei, se tipărește informația numerică și se trece la înregistrarea următoare.

9) Listare de la dreapta la stânga

Se propune ca exercițiu.

```
#include <iostream.h>

struct Nod
{
    Nod *as, *ad;
    int nr;
};

Nod *b,*s,*c;
int n,m,i;

void Creare (Nod*& b, Nod*& s)
{
    cout<<"n=";cin>>n;
    b=new Nod; b->nr=n;
    b->as=b->ad=0;
    s=b;
}

void Addr(Nod*& s)
{
    cout<<"n=";cin>>n;
    Nod* d=new Nod;
    d->nr=n;
    d->as=s;d->ad=0;
    s->ad=d;s=d;
}

void Listare(Nod*& b)
{
    Nod* d=b;
    while (d)
    {
        cout<<d->nr<<endl;
        d=d->ad;
    }
}

void Includ(int m, Nod* b)
{
    Nod *d=b, *e;
    while (d->nr!=m) d=d->ad;
    cout<<"n=";cin>>n;
    e=new Nod;
    e->nr=n;
    e->as=d;
    d->ad->as=e;
    e->ad=d->ad;
    d->ad=e;
}

void Sterg(int m, Nod* b)
{
    Nod* d=b;
    while (d->nr!=m) d=d->ad;
    d->as->ad=d->ad;
    d->ad->as=d->as;
    delete d;
}
```

```

main()
{
    cout<<"Creare lista cu o singura inregistrare "<<endl;
    Creare(b,s);
    cout<<"Cate inregistrari se adauga ?"; cin>>m;
    for (i=1;i<=m;i++) Addr(s);
    cout<<"Acum listez de la stanga la dreapta"<<endl;
    Listare(b);
    cout<<"Includem la dreapta o inregistrare "<<endl;
    cout<<"dupa care inregistrare se face includerea ?"; cin>>n;
    Includ(n,m,b);
    cout<<"Acum listez de la stanga la dreapta"<<endl;
    Listare(b);
    cout<<"Acum stergem o inregistrare din interior"<<endl;
    cout<<"Ce inregistrare se sterge ?";
    cin>>m;
    Sterg(m,b);
    cout<<"Acum listez de la stanga la dreapta"<<endl;
    Listare(b);
}

```

2.4 Stiva implementată ca listă liniară simplu înlățuită

- Stiva este o listă pentru care singurele operații permise sunt:
- Adăugarea unui element în stivă;
- Eliminarea, consultarea, sau modificarea ultimului element introdus în stivă.

Stiva funcționează pe principiul LIFO (Last In First Out) – "ultimul intrat primul ieșit".

Analizați programul următor, care creează o stivă prin utilizarea unei liste liniare simplu înlățuită. Adăugarea unui element în stivă se face cu funcția **PUSH**, eliminarea cu funcția **POP**. Vârful stivei este reținut de variabila **v**.

```

#include <iostream.h>
struct Nod
{
    int info;
    Nod* adr_inap;
};

Nod* v;
int n;

void Push (Nod*& v,int n)
{
    Nod* c;
    if (!v) { v= new Nod; v->info=n; v->adr_inap=0; }
    else
    { c= new Nod; c->info=n;c->adr_inap=v;
      v=c;
    }
}

```

```

void Pop (Nod*& v)
{
    Nod* c;
    if (!v) cout<<"stiva este vida";
    else
    { c=v;
      cout<<"am scos "<< c->info<< endl;
      v=v->adr_inap;
      delete c;
    }
}

main()
{
    push(v,1);Push(v,2);Push(v,3);
    Pop(v);Pop(v);Pop(v);Pop(v);
}

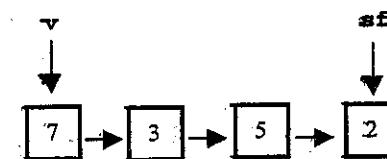
```

2.5 Coada implementată ca listă liniară simplu înlățuită

- O coadă este o listă pentru care toate inserările sunt făcute la unul din capete, toate stergerile (consultările, modificările) la celălalt capăt.

Coada funcționează pe principiul FIFO (First In First Out) – "primul intrat primul ieșit".

Alocarea dinamică înlățuită a cozi. O variabilă **v** va reține adresa elementului care urmează să fi scos (servit). O altă, numită **sf**, va reține adresa ultimului element introdus în coadă. Figura următoare prezintă o coadă în care primul element care urmează să fi scos are adresa în **v**, iar ultimul introdus are adresa în **sf**.



```

#include <iostream.h>
struct Nod
{
    int info;
    Nod* adr_sfum;
};

Nod* v,*sf;
int n;

```

```

void Punе (Nod*& v,Nod*& sf,int n)
{
    Nod* c;
    if (!v)
        {v=new Nod;v->info=n;v->adr_urm=0;
         sf=v;
        }
    else
        { c=new Nod;
         sf->adr_urm=c;
         c->info=n;
         c->adr_urm=0;
         sf=c;
        }
}

void Scoate (Nod*& v)
{
    Nod* c;
    if (!v)cout<<"coada este vida"<<endl;
    else
        { cout<<"Am scos "<<v->info<<endl;
         c=v;v=v->adr_urm;
         delete c;
        }
}

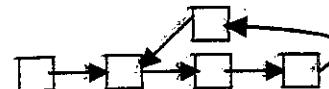
void Listare(Nod* v)
{
    Nod* c=v;
    while (c)
        { cout<<c->info<<" ";
         c=c->adr_urm;
        }
    cout<<endl;
}

main()
{
    Punе(v,sf,1);Punе(v,sf,2);Punе (v,sf,3); Listare (v);
    Scoate(v); Listare(v);
    Scoate(v); Listare(v);
    Scoate(v); Listare(v);
    Scoate(v); Listare(v);
}

```

Probleme propuse

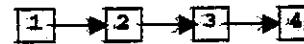
1. Asociați fiecărui tip de listă denumit în coloana din stânga desenul



corespunzător din coloana aflată în partea dreaptă a tabelului următor, scriind cifra asociată fiecărei litere:

A. listă liniară simplu înlățuită	1.
B. listă neliniară simplu înlățuită	2.
C. listă liniară dublu înlățuită	3.
D. listă neliniară dublu înlățuită	4.

2. Care dintre nodurile listei următoare (identificate prin numere între 1 și 4) este primul element al listei?

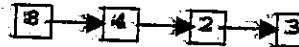


- a) 1 b) 4 c) 2 d) nu există un prim element

3. Dacă o listă formată din două noduri (identificate prin numerele 1 și 2) are proprietatea că elementul următor nodului 1 este nodul 2 și nu există un element următor nodului 1, atunci spunem că lista:

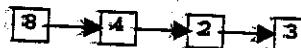
- a) este circulară
- b) este liniară simplu înlățuită
- c) nu este liniară
- d) este liniară dublu înlățuită

4. Stiind că adresa de început a listei reprezentate în desenul următor este memorată în variabila *p* și că fiecare nod al listei retine în câmpul *inf* numărul scris în desen și în câmpul *adr_urm* adresa elementului următor, stabiliți ce reprezintă expresia *p->adr_urm->adr_urm->nr*



- a) este o expresie încorrectă;
- b) valoarea memorată în nodul al treilea (valoarea 2);
- c) adresa elementului al treilea (elementului ce memorează valoarea 2);
- d) valoarea memorată în nodul al doilea (valoarea 4);
- e) adresa elementului al doilea (elementului ce memorează valoarea 4).

5. Stiind că adresa de început a listei reprezentate în desenul următor este memorată în variabila `p` și că fiecare nod al listei reține în câmpul `ini` numărul scris în desen și în câmpul `adr_urm` adresa elementului următor, stabiliți ce reprezintă expresia `p->adr_urm->nr->adr_urm`.



- a) este o expresie încorrectă;
- b) valoarea memorată în nodul al doilea (valoarea 4);
- c) adresa elementului al treilea (elementului ce memorează valoarea 2);
- d) valoarea memorată în nodul al treilea (valoarea 2).

6. Stiind că există o listă liniară simplu înlățuită nevidă, fiecare nod reținând în câmpul `ref` adresa elementului următor al listei, și stiind că variabilele `v` și `s` rețin adresa primului și respectiv adresa ultimului element al listei, explicați care este efectul instrucțiunii: `s->ref=v`.

7. Stiind că există o listă liniară simplu înlățuită cu cel puțin două noduri, fiecare nod reținând în câmpul `urm` adresa elementului următor al listei, și stiind că variabilele `ini` și `fin` rețin adresa primului și respectiv adresa ultimului element al listei, explicați care este efectul instrucțiunii: `ini->urm=fin`.

8. Scrieți o funcție care creează o listă liniară simplu înlățuită în care, fiecare nod, pe lângă informația de adresă, va contine o variabilă de tip `struct` care conține date referitoare la un elev. Funcția va returna adresa primului nod al listei. Datele se citesc de la tastatură.

- numele: `char[20]`;
- prenume: `char[20]`;
- un vector de numere reale cu 3 componente care rețin notele elevului.

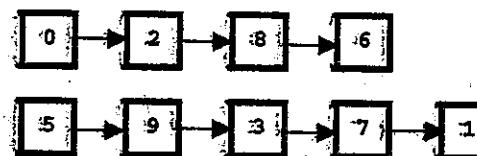
9. Scrieți o funcție care afisează pe monitor numele, prenumele și media generală a fiecărui elev din lista de mai sus. Funcția va primi ca parametru de intrare vârful listei.

10. Scrieți o funcție care returnează media generală a elevilor care se găsesc în listă.

11. Scrieți un program care creează și afisează o listă liniară simplu înlățuită. Fiecare nod al listei conține, pe lângă informația de adresă, un număr natural mai mic sau egal cu 100000. Numerele se găsesc, toate pe o linie, în ordine, separate prin un spatiu (blank) în fișierul text `lista.in`.

12. Scrieți un program care creează și afisează două liste liniare simplu înlățuite. Prima listă va conține, în ordinea citirii, numere pare, iar la doua va conține, în aceeași ordine, numere impare. Numerele se citesc din fișierul text `numere.in`. Ele se găsesc toate pe o linie și sunt separate prin cel puțin un spatiu. Exemplu: dacă fișierul text conține numerele:

0 2 5 9 8 3 6 7 1, atunci listele vor fi:



13. Scrieți o funcție care creează fișierul text `liste.out` cu informațile aflate în cele două liste de la problema anterioară. Prima linie va conține numerele din prima listă, și doua numerele din cea două listă. Exemplu: pentru liste din figura de mai sus fișierul va fi:

Linia 1 0 2 8 6
Linia 2 5 9 3 7 1

14. Scrieți o funcție care adaugă un nod la sfârșitul unei liste liniare simplu înlățuită. Fiecare nod al listei conține, pe lângă informația de adresă, un număr real. Funcția are ca parametri formali adresa primului element al listei și valoarea reală care se adaugă.

15. Scrieți o funcție care adaugă un nod la începutul unei liste liniare simplu înlățuită. Fiecare nod al listei conține, pe lângă informația de adresă, un număr real. Funcția are ca parametri formali adresa primului element al listei și valoarea reală care se adaugă. Ea returnează noua adresă de început a listei.

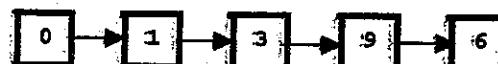
16. Se dă o listă liniară simplu înlățuită ale cărei noduri rețin, pe lângă informațiile de adresă, numere naturale cu o singură cifră. Lista are cel puțin un nod și cel mult 6 noduri. Se cere să se scrie o funcție care calculează și afisează valoarea întreagă obținută prin lipirea cifrelor memorate în listă în ordinea citirii. Funcția va primi ca parametru de intrare vârful listei. Exemplu: pentru lista de mai jos se afisează valoarea 956.



17. Prin operatia de *concatenare* a două liste liniare simplu înlățuite se obține o a treia listă liniară simplu înlățuită care conține, în ordine, nodurile primei liste, urmate de nodurile celei de-a doua liste. Să se scrie o funcție care concatenează două liste date prin adresele nodurilor de început. Funcția va returna adresa primului nod al noii liste.

18. Se citește un fișier text **cifre.in** care conține, pe o unică linie, numai numere naturale între 0 și 9. Numerele nu sunt separate prin spații. Se cere să se formeze o listă liniară simplu înlățuită în care fiecare nod reține o cifră. Exemplu:

Pentru : 01396 se obține lista:

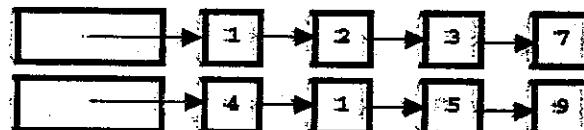


19. Să se scrie o funcție care primește ca parametru de intrare adresa unei liste liniare simplu înlățuită și are rolul de a inversa nodurile afiate pe prima și ultima poziție. Funcția va returna adresa primului nod al listei.

20. Scrieți o funcție care eliberează spațiul ocupat de o listă liniară simplu înlățuită.

21. Scrieți o funcție care memorează un tablou bidimensional cu m linii și n coloane ca m liste liniare simplu înlățuite, unde fiecare listă memorează, în ordine, elementele unei linii. Exemplu: pentru tabloul următor se obțin liste:

1 2 3 7
4 1 5 9

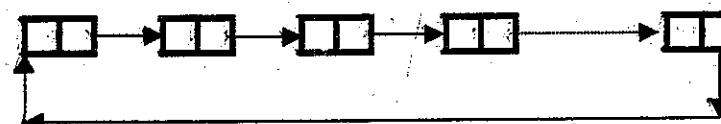


22. Se dă o listă liniară simplu înlățuită în care fiecare nod reține un caracter. Să se scrie o funcție care depistează dacă lista conține caractere distincte sau nu. Cel valoare trebuie să întoarcă o astfel de funcție?

23. Se dă o listă liniară simplu înlățuită în care fiecare nod reține o literă. Se cere să se scrie o funcție care depistează dacă cuvântul format prin alăturarea literelor citite este sau nu palindrom (se obține același rezultat dacă cuvântul se citește direct sau invers). De exemplu, lista următoare conține un cuvânt palindrom.



24. În cazul în care pentru o listă liniară simplu înlățuită câmpul de adresă al ultimului nod reține adresa primului nod se obține o listă circulară:



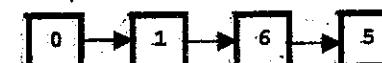
Creați o listă circulară în care fiecare nod reține un număr natural. De asemenea, scrieți subprograme de inserare și stergere la unui nod al listei create.

25. Se citește o permutare a numerelor 1, 2, ..., n. Se cere ca, prin utilizarea unei liste circulare, să se afișeze toate permutările circulare ale acesteia. Exemplu: Se citește 1 2 3. Se va afisa: 1 2 3, 2 3 1, 3 1 2.

26. Scrieți o funcție care transformă o listă liniară simplu înlățuită în una dublu înlățuită.

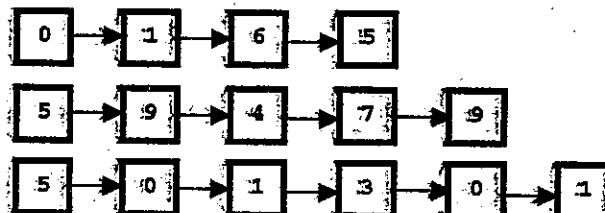
27. Lucrare în echipă. Realizați un asamblu de funcții care implementează stiva ca listă liniară simplu înlățuită. Dați exemple de utilizare. Cum credeți că trebuie să fie scrise subprogramele astfel încât acestea să lucreze cu minime modificări cu orice tip de date reținut de nodurile stivei?

28. După cum stii, nu se poate lucra în mod direct cu numere naturale oricăr de mari. Din acest motiv, vom memora un număr natural ca o listă liniară simplu înlățuită. De exemplu numărul 5610 se va memora sub forma de mai jos. Scrieți o funcție care citește de la tastatură cifrele unui număr natural, începând cu cifra cea mai semnificativă, și-l memorează ca listă liniară simplu înlățuită.

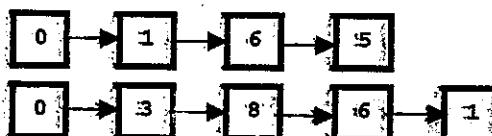


29*. Scrieți o funcție recursivă care primește ca parametru de intrare adresa unei liste liniare simplu înlățuită care reține un număr ca mai sus și afisează numărul. De exemplu, pentru lista de mai sus se afisează 5610.

30*. Scrieți o funcție care adună două numere naturale memorate ca mai sus și returnează adresa de început a numărului sumă, memorat ca listă. Exemplu: din primele liste rezultă a treia listă.



31*. Scrieți o funcție care calculează produsul dintre un număr natural memorat sub formă de listă și un altul, cu o singură cifră, transmis ca parametru. Funcția returnează adresa primului nod al listei care conține rezultatul. Exemplu: numărul de mai jos se înmulțeste cu 3 și rezultă:



32**. Scrieți o funcție care înmulțește două numere naturale memorate sub formă de liste și returnează adresa de început la listei rezultat.

33*. Lucrare în echipă. Scrieți un ansamblu de funcții memorate în modulul **NUMERE_MARI** care să ne ajute să lucrăm cu numere întregi, oricât de mari, memorate sub formă de liste. Utilizatorul poate efectua adunarea, scăderea, înmulțirea și împărțirea a două astfel de numere.

36*. Înzcrați modulul **NUMERE_MARI** cu funcții care să permită efectuarea comparațiilor între două numere: mai mare, mai mic, egal, mai mare sau egal, mai mic sau egal.

34*. Prin utilizarea modulului **NUMERE_MARI**, calculați maximul a n numere întregi, citite de la tastatură.

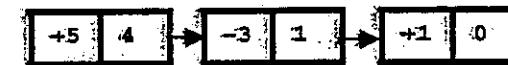
35*. Prin utilizarea modulului **NUMERE_MARI**, sortați crescător a n numere întregi, citite de la tastatură.

36*. Prin utilizarea modulului **NUMERE_MARI**, calculați π_1 , unde n este citit de la tastatură.

37*. Prin utilizarea modulului **NUMERE_MARI**, calculați: $\sum_{k=1}^n k!$

38**. Un polinom $P(x)$ se poate reprezenta printr-o listă liniară, ca în exemplul următor. Vă dată seama de avantajul unei astfel de memorări? Scrieți o funcție care citeste de la tastatură un polinom și-l memoră sub formă de listă liniară. Funcția returnează adresa listei.

$$P(x) = 5x^4 - 3x + 1.$$



39**. Scrieți o funcție care tipărește un polinom, memorat ca mai sus.

40**. Scrieți o funcție care adună două polinoame și returnează adresa polinomului sumă.

41**. Scrieți o funcție care înmulțește un polinom cu un monom, transmis ca parametru și returnează adresa polinomului rezultat.

42*. Scrieți o funcție care înmulțește două polinoame.

43**. Lucrare în echipă. Creați o unitate de program numită **Polinoame**, care conține subprograme care permit lucrul cu polinoame (adunare, scădere, înmulțire și împărțire).

44*. Prin utilizarea unității **Polinoame**, citiți un polinom P și calculați:

$$P^n(x)$$

Răspunsuri la testeile grilă. 1) Se realizează asocierile: A-3, B-1, C-2, D-4;
2) a); 3) b); 4) b); 5) a);

Capitolul 3

Elemente de teoria grafurilor

3.1 Grafuri orientate

3.1.1 Noțiuni introductive

Fie o mulțime finită $X = \{x_1, x_2, \dots, x_n\}$. Fie $\Gamma \subseteq X \times X$, unde $X \times X$ este produsul cartezian al mulțimii X cu ea însăși.

Definiție. Se numește graf orientat perechea ordonată $G = (X, \Gamma)$.

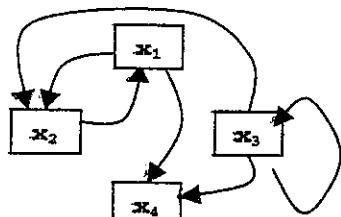
⇒ Elementele $x_i \in X$ se numesc noduri sau vârfuri.

⇒ Elementele mulțimii Γ se numesc arce. Un arc $(x_k, x_l) \in \Gamma$ se notează și cu $[x_k, x_l]$. Prin abuz de notație, vom scrie: $[x_k, x_l] \in \Gamma$.

Exemplu. Fie $X = \{x_1, x_2, x_3, x_4\}$. $X \times X = \{(x_1, x_1), (x_1, x_2), (x_1, x_3), (x_1, x_4), (x_2, x_1), (x_2, x_2), (x_2, x_3), (x_2, x_4), (x_3, x_1), (x_3, x_2), (x_3, x_3), (x_3, x_4), (x_4, x_1), (x_4, x_2), (x_4, x_3), (x_4, x_4)\}$.

Alegem $\Gamma = \{(x_1, x_2), (x_2, x_1), (x_1, x_4), (x_3, x_4), (x_3, x_2), (x_3, x_3)\}$. Evident, $\Gamma \subseteq X \times X$. Prin schimbarea notației obținem: $\Gamma = \{[x_1, x_2], [x_2, x_1], [x_1, x_4], [x_3, x_4], [x_3, x_2], [x_3, x_3]\}$.

În concluzie, graful este: $G = (X, \Gamma) = (\{x_1, x_2, x_3, x_4\}, \{[x_1, x_2], [x_2, x_1], [x_1, x_4], [x_3, x_4], [x_3, x_2], [x_3, x_3]\})$.

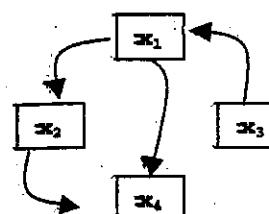


Un graf orientat poate fi reprezentat printr-un desen, astă cum rezultă din imaginea alăturată:

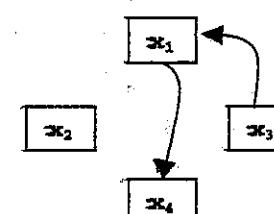
⇒ Dacă în graful $G = (X, \Gamma)$ $[x, y] \in \Gamma$ vom spune că x și y sunt adiacente, iar vârfurile x și y sunt incidente cu muchia $[x, y]$.

⇒ Dacă $\Gamma = \emptyset$ (mulțimea vidă), graful $G = (X, \Gamma)$ se numește graf nul și reprezentarea lui în plan se reduce la puncte izolate.

Definiție. Un graf parțial al unui graf orientat dat $G = (X, \Gamma)$ este un graf $G_1 = (X, \Gamma_1)$ unde $\Gamma_1 \subseteq \Gamma$.



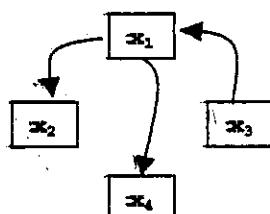
$G = (X, \Gamma)$



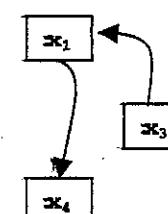
$G_1 = (X, \Gamma_1)$

Definiție. Un subgraf al unui graf orientat $G = (X, \Gamma)$ este un graf $H = (Y, \Gamma_1)$, unde $Y \subseteq X$, iar muchile din Γ_1 sunt toate muchile din Γ care au ambele extremități în mulțimea Y .

✓ un subgraf al unui graf G este graful G sau se obține din G prin suprimarea anumitor vârfuri și a tuturor muchiilor adiacente cu acestea.



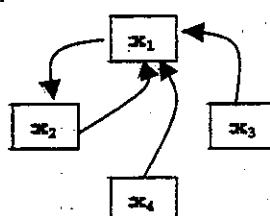
$G = (X_1, \Gamma)$



$G_1 = (Y, \Gamma_1)$

⇒ Gradul unui nod x ($d(x)$) este de două feluri:

- grad exterior $d^+(x)$ care reprezintă numărul arcelor care au ca extremitate inițială pe x ;
- grad interior $d^-(x)$ reprezintă numărul arcelor care au ca extremitate finală pe x .

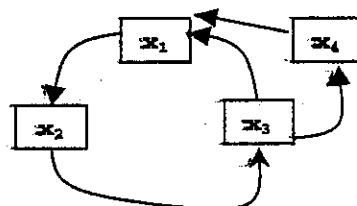


Exemplu. În graful orientat alăturat, avem:

$$\begin{aligned}d^+(x_1) &= 1, \quad d^-(x_1) = 3; \\d^+(x_2) &= 1, \quad d^-(x_2) = 1; \\d^+(x_3) &= 1, \quad d^-(x_3) = 0; \\d^+(x_4) &= 1, \quad d^-(x_4) = 0.\end{aligned}$$

Definiție. Un drum al unui graf orientat $\Sigma = \{x_0, x_1, \dots, x_p\}$ este o succesiune de vârfuri cu proprietatea că $[x_0, x_1] \in \Gamma$, $[x_1, x_2] \in \Gamma, \dots, [x_{p-1}, x_p] \in \Gamma$.

- ⇒ Vârfurile (nodurile) x_0 și x_p se numesc extremitățile drumului.
- ⇒ Numărul p se numește lungimea drumului.
- ⇒ Dacă vârfurile x_0, x_1, \dots, x_p sunt distincte, două căte două, drumul se numește elementar.
- ⇒ Un drum, Σ , pentru care $x_0 = x_p$, se numește circuit.
- ⇒ Dacă toate vârfurile circuitului, cu excepția primului și ultimului, sunt distincte, circuitul se numește elementar.



Exemplu: pentru graful orientat alăturat avem:
 $[x_1, x_2], [x_2, x_3], [x_3, x_4]$ este drum elementar de lungime 3.
 $[x_1, x_2], [x_2, x_3], [x_3, x_4], [x_4, x_1]$ este circuit elementar.

- ⇒ Un circuit elementar care trece prin toate nodurile grafului se numește circuit hamiltonian.

Exemplu. Pentru graful din figura anterioară avem:

$[x_1, x_2], [x_2, x_3], [x_3, x_4], [x_4, x_1]$ este un circuit hamiltonian.

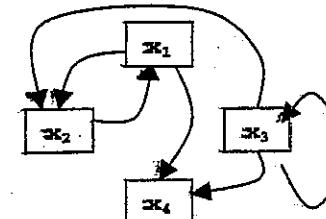
3.1.2 Metode de reprezentare în memorie a unui graf orientat

Există mai multe metode de reprezentare în memorie a unui graf orientat.

- ⇒ Un graf orientat cu n noduri poate fi memorat prin utilizarea unei matrice booleene cu n linii și n coloane, numită matricea de adiacență:

$$a_{i,j} = \begin{cases} 1, & \text{pentru } [i, j] \in \Gamma \\ 0, & \text{pentru } [i, j] \notin \Gamma \end{cases}$$

Exemplu: graful orientat următor se reprezintă astfel:



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- ✓ Dezavantajul memorării unui graf prin matricea de adiacență este dat de faptul că multe elemente ale matricei sunt nule, deci se consumă memorie înută.
- ✓ În aplicațiile care urmează vom căuta graful dintr-un fișier text, numit, de exemplu, **Graf.txt** în care prima linie va reține numărul de noduri, iar următoarele linii vor reține căte un arc.

Exemplu: graful de mai sus poate fi memorat în **Graf.txt** astfel:

4
1 2
1 4
2 1
3 2
3 3
3 4

- Pentru citirea fișierului și memorarea sa sub formă de matrice de adiacență vom utiliza funcția **Citire**.

```
#include <fstream.h>
void Citire(char Nume_fis[20], int A[50][50], int& n)
{
    ifstream f(Nume_fis, ios::in);
    int i, j;
    f >> n;
    while (f >> i >> j) A[i][j] = 1;
    f.close();
}
```

- ✓ Parametrii funcției **Citire** sunt:

- **Nume_fis** - numele fișierului text, transmis prin valoare;
- **A** - matricea de adiacență;
- **n** - numărul de noduri ale grafului orientat, transmis prin referință.

- ✓ Funcția de mai sus, este memorată în modulul numit **grafuri**.
- ✓ Iată, de exemplu, un program care citește matricea asociată și o afișează pe monitor:

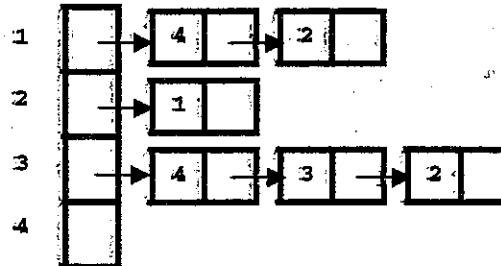
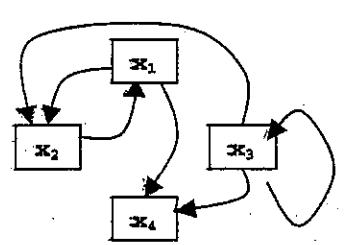
```

#include "grafuri.cpp"
int A[50][50],n,i,j;
main()
{
    Citire("Graf.txt",A,n);
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++) cout<<A[i][j]<<" ";
    cout<<endl;
}

```

- ⇒ Un graf orientat cu n noduri poate fi memorat prin utilizarea unor liste de adiacență. Pentru fiecare nod i , o listă dinară simplu înlățuită va retine toate nodurile j pentru care $(i,j) \in \Gamma$ - lista succesorilor.

Exemplu: graful orientat următor se reprezintă astfel:



- Funcția `Citire_1` are rolul de a citi graful din fisierul organizat ca anteriorul și de a-l memora sub această formă. În continuare sunt prezentate și tipurile de date utilizate. Toate sunt incluse în modulul `grafuri`.

```

struct Nod
{
    int nd;
    Nod* adr_urm;
};

```

- ✓ Iată, de exemplu, un program care creează, citește și afisează liste de adiacență:

```

void Citire_1( char Nume_fis[20], Nod* L[50], int& n)
{
    Nod* p;
    int i,j;
    fstream f(Nume_fis,ios::in);
    f>>n;
    for (i=1;i<=n;i++) L[i]=0;
    while (f>>i>>j)

```

```

    {
        p=new Nod;
        p->adr_urm=L[i];
        p->nd=j;
        L[i]=p;
    }
    f.close();
}

```

- ✓ Memorarea grafului prin intermediul listelor de adiacență prezintă avantajul că, în general, ocupă mai puțin spațiu. Totuși, accesul la informație, este puțin mai dificil.
- ✓ În practică se utilizează o formă sau alta de memorare a grafurilor, în funcție de algoritmul care lucrează asupra grafului.

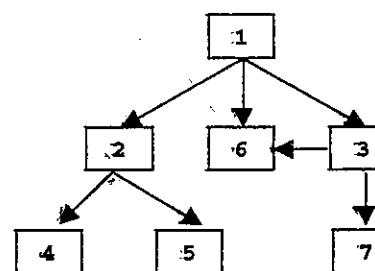
3.1.3 Parcursarea grafurilor orientate

Parcursarea grafurilor orientate se face, de regulă, în scopul prelucrării informațiilor asociate nodurilor. În acest paragraf nu ne interesează decât parcursarea propriu-zisă.

- ⇒ Practic, se pornește de la un anumit nod i și se vizitează toate nodurile j pentru care există drum de la i la j . Un nod este vizitat o singură dată.

Există două modalități generale de parcursere și anume: parcursarea în lătime (BF) și parcursarea în adâncime (DF). Acestea sunt tratate separat.

3.1.3.1 Parcursarea în lătime (BF- breadth first)



Priviți graful alăturat. Linile punctate prezintă o modalitate de parcursere a acestuia în lătime, pornind din nodul 1. După cum observați, ordinea de vizitare a nodurilor este:

1 3 6 2 7 5 4.

- ⇒ Parcursarea în lătime se face începând de la un anumit nod i , pe care îl considerăm parcurs.
- ⇒ Parcugem apoi toți descendenții săi - multimea nodurilor j pentru care $\exists (i,j) \in \Gamma$.
- ⇒ Parcugem apoi toți descendenții nodurilor parcurse la pasul anterior.

- ✓ Un nod este parcurs în singură dată. De exemplu, nodul 6 este descendental atât al nodului 1, cât și al nodul 3. El va fi listat în singură dată, ca descendant al nodului 1.
 - ✓ Există mai multe soluții ale unei astfel de parcurgeri, pentru că ordinea de parcurgere a descendentilor unui nod nu este impusă. Ea depinde și de modul în care a fost memorat graful. Exemplu: 1 2 6 3 4 5 7, este o altă soluție.
- ⇒ Parcurgerea BF se efectuează prin utilizarea structurii numită coadă, având grija ca un nod să fie vizitat în singură dată. Coada va fi alocată prin utilizarea unui vector.

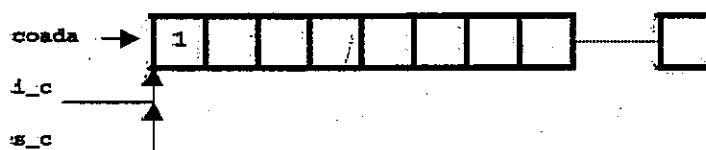
Algoritmul va utiliza următoarele notății:

- i_c - indicele componentei cozii care urmează să fie prelucrată;
- s_c - ultima componentă a cozii;
- $coada$ - vectorul care memorează coada propriu-zisă;
- s - vector boolean ce retine nodurile introduse în coadă.

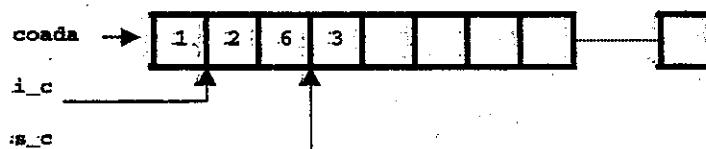
$$s[i] = \begin{cases} 0, & \text{nodul } i \text{ nu a fost introdus în coadă;} \\ 1, & \text{nodul } i \text{ a fost introdus în coadă;} \end{cases}$$

În continuare, prezentăm funcționarea algoritmului în ipoteza că nodul de pomire este 1.

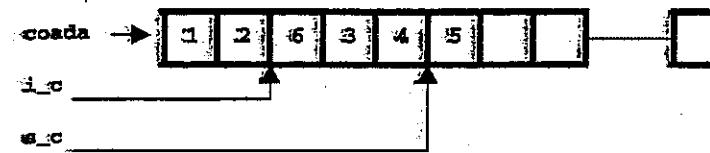
Nodul 1 se introduce în coadă:



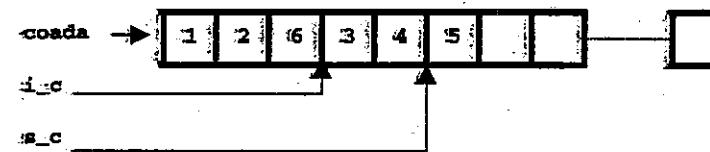
Se introduc în coadă toate nodurile j , pentru care $[1,j] \in \Gamma$.



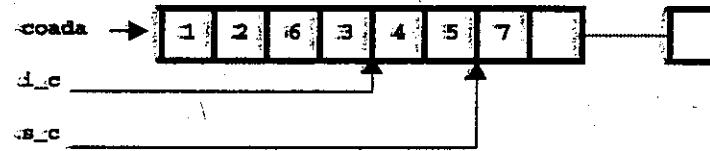
Se introduc în coadă toate nodurile j , pentru care $[2,j] \in \Gamma$.



Nu există noduri j , pentru care $[6,j] \in \Gamma$.



Există două noduri j , pentru care $[3,j] \in \Gamma$ și anume 6, care a fost deja vizitat și 7, care se introduce în coadă.



Nu mai există noduri j , pentru care $[4,j] \in \Gamma$, $[5,j] \in \Gamma$, $[7,j] \in \Gamma$.

Algoritmul se termină atunci când $i_c > s_c$, adică au fost prelucrate toate nodurile din coadă. Evident, acum se poate tipări vectorul $coada$, făcând abstracție de faptul că acesta a fost prelucrat ca o coadă.

În continuare vom prezenta programele care parcurg în lățime grafurile orientate. Memorarea grafurilor se face ca în paragraful anterior, prin utilizarea listelor și a matricei de adiacență.

1. Cazul grafului memorat prin liste de adiacență - recursiv și nerecursiv.

```
#include "grafuri.cpp"
```

```
Nod* L[50];
int coada[50], s[50], i_c, sf_c, i, n;

// funcția de parcurgere BF_R sau BF.
main()
{
    Citire_1("Graf.txt", L, n);
    i_c=1; sf_c=1; coada[i_c]=1; s[1]=1;
```

```

bf_x();
for (i=1;i<=sf_c;i++) cout<<coada[i]<<" ";
}

```

```

void bf_x()
{
    Nod* p;
    if (i_c<=sf_c)
    {
        p=L[coada[i_c]];
        while (p)
        {
            if (s[p->nd]==0)
            {
                sf_c++;
                coada[sf_c]=p->nd;
                s[p->nd]=1;
            }
            p=p->adr_urm;
        }
        i_c++;
        bf_x();
    }
}

```

2. Cazul grafului memorat prin matricea de adiacență - varianta recursivă.

```

#include "grafuri.cpp"
int coada[50],s[50],A[50][50],i_c,sf_c,i,n;
void bf_r1()
{
    int k;
    if (i_c<=sf_c)
    {
        for (k=1;k<=n;k++)
            if ((A[coada[i_c]][k]==1) && (s[k]==0) )
            {
                sf_c++;
                coada[sf_c]=k;
                s[k]=1;
            }
        i_c++;
        bf_r1();
    }
}

main()
{
    Citire("graf.txt",A,n);
    i_c=sf_c=1;
    coada[i_c]=s[1]=1;
    bf_r1();
    for(i=1;i<=sf_c;i++)cout<<coada[i]<<" ";
}

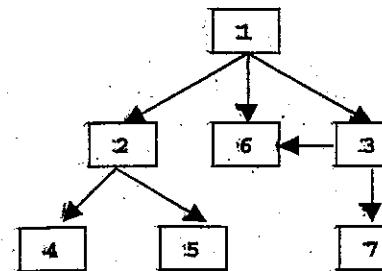
```

```

void bf()
{
    Nod* p;
    while (i_c<=sf_c)
    {
        p=L[coada[i_c]];
        while (p)
        {
            if (s[p->nd]==0)
            {
                sf_c++;
                coada[sf_c]=p->nd;
                s[p->nd]=1;
            }
            p=p->adr_urm;
        }
        i_c++;
    }
}

```

3.1.3.2 Parcursul în adâncime (DF-depth first)



Priviti graful alăturat. El este parcurs în adâncime pornind de la nodul 1. Si aici există mai multe soluții de parcurs, de exemplu:

1 2 4 5 3 6 7
1 3 7 6 2 5 4.

⇒ Parcursul în adâncime se face începând de la un anumit nod și.

⇒ După parcursul unui nod se trece la primul dintre descendenții săi, neparcursi încă.

Mai jos, puteți observa programele pentru parcursarea în adâncime realizată recursiv pornind de la graful memorat prin matricea de adiacență și liste de adiacență.

```

#include "grafuri.cpp"
int *s[50],A[50][50],n;
void df_r(int nod)
{
    int k;
    cout<<nod<<" ";
    s[nod]=1;
    for (k=1;k<=n;k++)
        if ((A[nod][k]==1)&& (s[k]==0))
            df_r(k);
}

main()
{
    Citire("Graf.txt",A,n);
    df_r(1);
}

```

```

#include "grafuri.cpp"
int *s[50],n;
Nod *L[50];
void df_r1(int nod)
{
    Nod* p;
    cout<<nod<<" ";p=L[nod];
    s[nod]=1;
    while (p)
    {
        if (s[p->nd]==0)
            df_r1(p->nd);
        p=p->adr_urm;
    }
}

main()
{
    Citire_l("graf.txt",L,n);
    df_r1(1);
}

```

- ✓ Parcurgerea în adâncime se face prin utilizarea stivei. Dar aceasta se realizează în mod implicit în variantele recursive, pentru că, după cum stim, recursivitatea lucrează pe principiul stivei.

3.1.3.3 Estimarea timpului necesar parcurgerii grafurilor.

Fie un graf orientat cu n noduri. Notăm numărul arcelor grafului cu m . Care este numărul maxim de arce într-un graf orientat? Vezi definitia grafului orientat. Produsul cartezian $X \times X$ are n^2 elemente, unde fiecare element este un arc. În concluzie, numărul maxim de arce este n^2 .

- ✓ La același rezultat se ajunge dacă se consideră că oricare două noduri sunt unite prin două arce și fiecare nod să conțină un arc de forma $[i, i]$. Atunci numărul maxim de arce este:

$$n + 2C_n^2 = n + 2 \frac{n(n-1)}{2} = n + n^2 - n = n^2.$$

De regulă, m este cu mult mai mic decât n^2 . Din acest motiv, atunci când ne referim la estimarea timpului vom utiliza pe m .

⇒ Oricare dintre cele două metode de parcurgere a grafului (BF, DF) selectează nodurile care se parcurg prin intermediul arcelor.

⇒ În oricare din metodele de parcurgere ale grafului un arc este selectat o singură dată.

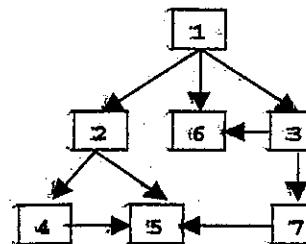
Din acest motiv, neglijând celelalte operații, parcurgerea grafurilor se realizează în $O(m)$. În concluzie, parcurgerea grafurilor se face în timp polinomial, adică cel mult $O(n^3)$.

3.1.4 Matricea drumurilor

Fie graful orientat $G = (X, T)$, unde X are n elemente - graful are n noduri. Lui G îi se asociază următoarea matrice, numită matricea drumurilor cu n linii și n coloane, notată M .

$$m_{i,j} = \begin{cases} 1, & \text{daca există drum de la } i \text{ la } j, \\ 0, & \text{daca nu există drum de la } i \text{ la } j, \text{ sau } i = j. \end{cases}$$

Grafului de mai jos îi se asociază următoarea matrice a drumurilor:



0	1	1	1	1	1	1
0	10	0	1	1	0	0
0	0	0	0	1	1	1
0	0	0	0	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	1	0	0

- Fieind dat un graf orientat, memorat prin intermediul listelor de adiacență, se cere să se creeze și să se calculeze matricea drumurilor.

1. Să observăm că în urma parcurgerii DF a grafului, pomind de la un anumit vîrf - să-l numim baza - vectorul s reține 1 pentru toate nodurile la care s-a ajuns. Cu alte cuvinte vectorul s , reține 1 pentru toate nodurile pentru care există drum de la nodul de pornire la ele. În acest fel obținem o linie a matricei.

2. Apelul funcției DF pentru fiecare din cele n noduri, permite obținerea matricei drumurilor.

```

#include "grafuri.cpp"

int s[50], n, i, j, M_dr[50][50];
Nod* L[50];

void df(int baza, int nod)
{
    Nod* p;
    if (baza != nod) M_dr[baza][nod] = 1;
    p = L[nod]; s[nod] = 1;
    while (p)
    {
        if (s[p->nd] == 0) df(baza, p->nd);
        p = p->adr_urm;
    }
}

main()
{
    Citire_1("Graf.txt", L, n);
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++) M[j][i] = 0;
        df(i, i);
    }
}
  
```

```

for (i=1; i<=n; i++)
{
    for (j=1; j<=n; j++)
        cout << M_d[1][i][j] << " ";
    cout << endl;
}

```

- ✓ Întrucât funcția este apelată de n ori și timpul estimat este de $O(n)$, rezultă că timpul total estimat este $O(n^2)$, sau, în cazul cel mai favorabil, $O(n^3)$.

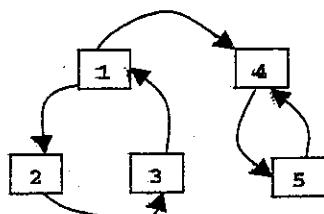
3.1.5 Componente tare conexe

Fie $G=(X, \Gamma)$ un graf orientat. Fie $G_1=(X_1, \Gamma_1)$ un subgraf al lui G .

Definție. $G_1=(X_1, \Gamma_1)$ este o componentă tare conexă dacă:

1. $\forall x, y \in X_1, \exists$ drum de la x la y și drum de la y la x .
2. Nu există un alt subgraf al lui G , $G_2=(X_2, \Gamma_2)$ cu $X_1 \subset X_2$ care îndeplinește condiția 1.

Exemplu.



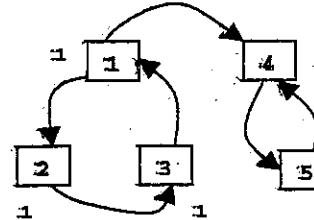
Graful alăturat are două componente tare conexe.

- subgraful care conține vârfurile: 1, 2, 3
- subgraful care conține vârfurile: 4, 5

Definție. Un graf $G=(X, \Gamma)$ este tare conex dacă $\forall x, y \in X, \exists$ drum de la x la y și drum de la y la x .

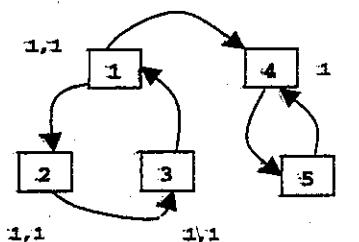
- ✓ Cu alte cuvinte, un graf este tare conex dacă admite o singură componentă tare conexă.
- Fie un graf orientat $G=(X, \Gamma)$, memorat prin intermediul matricei de adiacență și un nod i . Se cere să se determine componenta conexă căreia îl aparține i .

1. Vom numi succesiuni ai nodului i , toate nodurile j , pentru care există drum de la i la j , la care se adaugă i .



Exemplu. Orice nod al grafului alăturat este succesor al nodului 1, pentru că din 1 există drum către oricare alt nod al grafului. Vom marca cu 1 succesiunii nodului 1.

2. Fie i un nod al grafului. Vom numi predecesori ai nodului i , toate nodurile j , pentru care există drum de la j la i , la care se adaugă i .



Exemplu. Nodurile 1, 2, 3 sunt predecesori ai nodului 1.

Vom marca cu 1 predecesorii nodului 1.

3. Intersecția dintre multimea succesiunilor unui nod și cea a predecesorilor unui nod este multimea vârfurilor componente tare conexă care conține nodul respectiv. Nodul i aparține unei componente tare conexe. Pentru orice alt nod j din componentă există drum de la i la j și de la j la i .

Pentru graful de mai sus nodurile 1, 2, 3 constituie componenta conexă care conține nodul respectiv. Aceste noduri sunt marcate de două ori cu 1.

4. Succesiuni unui nod i se determină, de exemplu, prin parcurgerea în adâncime a grafului, pornind de la nodul i , marcându-i succesiunii în vectorul suc . Inițial, $suc[i]$ va fi marcat cu 1.

5. Predecesori unui nod i se determină, tot printr-o parcurgere în adâncime a grafului, pornind de la nodul i și marcându-i predecesorii. Inițial, $pred[i]$ va fi marcat cu 1.

```

#include "grafuri.cpp"
int suc[50], pred[50], A[50][50], n, i, j;

```

```

void df_x1(int nod)
{
    int k;
    suc[nod]=i;
    for (k=1;k<=n;k++)
        if ( (A[nod][k]==1)&&(suc[k]==0) )
            df_x1(k);
}

void df_x2(int nod)
{
    int k;
    pred[nod]=i;
    for (k=1;k<=n;k++)
        if ( (A[k][nod]==1)&&(pred[k]==0) )
            df_x2(k);
}

main()
{
    Citire("Graf.txt",A,n);
    cout<<"Introduceti nodul de pornire ";cin>>i;
    suc[i]=pred[i]=i;
    df_x1(i);df_x2(i);
    for (j=1;j<=n;j++)
        if ( (suc[j]==pred[j])&& (suc[j]==i) )
            cout<<j<<" ";
}

```

- Fie un graf orientat $G=(X,T)$, memorat prin intermediul matricei de adiacență. Se cere să se descompună graful în componente tare conexe.

Variabila `nrc` va reține numărul curent al componentei tare conexe care urmează să fie identificată. Doi vectori, `suc` și `pred` marchează succesorii și predecesorii unui nod. Marcarea propriu-zisă se face prin memorarea numărului componente tare conexe. Prezentăm algoritmul pentru graful anterior.

Selectăm primul nod - în exemplu 1. Variabila `nrc` va reține 1.

<code>suc</code>	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0
0	0	0	0	0		
<code>pred</code>	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0
0	0	0	0	0		

Apelăm funcțiile `df_x1` și `df_x2` pentru marcarea succesorilor și predecesorilor unui nod (prin conținutul variabilei `nrc`).

<code>suc</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table>	1	1	1	1	1
1	1	1	1	1		
<code>pred</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	1	1	0	0
1	1	1	0	0		

În `suc` și `pred` componentele ai căror conținut diferă sunt anulate.

<code>suc</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	1	1	0	0
1	1	1	0	0		
<code>pred</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	1	1	0	0
1	1	1	0	0		

Variabila `nrc` va reține 2.

Pentru primul nod care nu aparține unei componente tare conexe se parcurge din nou graful în adâncime. În exemplu, acest nod este 4.

<code>suc</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td></tr> </table>	1	1	1	2	2
1	1	1	2	2		
<code>pred</code>	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td></tr> </table>	1	1	1	2	2
1	1	1	2	2		

Întrucât nu mai există noduri neparcuse se listează componentele conexe:

Componenta 1 : 1 2 3.

Componenta 2 : 4.

```

#include "grafuri.cpp"
int suc[50],pred[50],A[50][50],n,nrc,i,j;

void df_x1(int nod)
{
    int k;
    suc[nod]=nrc;
    for (k=1;k<=n;k++)
        if ( (A[nod][k]==1)&&(suc[k]==0) )
            df_x1(k);
}

void df_x2(int nod)
{
    int k;
    pred[nod]=nrc;
    for (k=1;k<=n;k++)
        if ( (A[k][nod]==1)&&(pred[k]==0) )
            df_x2(k);
}

```

```

main()
{
    Citire("Graf.txt", A, n);
    nrc=1;
    for (i=1; i<=n; i++)
        if (suc[i]==0)
    {
        suc[i]=nrc;
        df_x1(i); df_x2(i);
        for (j=1; j<=n; j++)
            if (suc[j]!=pred[j]) suc[j]=pred[j]=0;
        nrc++;
    }
    for (i=1; i<nrc; i++)
    {
        cout<<"Componenta "<<i<<endl;
        for (j=1; j<=n; j++)
            if (suc[j]==i) cout<<j<<" ";
        cout<<endl;
    }
}

```

3.1.6* Drumuri în grafuri

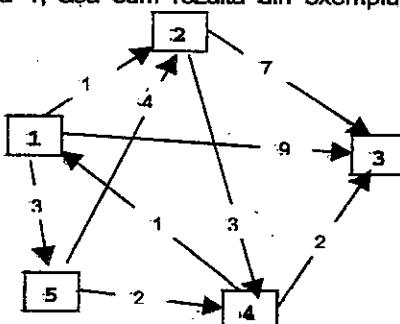
3.1.6.1 Matricea ponderilor

Fie $G=(X, \Gamma)$ un graf orientat. Atășăm fiecărui arc $[x, y] \in \Gamma$ o pondere (cost) $c_{x,y} > 0$.

1. Definim pe mulțimea $X \times X$ o funcție, astfel:

$$f: X \times X \rightarrow \mathbb{R}_+, \quad f([x, y]) = \begin{cases} c_{x,y}, & x \neq y, [x, y] \in \Gamma, \\ \infty, & x \neq y, [x, y] \notin \Gamma, \\ 0 & x = y. \end{cases}$$

Pentru un graf funcția este reținută de o matrice, numită matricea ponderilor forma 1, așa cum rezultă din exemplul următor.



$$\begin{pmatrix} 0 & 1 & 9 & \infty & 3 \\ \infty & 0 & 7 & 3 & \infty \\ \infty & \infty & 0 & \infty & \infty \\ 1 & \infty & 2 & 0 & \infty \\ \infty & 4 & \infty & 2 & 0 \end{pmatrix}$$

2. Definim pe mulțimea $X \times X$ o funcție, astfel:

$$g: X \times X \rightarrow \mathbb{R}, \quad g([x, y]) = \begin{cases} c_{x,y}, & x \neq y, [x, y] \in \Gamma, \\ -\infty, & x \neq y, [x, y] \notin \Gamma, \\ 0 & x = y. \end{cases}$$

Pentru un graf funcția este reținută de o matrice, numită matricea ponderilor forma 2.

Pentru graful de mai sus matricea ponderilor în forma 2 este:

$$\begin{pmatrix} 0 & 1 & 9 & -\infty & 3 \\ -\infty & 0 & 7 & 3 & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty \\ 1 & -\infty & 2 & 0 & -\infty \\ -\infty & 4 & -\infty & 2 & 0 \end{pmatrix}$$

> Modulul **grafuri.cpp** va contine cele două funcții de citire a matricei ponderilor din fișier, două constante care rețin valori foarte mari pentru $\pm\infty$.

```
const float PInfinite = 1.e20;
float MInfinite = -1.e20;
```

Modul în care fișierul reține datele:

```
5
1 2 1
1 3 9
1 5 3
2 4 3
2 3 7
4 3 2
4 1 1
5 2 4
5 4 2
```

> Programul va citi graful, și va crea matricea ponderilor forma 1, cu funcția următoare:

```
void Citire_cost (char Nume_fis[20], float A[50][50], int& n)
{
    int i, j;
    float c;
    ifstream f(Nume_fis, ios::in);
    f>>n;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            if (i==j) A[i][j]=0;
            else A[i][j]=PInfinite;
    while (f>>i>>j>>c) A[i][j]=c;
    f.close();
}
```

- ⇒ Programul va citi graful, și va crea matricea ponderilor forma 2, cu funcția de mai jos. Fisierul reține datele în același mod:

```
void Citire_cost1(char Nume_fis[20], float A[50][50], int& n)
{
    int i,j;
    float c;
    fstream f(Nume_fis,ios::in);
    f>>n;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (i==j) A[i][j]=0;
            else A[i][j]=Minfinit;
    while (f>>i>>j>>c) A[i][j]=c;
    f.close();
}
```

- ✓ Se utilizează matricea ponderilor într-o formă sau altă în funcție de algoritmul folosit.

3.1.6.2 Algoritmul Roy-Floyd

- Fiind dat un graf $G = (X, \Gamma)$ memorat prin matricea ponderilor în formă 1, se cere să se determine pentru orice $x, y \in X$ lungimea minimă a drumului de la nodul x la nodul y . Prin lungimea unui drum înțelegem suma costurilor arcelor care-l alcătuiesc.

- Initial, matricea ponderilor reține numai lungimea drumurilor directe între două noduri, adică nu este permis ca un drum între două noduri să treacă prin un alt nod - pentru arce inexistente se reține $+\infty$.
- La început, încercăm să obținem drumuri mai scurte, între oricare două noduri $i, j \in X$, permitând ca acestea să poată trece prin nodul 1. Aceasta înseamnă că pentru $\forall i, j \in X$, se face comparația: $A[i][j] > A[i][1] + A[1][j]$, adică se compară dacă lungimea drumului direct (care nu trece prin alte noduri) este mai mare decât cea a drumului care trece prin nodul 1. În caz afirmativ se face atribuirea $A[i][j] := A[i][1] + A[1][j]$. După acest pas, matricea va reține lungimea optimă a drumurilor între oricare două noduri, drumuri care pot trece prin nodul 1.
- Încercăm să obținem drumuri mai scurte, între oricare două noduri $i, j \in X$, permitând ca acestea să poată trece și prin nodul intermediar 2. Aceasta înseamnă că pentru $\forall i, j \in X$, se face comparația: $A[i][j] > A[i][2] + A[2][j]$, adică se compară dacă lungimea drumului care nu trece decât prin nodul 1 este mai mare decât cea a drumului care trece prin nodul 2. În caz afirmativ se face atribuirea

$A[i][j] = A[i][2] + A[2][j]$. După acest pas, matricea va reține lungimea optimă a drumurilor între oricare două noduri, drumuri care pot trece prin nodurile intermediare 1 și 2.

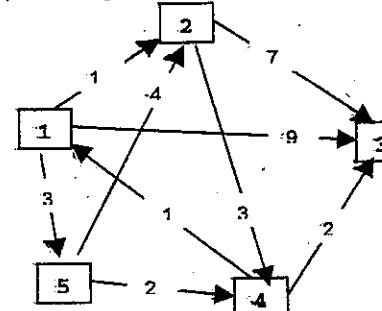
...

Algoritmul continuă în acest mod prin eventuale îmbunătățiri successive ale lungimii drumurilor, considerând ca noduri intermediare 3, 4, ..., n.

Acet algoritm conduce într-adevăr la soluția optimă?

- ⇒ Să observăm că dacă drumul optim între două noduri i și j trece prin nodul k , atunci drumul de la i la k este optim și drumul de la k la j este optim. Dacă, prin absurd, ar exista un drum cu o lungime mai mică între i și k sau k și j atunci, în mod evident, drumul de la i la j nu-ar fi optim.
- ⇒ De aici se trage concluzia că algoritmul caută drumul optim între i și j , printre noduri k , pentru care se cunoaște lungimea optimă a drumurilor de la i la k și de la k la j .
- ⇒ La pasul k se cunoaște lungimea drumului optim între oricare două noduri i și j - în particular, i sau j poate fi k - drumuri care trec numai prin nodurile intermediare 1, 2, ..., $k-1$. După pasul k se cunoaște lungimea drumului optim între oricare două noduri i și j , drumuri care trec numai prin nodurile intermediare 1, 2, ..., k . După pasul n problema este rezolvată!
- ✓ La pasul k nu se optimizează drumurile de la i la k și de la k la j , deoarece nu se obține nici o îmbunătățire prin faptul că aceste drumuri pot trece și prin nodul k . Din acest motiv se poate utiliza o singură matrice, pentru care se efectuează calculele $(A[i][k] \text{ și } A[k][j])$ rămân nemodificate la acest pas).
- ✓ Algoritmul are complexitatea $O(n^3)$.

Exemplu: Fie graful următor:



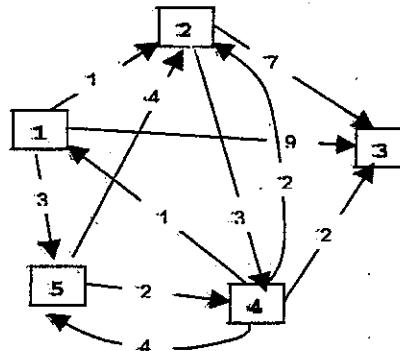
0	1	9	∞	3
∞	0	7	3	∞
∞	∞	0	∞	∞
1	∞	2	0	∞
∞	4	∞	2	0

Pasul 1. Se caută drumurile optime între oricare două noduri, unde drumurile pot trece numai prin nodul intermediar 1.

$$A[4][2] = \infty > A[4][1] + A[1][2] = 1 + 1 = 2. \Rightarrow A[4][2] = 2.$$

$$A[4][5] = \infty > A[4][1] + A[1][5] = 1 + 3 = 4. \Rightarrow A[4][5] = 4.$$

Se obține:



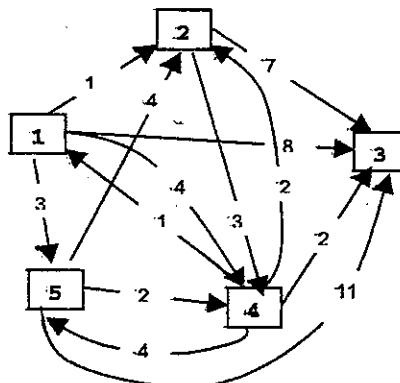
0	1	9	∞	3
∞	0	7	3	∞
∞	∞	0	∞	∞
1	2	2	0	4
∞	4	∞	2	0

Pasul 2. Se caută drumurile optime între oricare două noduri, unde drumurile pot trece și prin nodul intermediar 2.

$$A[1][3] = 9 > A[1][2] + A[2][3] = 1 + 7 = 8. \Rightarrow A[1][3] = 8.$$

$$A[1][4] = \infty > A[1][2] + A[2][4] = 1 + 3 = 4. \Rightarrow A[1][4] = 4.$$

$$A[5][3] = \infty > A[5][2] + A[2][3] = 4 + 7 = 11. \Rightarrow A[5][3] = 11.$$



0	1	8	4	3
∞	0	7	3	∞
∞	∞	0	∞	∞
1	2	2	0	4
∞	4	11	2	0

Pasul 3. Se caută drumurile optime între oricare două noduri, unde drumurile pot trece și prin nodul intermediar 3. Nu se obține nici o îmbunătățire.

Pasul 4. Se caută drumurile optime între oricare două noduri, unde drumurile pot trece și prin nodul intermediar 4.

$$A[1][3] = 8 > A[1][4] + A[4][3] = 4 + 2 = 6. \Rightarrow A[1][3] = 6.$$

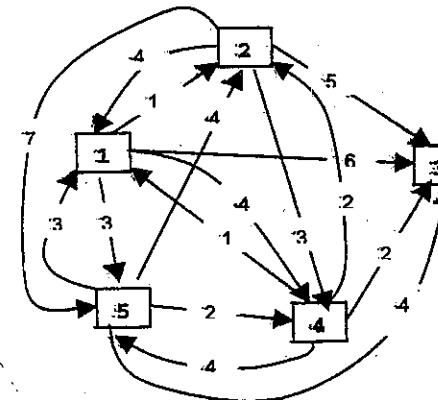
$$A[2][1] = \infty > A[2][4] + A[4][1] = 3 + 1 = 4. \Rightarrow A[2][1] = 4.$$

$$A[2][3] = 7 > A[2][4] + A[4][3] = 3 + 2 = 5. \Rightarrow A[2][3] = 5.$$

$$A[2][5] = 7 > A[2][4] + A[4][5] = 3 + 4 = 7. \Rightarrow A[2][5] = 7.$$

$$A[5][1] = \infty > A[5][4] + A[4][1] = 2 + 1 = 3. \Rightarrow A[5][1] = 3.$$

$$A[5][3] = 11 > A[5][4] + A[4][3] = 2 + 2 = 4. \Rightarrow A[5][3] = 4.$$



0	1	6	4	3
4	0	5	3	7
∞	∞	0	∞	∞
1	2	2	0	4
3	4	4	2	0

Pasul 5. Se caută drumurile optime între oricare două noduri, unde drumurile pot trece și prin nodul intermediar 5. La acest pas nu se obțin îmbunătățiri. În concluzie, am obținut matricea de mai sus, a drumurilor minime.

- Se pune următoarea problemă: fiind dată matricea drumurilor optime și fiind date două noduri i și j se cere să se reconstituie nodurile prin care trece unul din drumurile optime între i și j - pot exista mai multe de aceeași lungime, minimă.

1. Dacă drumul optim între i și j trece prin nodul intermediar k , atunci:

$$A[i][j] = A[i][k] + A[k][j].$$

2. Dacă există k astfel încât $A[i][j] = A[i][k] + A[k][j]$, atunci k se găsește pe unul din drumurile optime de la i la j .

În concluzie, vom aplica strategia generală, **Divide et Impera** pentru depistarea nodurilor pe unde trece drumul optim între i și j . Vezi funcția **drum**.

Dată programul:

```
#include "grafuri.cpp"
float A[50][50];
int n;

void Drum(int i,int j)
{
    int k=1,gasit=0;
    while ((k<=n) && !gasit)
    {
        if ((i!=k) && (j!=k) && (A[i][j]==A[i][k]+A[k][j]) )
        {
            Drum(i,k);Drum(k,j);
            gasit=1;
        }
        k++;
    }
    if (!gasit) cout<<j<<" ";
}

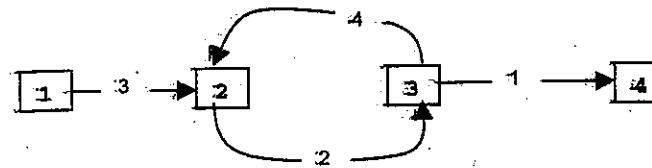
void Scriu_drum(int Nod_Initial, int Nod_Final)
{
    if (A[Nod_Initial][Nod_Final]<PInfininit)
    {
        cout<<"Drumul de la "<<Nod_Initial<<" la "<<Nod_Final
        <<" are lungimea "<<A[Nod_Initial][Nod_Final]<<endl;
        cout<<Nod_Initial<<" ";
        Drum(Nod_Initial,Nod_Final);
    }
    else
        cout<<"Nu există drum de la "<<Nod_Initial<<" la "<<Nod_Final;
}

void Lungime_Drumuri()
{
    int i,j,k;
    for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                if (A[i][j]>A[i][k]+A[k][j])
                    A[i][j]=A[i][k]+A[k][j];
}

main()
{
    Citire_cost("Graf.txt",A,n);
    Lungime_Drumuri();
    Scriu_drum(4,2);
}
```

- Problema pe care o punem este următoarea: se poate adăuga algoritmul Roy Floyd pentru a găsi drumuri de lungime maximă?

Priviți graful următor. Care este lungimea drumului maxim de la nodul 1 la nodul 4? După cum vedeti, graful admite un circuit. Prin urmare, printre-un drum de forma 1 2 3 2 3... 2 3..., 4 se obține un costoricat de mare. Aceasta este, de fapt, $+\infty$.



⇒ Pentru ca algoritmul să găsească soluția corectă este necesar ca graful să nu contină circuite.

Dacă se utilizează matricea ponderilor în forma 1, atunci când între două noduri nu există o muchie care le unește în matrice se reține $+\infty$. Cum drumul care se caută este cel de lungime maximă, algoritmul va alege de fapt acea pseudomuchie, deci rezultatul va fi eronat.

⇒ Pentru ca algoritmul să găsească soluția corectă este necesar să utilizăm matricea ponderilor în forma 2, cînd cu **Citire_cost**.

⇒ Pentru ca algoritmul să găsească soluția corectă este necesar ca testul de comparare a lungimii drumurilor să fie invers:

if $A[i][j] < A[i][k] + A[k][j] \dots$

```
#include "grafuri.cpp"
float A[50][50];
int n;

void Drum(int i,int j)
{
    int k=1,gasit=0;
    while ((k<=n) && !gasit)
    {
        if ((i!=k) && (j!=k) && (A[i][j]==A[i][k]+A[k][j]) )
        {
            Drum(i,k);Drum(k,j);
            gasit=1;
        }
        k++;
    }
    if (!gasit) cout<<j<<" ";
}
```

```

void Scriu_drum(int Nod_Initial, int Nod_Final)
{
    if (A[Nod_Initial][Nod_Final]>Minfinit)
    {
        cout<<"Drumul de la "<<Nod_Initial<<" la "<<Nod_Final
        <<" are lungimea "<<A[Nod_Initial][Nod_Final]<<endl;
        cout<<Nod_Initial<<" ";
        Drum(Nod_Initial,Nod_Final);
    }
    else
        cout<<"Nu exista drum de la "<<Nod_Initial<<" la "<<Nod_Final;
}

void Lungime_Drumuri()
{
    int i,j,k;
    for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                if (A[i][j]<A[i][k]+A[k][j])
                    A[i][j]=A[i][k]+A[k][j];
}

main()
{
    Citire_coste("Graf.txt",A,n);
    Lungime_Drumuri();
    Scriu_drum(1,2);
}

```

3.1.6.3 Algoritmul Dijkstra

- Fieind dat un graf $G=(X,\Gamma)$ memorat prin matricea ponderilor în forma 1, se cere să se determine pentru orice $x,y \in X$ lungimea minimă a drumului de la nodul x la nodul y . Prin lungimea unui drum înțelegem suma ponderilor arcelor care-l alcătuiesc.
- ⇒ Algoritmul selectează nodurile grafului, unul câte unul, în ordinea crescătoare a costului drumului de la nodul R la ele, într-o mulțime S , care initial conține numai nodul R . În felul acesta ne încadrăm în strategia generală **GREEDY**.
- ⇒ În procesul de preluare se folosesc 3 vectori: D , S și T .
- Vectorul D este vectorul lungimii drumurilor de la nodul R la celelalte noduri ale grafului. Prin $D[i]$, unde $i \in \{1, \dots, n\}$, se înțelege costul drumului găsit la un moment dat, între nodul R și nodul i .

- Vectorul T indică drumurile găsite între nodul R și celealte noduri ale grafului. Pentru aceasta se utilizează o memorare specială a drumurilor, numită legătura de tip "tată", în care pentru nodul i se retine nodul precedent pe unde trece drumul de la R la i . Pentru R se memorează 0.
- Vectorul S , numit și vector caracteristic, indică mulțimea S a nodurilor selectate: $S[i]=0$ dacă nodul i este neselectat și $S[i]=1$ dacă nodul i este selectat.

Prezentarea algoritmului.

Pasul 1 Nodul R este adăugat mulțimii S inițial vidă ($S[R]=1$);

- costurile drumurilor de la R la fiecare nod al grafului se preiau în vectorul D de pe linia R a matricei A ;
- pentru toate nodurile i , având un cost al drumului de la R la ele, finit, se pune $T(i)=R$.

Pasul 2 Se execută de $n-1$ ori secvența:

- printre nodurile neselectate se caută cel aflat la distanță minimă față de R și se selectează, adăugându-l mulțimii S ;
- pentru nodurile neselectate se actualizează în D costul drumurilor de la R la ele, utilizând ca nod intermedian nodul selectat, procedând în felul următor:
 - se compară distanța existentă în vectorul D , cu suma dintre distanța existentă în D pentru nodul selectat și distanța de la nodul selectat la nodul pentru care se face actualizarea distanței (preluată din A), iar în cazul în care suma este mai mică, elementul din D corespunzător nodului pentru care se face actualizarea reține suma și elementul din T corespunzător aceluiași nod ia valoarea nodului selectat (drumul trece prin acest nod).

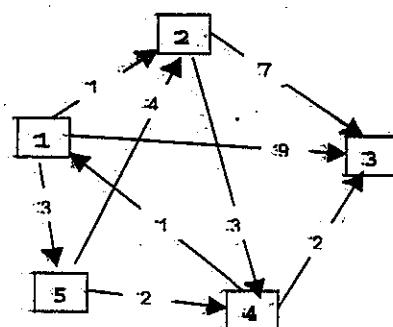
```

if (D[j]>D[poz]+A[poz][j])
{
    D[j]=D[poz]+A[poz][j];
    T[j]=poz;
}

```

Pasul 3. Pentru fiecare nod al grafului, cu excepția lui R , se trasează drumul de la R la el.

Exemplu: fie graful de mai jos, pentru care se dorește afișarea drumurilor minime de la nodul 1 la toate celelalte:



$$\begin{pmatrix} 0 & 1 & 9 & \infty & 3 \\ \infty & 0 & 7 & 3 & \infty \\ \infty & \infty & 0 & \infty & \infty \\ 1 & \infty & 2 & 0 & \infty \\ \infty & 4 & \infty & 2 & 0 \end{pmatrix}$$

Valorile inițiale ale lui S , D , T sunt:

D	0	1	9	∞	3
S	1	0	0	0	0
T	0	1	1	0	1

Cel mai apropiat nod de 1 este nodul 2. Avem:

$$D[3]=9>D[2]+A[2][3]=1+7=8 \Rightarrow D[3]=8, T[3]=2;$$

$$D[4]=\infty>D[2]+A[2][4]=1+3=4 \Rightarrow D[4]=4, T[4]=2;$$

$$D[5]=3<D[2]+A[2][5]=1+\infty=\infty$$

D	0	1	8	4	3
S	1	1	0	0	0
T	0	1	2	2	1

Se selectează nodul 5, dar pentru acesta nu se obține nici o îmbunătățire.

D	0	1	8	4	3
S	1	1	0	0	1
T	0	1	2	2	1

Se selectează nodul 4.

$$D[3]=8>D[4]+A[4][3]=4+2=6 \Rightarrow D[3]=6, T[3]=4;$$

D	0	1	6	4	3
S	1	1	0	1	1
T	0	1	4	2	1

Se selectează nodul 3 pentru care nu se mai pot face îmbunătățiri.

Cum interpretăm rezultatele? De exemplu, distanța de la nodul 1 la nodul 3 este $D[3]=6$, $T[3]=4$, $T[4]=2$, $T[2]=1$, $T[1]=0$. Drumul este 1, 2, 4, 3.

- ✓ Pentru a determina lungimea drumului de la un nod la toate celelalte algoritmul are complexitatea $O(n^2)$.
- ✓ Pentru a determina lungimea drumului de la orice nod la toate celelalte algoritmul are complexitatea $O(n^3)$, se procedează asa cum am văzut pentru fiecare nod în parte, nu numai pentru nodul 1.

```
#include "grafuri.cpp"
float A[50][50], D[50], min;
int S[50], T[50], n, i, j, r, poz;

void drum(int i)
{
    if (T[i]) drum(T[i]);
    cout << i << " ";
}

main()
{
    Citire_cost("Graf.txt", A, n);
    cout << "Introduceti nodul de pornire " << endl;
    cout << "r="; cin >> r; S[r] = 1;
    for (i=1; i<=n; i++)
    {
        D[i] = A[r][i];
        if (i!=r)
            if (D[i]<PInfinit) T[i] = r;
    }

    for (i=1; i<=n-1; i++)
    {
        min = PInfinit;
        poz = -1;
        for (j=1; j<=n; j++)
            if (T[j]==-1)
                if (D[j] < min)
                    min = D[j];
                    poz = j;
        if (poz != -1)
            drum(poz);
    }
}
```

```

for(j=1;j<=n;j++)
if (S[j]==0)
if (D[j]<min)
{
  min=D[j];
  poz=j;
}
S[poz]=1;
for (j=1;j<=n;j++)
if (S[j]==0)
if (D[j]>D[poz]+A[poz][j])
{
  D[j]=D[poz]+A[poz][j];
  T[j]=poz;
}
}

for (i=1;i<=n;i++)
if (i!=r)
if(r[i])
{
  cout<<"distanța de la "<<r<<" la "<<i<<" este "
  <<D[i]<<endl;
  drum(i);
  cout<<endl;
}
else
cout<<" nu există drum de la "<<r<<" la "<<i<<endl;
}

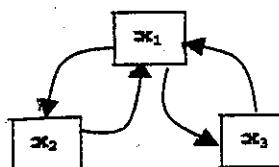
```

3.2 Grafuri neorientate

3.2.1 Notiuni generale

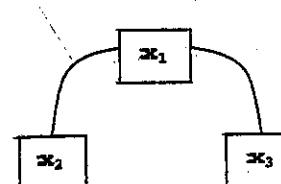
Definiție. Fie $G=(X, \Gamma)$ un graf. Multimea Γ are proprietatea de simetrie dacă și numai dacă din $[x, y] \in \Gamma$, rezultă $[y, x] \in \Gamma$.

Exemplu. Fie $X=\{x_1, x_2, x_3\}$ și $\Gamma=\{(x_1, x_2), (x_2, x_1), (x_1, x_3), (x_3, x_1)\}$. Multimea Γ are proprietatea de simetrie.



$$\begin{aligned} (x_1, x_2) \in \Gamma &\Rightarrow (x_2, x_1) \in \Gamma \\ (x_2, x_1) \in \Gamma &\Rightarrow (x_1, x_2) \in \Gamma \\ (x_1, x_3) \in \Gamma &\Rightarrow (x_3, x_1) \in \Gamma \\ (x_3, x_1) \in \Gamma &\Rightarrow (x_1, x_3) \in \Gamma \end{aligned}$$

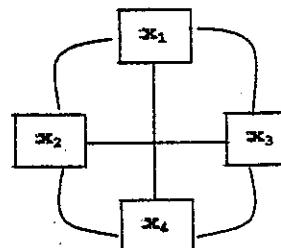
Definiție. Dacă multimea Γ are proprietatea de simetrie, graful $G=(X, \Gamma)$ se numește graf neorientat.



- ✓ În cazul grafurilor neorientate, este lipsit de sens să desenăm ambele arce care unește două noduri. Trasăm una singură, în care nu se fixează sensul și care se numește, de această dată, muchie.

Definiție. Notăm $D=\{(x, x) | x \in X\}$, multimea numită *diagonala produsului cartezian* XX . Considerăm $\Gamma=XX-D$, atunci graful $G=(X, \Gamma)$, se numește graf complet și se notează K_n (n este numărul de vârfuri ale grafului).

- ✓ Într-un graf complet oricare două vârfuri sunt adiacente.

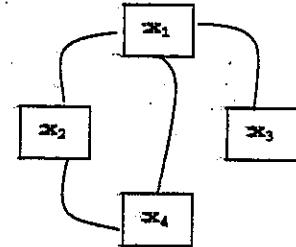


Alăturat observați un graf complet cu patru noduri - K_4 .

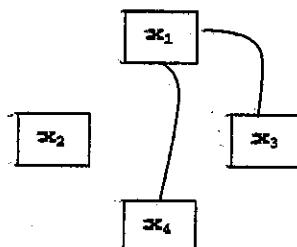
Definiție. Un graf parțial al unui graf neorientat dat $G=(X, \Gamma)$ este un graf $G_1=(X, \Gamma_1)$ unde $\Gamma_1 \subseteq \Gamma$.

- ✓ Un graf parțial al unui graf dat, este el însuși, sau se obține din G prin suprimarea anumitor muchii.

Exemplu:



$G = (X, \Gamma)$

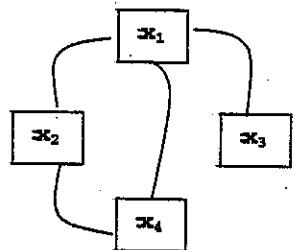


$G_1 = (X, \Gamma_1)$

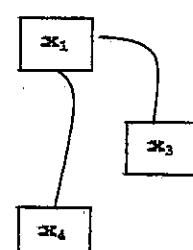
Definiție. Un subgraf al unui graf neorientat $G=(X, \Gamma)$ este un graf $H=(Y, \Gamma_1)$, unde $Y \subseteq X$, iar muchiile din Γ_1 sunt toate muchiile din Γ care au ambele extremități în multimea Y .

- ✓ Un subgraf al unui graf G este graful G sau se obține din G prin suprimarea anumitor vârfuri și a tuturor muchiilor adiacente cu acestea.

Exemplu:

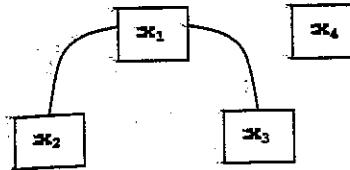


$G = (X, \Gamma)$



$H = (Y, \Gamma_1)$

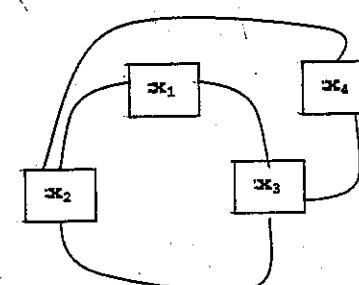
Definiție. Într-un graf neorientat prin gradul unui vârf x se înțelege numărul muchiilor incidente cu vârful x și se notează cu $d(x)$. Un vârf cu gradul 0 este un vârf izolat.



Pentru graful alăturat avem:
 $d(x_1)=2$, $d(x_2)=1$, $d(x_3)=1$,
 $d(x_4)=0$ (x_4 este vârf izolat).

Definiție. Lant. Pentru graful neorientat $G=(X, \Gamma)$ un lant $L = [x_0, x_1, \dots, x_p]$ este o succesiune de vârfuri cu proprietatea că oricare două vârfuri vecine sunt adiacente, adică $[x_0, x_1] \in \Gamma$, $[x_1, x_2] \in \Gamma, \dots, [x_{p-1}, x_p] \in \Gamma$.

- ⇒ Vârfurile x_0 și x_p se numesc extremitățile lantului.
- ⇒ Numărul p se numește lungimea lantului.
- ⇒ Dacă vârfurile x_0, x_1, \dots, x_p sunt distințe, două câte două, lantul se numește elementar.
- ⇒ Un lant L , pentru care $x_0=x_p$, lantul se numește ciclu.
- ⇒ Dacă toate vârfurile ciclului, cu excepția primului și ultimului, sunt distințe, ciclul se numește elementar.



Exemplu: pentru graful neorientat alăturat avem:

$[x_1, x_2], [x_2, x_3], [x_3, x_4]$
este lant de lungime 3.

$[x_1, x_2], [x_2, x_3], [x_3, x_1]$
este ciclu elementar.

Cu mici modificări, de cele mai multe ori simplificări, toți algoritmii învățați pentru grafurile orientate pot fi folosiți și pentru grafurile neorientate.

□ Citirea grafurilor neorientate din fișiere text.

1. În cazul utilizării matricei de adiacență se poate folosi funcția **CitireN**. Dacă nodurile i și j sunt unite, atunci fiind un graf neorientat se consideră că și j și i sunt unite. În acest fel se evită ca fișierul text să contină atât $[i, j]$ cât și $[j, i]$.

```
void CitireN(char Nume_fis[20], int A[50][50], int& n)
{ int i,j;
  fstream f(Nume_fis,ios::in);
```

```

f>>n;
while (f>>i>>j)A[i][j]=A[j][i]=1;
f.close();
}

```

2. În cazul utilizării listelor de adiacență se poate folosi funcția următoare:

```

void Citire_1_N(char Nume_fis[20], Nod* L[50], int& n)
{
Nod* p;
int i,j;
fstream f(Nume_fis,ios::in);
f>>n;
while (f>>i>>j)
{
    p=new Nod; p->adr_urm=L[i]; p->nd=j; L[i]=p;
    p=new Nod; p->adr_urm=L[j]; p->nd=i; L[j]=p;
}
f.close();
}

```

3. Citirea matricei ponderilor în forma 1.

```

void Citire_cost_N(char Nume_fis[20],float A[50][50],int &n)
{
int i,j;
float c;
fstream f(Nume_fis,ios::in);
f>>n;
for (i=1;i<=n;i++)A[i][i]=0;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++) A[i][j]=A[j][i]=PIInfinite;
while(f>>i>>j>>c)A[i][j]=A[j][i]=c;
f.close();
}

```

4. Citirea matricei ponderilor în forma 2 se efectuează cu `Citire_cost_N1`, unde singura deosebire față de funcția anterioară este că am înlocuit constanta PIInfinite cu MIInfinite.

- Parcurgerea grafurilor neorientate (`BF`, `DF`) se face la fel ca parcurgerea grafurilor neorientate - cu aceeași funcții.
- Componentele tare conexe ale grafurilor orientate se numesc componente conexe, în cazul grafurilor orientate. Întrucât existența unui lanț de la x la y implică și existența unui lanț de la y la x , algoritmul se simplifică. O simplă parcurgere în adâncime determină o componentă conexă. Mai jos este prezentat programul care determină componentele conexe ale unui graf neorientat.

```

#include "grafuri.cpp"
int S[50],A[50][50],n,i,k;

void df_r(int nod)
{
    int k;
    cout<<nod<<" "; S[nod]=1;
    for (k=1;k<=n;k++)
        if ((A[nod][k]==1)&&(S[k]==0)) df_r(k);
}

main()
{
    CitireN("Graf.txt",A,n);
    k=1;
    for (i=1;i<=n;i++)
        if (S[i]==0)
        {
            cout<<" componenta "<<k<<endl;
            df_r(i);
            cout<<endl;
            k++;
        }
}

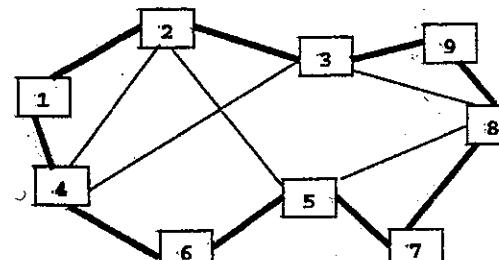
```

- Matricea drumurilor are aceeași semnificație și se determină la fel și pentru grafurile neorientate. În plus, de această dată ea este simetrică, pentru că este evident că dacă există drum de la i la j există și drum de la j la i .
- Algoritmii pentru determinarea drumurilor optime, prezenți pentru grafuri orientate, funcționează fără probleme și în cazul grafurilor neorientate.

3.2.2 Grafuri hamiltoniene

Definiție. Fie $G=(x, T)$ un graf. Se numește ciclu hamiltonian un ciclu elementar care trece prin toate vîrfurile grafului.

- ✓ Un graf care admite un ciclu hamiltonian se numește graf hamiltonian.



Graful alăturat este hamiltonian. Un drum hamiltonian este:

1,2,3,9,8,7,5,6,4,1

⇒ Pentru determinarea ciclurilor hamiltoniene nu se cunosc algoritmi care să rezolve problema în timp polinomial. Acesta este motivul pentru care folosim backtracking.

□ Se dă un graf neorientat, prin matricea de adiacență. Se cere să se determine, dacă există, un ciclu hamiltonian.

```
#include "grafuri.cpp"
int st[50], a[50][50], n, k;

void Init()
{
    st[k]=1;
}

int Am_Succesor()
{
    if (st[k]<n)
        { st[k]++;
          return 1;
        }
    else return 0;
}

int Is_Valid()
{
    if (!a[st[k-1]][st[k]]) return 0;
    else
        for (int i=1; i<=k-1; i++)
            if (st[i]==st[k]) return 0;
        if (k==n && a[1][st[k]]) return 0;
    return 1;
}

int Solutie()
{
    return b==n;
}

void Tipar()
{
    for (int i=1; i<=n; i++) cout<<"Nodul "<<st[i]<<endl;
    k=0;
}

void back()
{
    int AS;
    k=2; Init();
    while (k>1)
        { do {} while ((AS=Am_Succesor()) && Is_Valid());
          if (AS)
              if (Solutie()) Tipar();
              else {k++;Init();}
              else k--;
        }
}

main()
{
    st[1]=1; k=2; st[k]=1;
    CitireN("graf.txt", a, n);
    back();
}
```

✓ În determinarea unui ciclu hamiltonian nu contează nodul de pornire. De exemplu, fie ciclul $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$, atunci $2 \rightarrow 3 \rightarrow 1 \rightarrow 2$ sau $3 \rightarrow 1 \rightarrow 2 \rightarrow 3$ reprezintă același ciclu.

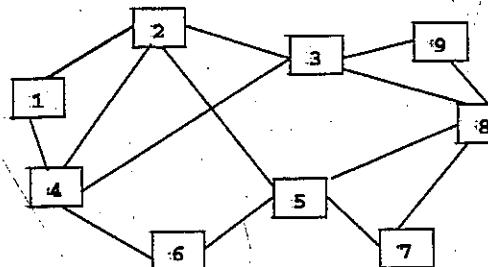
✓ N-ai mai întâlnit această problemă? Este la comis voiajorului...

3.2.3 Grafuri euleriene

Definiție. Un lanț π al unui graf $G=(X, \Gamma)$ care conține fiecare muchie o dată și numai o dată se numește lanț eulerian. Dacă $x_0=x_n$ și lanțul este eulerian atunci, ciclul respectiv se numește ciclu eulerian.

Definiție. Un graf care conține un ciclu eulerian se numește eulerian.

Observație. Faptul că un graf este eulerian nu înseamnă că nu are vîrfuri izolate.



Graful alăturat este
eulerian. Un ciclu
eulerian este:

1, 2, 4, 6, 5, 7, 8, 3, 9, 8,
5, 2, 3, 4, 1

Teoremă. Un graf $G=(X, \Gamma)$, fără vîrfuri izolate, este eulerian dacă și numai dacă este conex și gradele tuturor vîrfurilor sale sunt numere pare.

Demonstrație:

⇒ Fie un graf fără vîrfuri izolate care conține un ciclu eulerian (graf eulerian).

Demonstrăm că graful este conex și gradele tuturor vîrfurilor sale sunt pare.

- Graful este conex. Fie un ciclu eulerian. Fie x și y două vîrfuri ale grafului. Cum x nu este izolat, rezultă că există o muchie incidentă lui x . Tot așa, există o muchie incidentă cu y . Cum ciclul este eulerian (conține toate muchiile) va conține și cele două muchii. Prin urmare, x și y sunt unite printr-un lanț. Cum x și y au fost alese întâmplător, rezultă că oricare două vîrfuri sunt unite printr-un lanț. În concluzie, graful este conex.
- Gradele tuturor muchiilor sunt pare. Să presupunem că vîrful x apare de k ori în lanț. La fiecare apariție a lui x , se vor utiliza 2 muchii (care nu

mai pot fi reutilizate). În concluzie, vom avea $d(v)=2k$. Cum vârful v a fost luat arbitrar, rezultă c.c.t.d.

← Să presupunem că graful G este conex și are gradele tuturor vârfurilor numere pare. Să arătăm că G conține un ciclu eulerian.

Pomim cu unul din vârfurile grafului. Fie el v . Înțeționăm să formăm un ciclu, (nu neapărat eulerian), care începe și se termină cu v . Conform ipotezei, există un număr par de muchii incidente cu v . Selectăm una dintre ele, incidentă la vârful v_1 . Atât în v cât și în v_1 rămân cu un număr impar de muchii incidente. Selectăm, o muchie incidentă la v_1 și repetăm rationamentul. Multimea vârfurilor este finită. În concluzie, vom ajunge, la un moment dat, într-unul din vârfurile prin care am trecut (în procesul de selecție a muchiilor). Dacă acesta este chiar v am obținut ciclul căutat. Să presupunem că am ajuns în alt vârf decât v . Înțețând cont de algoritm, au fost selectate un număr impar de muchii incidente aceluia vârf. Prin ipoteză, există un număr par de muchii incidente acestuia. Prin urmare, există o muchie pe care o putem selecta. Procedeul se repetă până când selectăm o muchie incidentă la v .

În acest fel am găsit un ciclu (care începe și se termină cu v). Să-l notăm cu C . Există două posibilități:

- 1) Ciclul obținut este eulerian, caz în care teorema este demonstrată.
- 2) Ciclul nu este eulerian. În acest caz, operăm asupra grafului G , renunțând la muchiile selectate. În acest fel, se obține un graf parțial al lui G pe care îl notăm H . În același timp, refinem ciclul găsit. Din analiza grafului G rezultă:
 - ⇒ gradele tuturor vârfurilor sale sunt pare (pentru fiecare vârf din ciclu, au fost eliminate muchii în număr par, deci gradul său a rămas par);
 - ⇒ multimea muchiilor lui H este nevidă (ciclul găsit nu este eulerian);
 - ⇒ cel puțin una din muchiile lui H are o extremitate comună (vârf) cu una din muchiile ciclului selectat (contrar înseamnă că n-ar exista drum între un vârf care nu aparține ciclului C și unul care aparține acestuia, caz în care se contrazice faptul că graful este conex).
 - ⇒ Pomind dintr-un astfel de vârf comun, selectăm întotdeauna ca la început un alt ciclu C_1 . Formăm din C și C_1 un nou ciclu (prin intermediul vârfului comun), extregem muchiile ciclului C_1 ...
 - ⇒ Repetăm procedeul până la selectia unui ciclu eulerian.

□ Programul care urmează verifică dacă un graf este conex și are gradele tuturor vârfurilor pare. Dacă aceste condiții sunt îndeplinite, programul găsește un ciclu eulerian.

1. Verificarea conexității. Dacă în urma parcurgerii grafului în adâncime rămân vârfuri neatinse înseamnă că graful nu este conex. Prin urmare, funcția **conex** apelează funcția **EF**, după care se explorează vectorul **is** al vârfurilor atinse.

2. Verificarea faptului că toate vârfurile au grade pare. Această operație este realizată de funcția booleană **grade_pare**. În fapt, se verifică dacă fiecare linie a matricei de adiacență are suma elementelor pară.

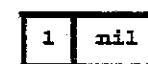
În cazul în care condițiile sunt îndeplinite se trece la găsirea unui drum eulerian. Drumul eulerian va fi reținut de o listă liniară simplu înăntuită, în care fiecare element este un nod al grafului. Variabila **Alista** reține adresa primului element al ei.

Lista se initializează cu primul nod de la care pomenește căutarea și anume nodul **1**. Ca și în cazul căutării ciclului hamiltonian, nu contează nodul de pomire.

Funcția booleană **Adauga** explorează lista, pomind de la primul nod al ei și căută un nod pentru care există muchii incidente. În cazul în care un astfel de nod este găsit, fie el v , apelează funcția **Ciclu**. Aceasta identifică un ciclu care începe și se sfârșeste cu acest nod. Pomind de la el se selectează o muchie incidentă lui, care este incidentă și în alt nod, fie el v_1 . Apoi se selectează o altă muchie incidentă în v_1 care este incidentă și la altul, v_2 . Selectia continuă în acest mod până se ajunge din nou la v . Fiecare nod incident la muchiile găsite se introduce în lista liniară simplu înăntuită. Fiecare muchie selectată este stearsă din matricea de adiacență. Algoritmul se termină atunci când nici un nod din listă nu mai are muchii incidente.

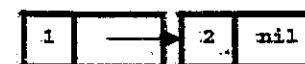
Iată cum funcționează programul, pentru graful prezentat la început:

Initial, lista liniară simplu înăntuită conține un nod: **1**.



Funcția **Adauga** explorează lista și găsește că nodul **1** are muchii incidente. Se apelează **Ciclu**.

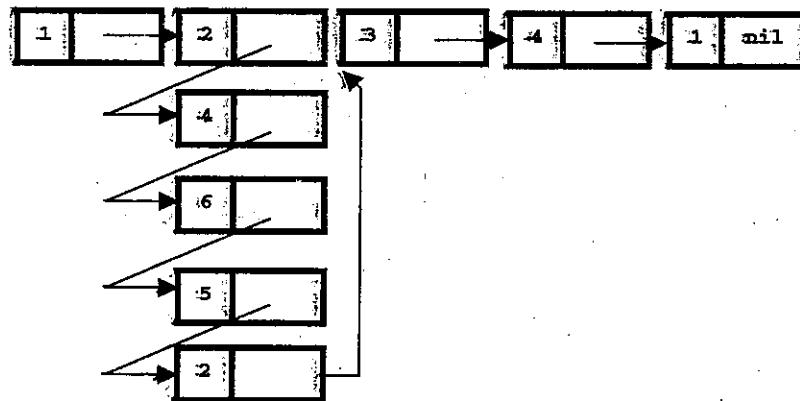
Se selectează prima muchie **1 2**, deci se adaugă în listă nodul **2**.



Se selectează apoi muchia 2-3, apoi 3-4, apoi 4-1. Funcția Ciclu își încheie executarea și se revine în Adauga. Lista este:



Se explorează în continuare lista. Nodul 1 nu mai are muchii incidente, dar nodul 2 da, și anume 2-4. Se selectează 2-4, apoi 4-6 (4-3 a fost selectată) apoi 6-5, apoi 5-2.



Algoritmul continuă în acest mod până când au fost selectate toate muchiile. La sfârșit se afisează lista diniară.

```
#include "grafuri.cpp"

int A[50][50], S[50], n;
Nod *Alista, *Indice;

void ciclu(Nod* v)
{
    int Nodul;
    Nod *ANod_baza, *ANod_gasit, *ANod_urm;
    ANod_urm=v->adr_urm;
    ANod_baza=v;
    do
    {
        Nodul=1;
        while (A[ANod_baza->nd][Nodul]==0) Nodul++;
        A[ANod_baza->nd][Nodul]=0; A[Nodul][ANod_baza->nd]=0;
        ANod_gasit=new Nod; ANod_gasit->nd=Nodul;
        ANod_gasit->adr_urm=0;
        ANod_baza->adr_urm=ANod_gasit; ANod_baza=ANod_gasit;
    } while (ANod_gasit->nd!=v->nd);
    ANod_baza->adr_urm=ANod_urm;
}
```

```
int adauga()
{
    int i,gasit=0;
    Indice=Alista;
    while ((Indice && !gasit))
    {
        for (i=1;i<=n;i++)
            if (A[Indice->nd][i]==1) gasit=1;
        if (!gasit) Indice=Indice->adr_urm;
    }

    if (Indice)
    {
        ciclu(Indice);
        return 1;
    }
    else return 0;
}

int grade_pare()
{
    int i=1,j,s,gasit=0;
    while ( (i<=n) && !gasit )
    {
        s=0;
        for (j=1;j<=n;j++) s+=A[i][j];
        if (s%2) gasit=1;
        i++;
    }
    return !gasit;
}

void df(int nod)
{
    int k;
    S[nod]=1;
    for (k=1;k<=n;k++)
        if ( (A[nod][k]==1) && (S[k]==0)) df(k);
}

int conex()
{
    int gasit=0,i;
    df(1);
    for (i=1;i<=n;i++)
        if (S[i]==0) gasit=1;
    return !gasit;
}
```

```

main()
{
    CitireN("Graf.txt", A, n);
    if (!conex())
        if (!grade_pare())
    {
        Alista=new Nod;
        Alista->nd=1; Alista->adr_urm=0;
        while (adauga());
        Indice=Alista;
        while (Indice)
        {
            cout<<Indice->nd<<" ";
            Indice=Indice->adr_urm;
        }
    }
    else cout<<"Graful nu indeplineste conditiile de paritate";
    else cout<<"Graful nu este conex ";
}

```

- ✓ Dacă luăm în considerare faptul că la fiecare pas se selectează o muchie, atunci algoritmul are complexitatea $O(m)$, unde m este numărul de muchii.
- ✓ În realitate, organizarea datelor de la care am pornit, matricea de adiacență nu ne permite acest lucru. Se poate, totuși, găsi o structură care să ne permită selecția imediată a unei muchii. Puteti modifica programul astfel încât să țină cont de această ultimă observație? Exercițiul!

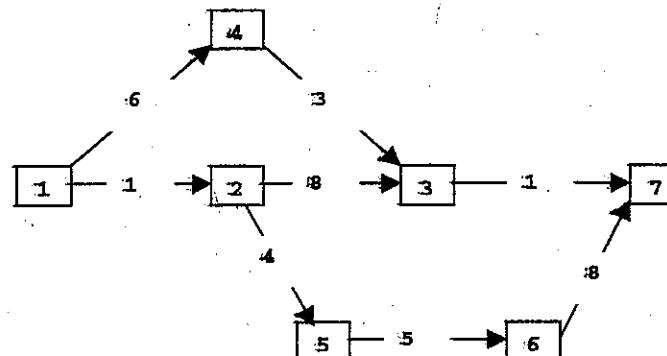
3.3* Rețele de transport -facultativ-

3.3.1 Ce este o rețea de transport

Definiție. Fie $G=(X,T)$ un graf orientat. Se numește rețea de transport ansamblul $R=(G,st,fin,c)$, unde:

- G este graful orientat;
- $st \in X$, și $d^+(st) > 0$, $d^-(st) = 0$;
- $fin \in X$, $fin \neq st$, $d^-(fin) > 0$, $d^+(st) = 0$.
- $c: X \times X \rightarrow N$. Daca $[i,j] \notin T$, atunci $c_{i,j} = 0$.

Exemplu: Mai jos puteti observa o rețea de transport în care st este nodul 1, fin este nodul 7, iar valorile diferențe de 0 ale funcției c sunt trecute pe arcuri:



Semnificație: De la st la fin trebuie transportat ceva anume. Funcția c are semnificația de capacitate de transport între oricare două noduri.

O rețea de transport se citește dintr-un fișier text cu datele organizate ca mai jos:

Linia 1 - n -numărul de noduri;

Linia 2 - st și fin ;

următoarele linii sunt de forma i j c și au semnificația că de la nodul i la nodul j există capacitatea de transport c .

Iată cum se citește rețeaua de mai sus:

7
1 7
1 2 1
2 3 8
3 7 1
1 4 6
4 3 3
2 5 4
5 6 5
6 7 8

Pentru citirea rețelei vom utiliza funcția de mai jos, inclusă în modulul grafuri:

```
void Citire_Cap(char Nume_fis[20], int A[50][50][2],
                 int n, int st, int fin)
{
    int i,j,cost;
    ifstream f(Nume_fis,ios::in);
    f>>n>>st>>fin;
    while (f>>i>>j>>cost) A[i][j][0]=cost;
    f.close();
}
```

3.3.2 Flux într-o rețea de transport

Definiție: fie rețeaua de transport $R=(G, st, fin, c)$, unde $G=(X, \Gamma)$. Vom numi flux al rețelei R , o funcție $\phi: X \times X \rightarrow N$ care îndeplinește simultan următoarele două condiții:

1. Pentru orice arc avem: $0 \leq \phi(i, j) \leq c(i, j)$ – adică fluxul nu poate depăși capacitatea arcului respectiv;

✓ Evident, pe arce inexistente fluxul este 0.

2. Pentru orice vârf din $i \in X - \{st, fin\}$ avem:

$$\sum_{j \in X} \phi(i, j) = \sum_{j \in X} \phi(j, i)$$

✓ Aceasta are semnificația că suma valorilor asociate prin ϕ pentru arcele care ieș din nodul i este egală cu suma valorilor asociate prin ϕ pentru arcele care intră în nodul i .

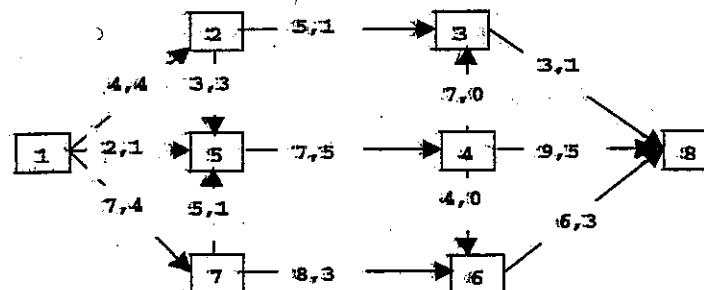
✓ De aici rezultă că suma valorilor asociate prin ϕ pentru arcele care pleacă din st este egală cu suma valorilor asociate prin ϕ pentru arcele care ajung în fin .

$$\sum_{j \in X} \phi(st, j) = \sum_{j \in X} \phi(j, fin)$$

Definiție. Se numește valoarea fluxului ϕ suma:

$$S = \sum_{j \in X} \phi(st, j) = \sum_{j \in X} \phi(j, fin)$$

Exemplu: fie rețeaua de transport de mai jos în care $st=1$, $fin=8$ și pentru fiecare arc se asociază două numere: primul dintre ele este capacitatea arcului, iar al doilea valoarea asociată prin ϕ .

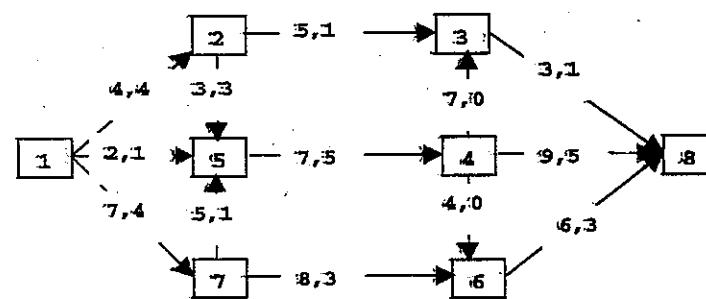


- ✓ Pentru fiecare nod valoarea asociată prin ϕ este mai mică sau egală decât capacitatea arcului respectiv. De exemplu, pentru $[1,2]$ capacitatea este 7, iar valoarea asociată prin ϕ este 4.
- ✓ Pentru orice nod diferit de st și fin , suma valorilor asociate prin ϕ arcelor care intră în nodul respectiv este egală cu suma valorilor asociate prin ϕ arcelor care ieș din nodul respectiv. De exemplu, pentru nodul 7 intră un arc cu valoarea 4 și ieș două arce cu valorile 1 și 3.
- ✓ Valoarea și a fluxului este 9. Ea se obține ca sumă a valorilor asociate arcelor care ieș din st ($4+1+4$), fie ca sumă a valorilor arcelor care intră în fin ($1+5+3$).
- ✓ Fluxul are semnificația de transport la un moment dat de la st la fin , transport care se face în limita capacitatii drumurilor între noduri și în care suma valorilor transportate care ajung într-un nod intermediu este egală cu suma celor care ieș.
- ✓ În mod evident, fiind dată o rețea de transport există mai multe funcții ϕ , fiecare având valoarea ei (valoarea fluxului).
- ✓ Fiecare rețea R admite fluxul nul –adică toate valorile lui ϕ sunt nule.

3.3.3 Determinarea fluxului de valoare maximă

Problemă. Se dă rețeaua $R=(G, st, fin, c)$, pe care este definit un flux ϕ . Se cere un flux de valoare maximă pentru rețeaua respectivă: Φ_{max} .

Pentru a înțelege algoritmul care urmează să îl prezentăm, trebuie să ne înșușim anumite noțiuni. Exemplul vor fi date pe rețeaua următoare în care **st=1**, **fin=8** și pentru care avem un flux ϕ .



Fie un **drum** în rețeaua \mathcal{R} , de la **st** la **fin**, în care orice arc poate fi parcurs fie în sensul în care este orientat, fie în sens invers.

Exemple de drume în \mathcal{R} de la **st** la **fin**:

- a) 1 2 3 :8 este un astfel de drum, în care toate arcele sunt parcurse în sensul pe care îl au.
- b) 1 5 7 :6 :8 este un astfel de drum, dar aici arcul [5 7] este parcurs în sens invers. Celelalte arce sunt parcurse în sensul pe care îl au.
- Fieind dat un drum de la **st** la **fin**, fiecărui arc $[i, j]$ care-l alcătuiește i se asociază un număr numit **valoare reziduală**, în felul următor:

$$vr(i, j) = \begin{cases} c(i, j) - \phi(i, j), & \text{pentru } [i, j] \text{ parcurs direct;} \\ \phi(j, i) & \text{pentru } [i, j] \text{ parcurs invers.} \end{cases}$$

Exemplu. Pentru drumul 1 5 7 6 8 avem: $vr(1, 5) = 2 - 1 = 1$; $vr(5, 7) = 1 - 1 = 0$; $vr(7, 6) = 8 - 3 = 5$; $vr(6, 8) = 6 - 3 = 3$.

- Se numește **drum în creștere** în rețeaua \mathcal{R} un drum de la **st** la **fin** în care orice arc poate fi parcurs, fie în sensul în care este dat, fie în sens invers și în care valoarea reziduală a fiecărui arc este diferită de 0.

Exemple:

- a) 1 5 6 7 :8 este un drum în creștere pentru că este de la **st** la **fin** și valoarea reziduală a fiecărui arc este diferită de 0.

- b) 1 2 3 :8 nu este drum în creștere, pentru că valoarea reziduală a arcului [1, 2] este 0.

- Definim **capacitatea reziduală** a unui drum în creștere și o notăm cu ε , minimul valorilor reziduale ale arcelor care-l alcătuiesc.

Exemplu: pentru drumul în creștere 1 5 7 6 :8 avem:

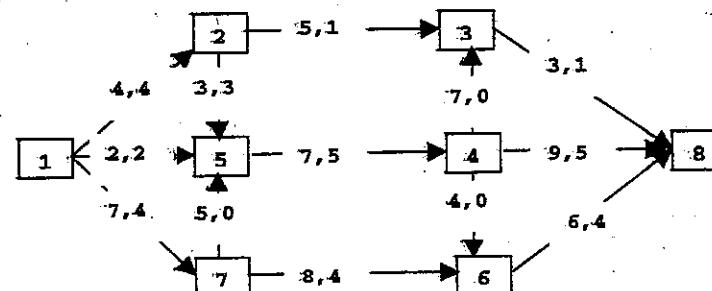
$$\varepsilon = \min \{1, 1, 5, 3\} = 1.$$

- Criteriu de mărire a fluxului: fiind dat un drum în creștere de capacitate reziduală ε , modificăm valorile determinate de ϕ , pentru a obține un flux de valoare mai mare ϕ' , astfel:

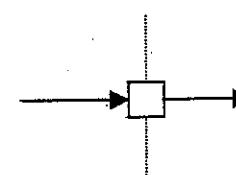
$$\phi'(i, j) = \begin{cases} \phi(i, j), & \text{daca } [i, j] \text{ nu aparține drumului în creștere;} \\ \phi(i, j) + \varepsilon, & \text{daca } [i, j] \text{ este parcurs în sens direct;} \\ \phi(i, j) - \varepsilon, & \text{daca } [i, j] \text{ este parcurs în sens invers.} \end{cases}$$

Exemplu pentru rețeaua anterioară și drumul în creștere 1 5 7 6 :8 avem:

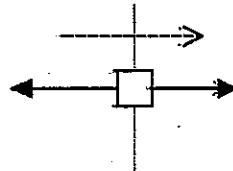
$$\phi'(1, 5) = 1 + 1 = 2; \phi'(5, 7) = 1 - 1 = 0; \phi'(7, 6) = 3 + 1 = 4; \phi'(6, 8) = 3 + 1 = 4;$$



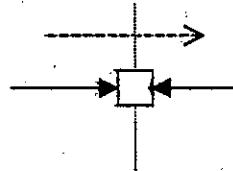
- ✓ $\phi' > \phi$ – pentru că va fi mărită valoarea pe un arc care ieșe din **st**.
- ✓ Drumul respectă condiția 2 a fluxului pentru că, un nod prin care trece drumul în creștere se poate găsi într-o singură situație:



Se mărește valoarea pe ambele arce, deci, ce intră, este egal cu ce ieșe.



Pe primul arc se scade valoarea, pe al doilea se mărește, deci ce intră este egal cu ce ieșe.



Pe primul arc se mărește valoarea, pe al doilea se scade, deci ce intră este egal cu ce ieșe.

- ✓ Pentru a găsi fluxul de valoare maximă suntem conduși la ideea de a găsi un drum în creștere și de a mări fluxul prin capacitatea sa reziduală, apoi să găsim un alt drum în creștere și să mărim din nou fluxul, până când nu mai avem drumuri în creștere. Atunci când nu mai avem drumuri în creștere vom avea fluxul de valoare maximă.
- ✓ Un astfel de algoritm se încadrează în strategia **Greedy**, pentru că la fiecare pas alege un drum în creștere și mărește valoarea funcției pentru el, până când nu mai avem drumuri în creștere.
- ✓ Evident, am prezentat algoritmul doar intuitiv, fără a oferi o demonstrație a acestuia, pentru că ea depășește nivelul clasei a XII-a.

În cele ce urmează vom prezenta algoritmul lui **Ford Fulkerson** optimizat. De ce optimizat? Pentru că algoritmul va găsi drumurile în creștere în ordinea crescătoare a numărului de muchii parcuse.

- ✓ Se demonstrează faptul că dacă selecția drumurilor în creștere se face în ordinea crescătoare a numărului de muchii parcuse, algoritmul va fi mai rapid, deși are aceeași complexitate.
- ✓ Generarea unui drum în creștere cu un număr minim de muchii se face printr-o simplă parcurgere în fățime (BFS) a grafului, până când în coadă este introdus nodul **fin**.

Algoritmul lui **Ford Fulkerson** optimizat. Se pomenește de la o rețea de transport R.

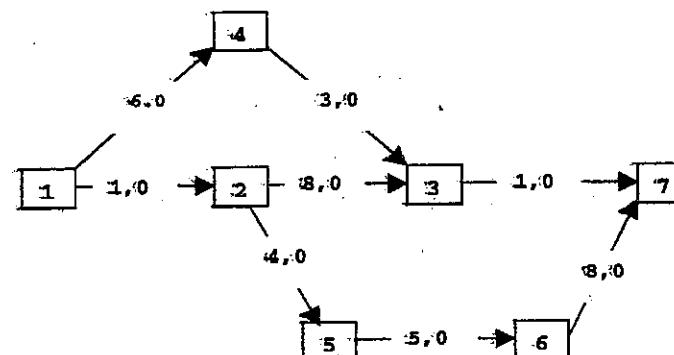
P1. Se asociază rețelei un flux ϕ . Pentru simplitate, ϕ este fluxul nul.

P2. Se caută un drum în creștere cu un număr minim de muchii. Aceasta înseamnă că se face o parcurgere BFS a grafului.

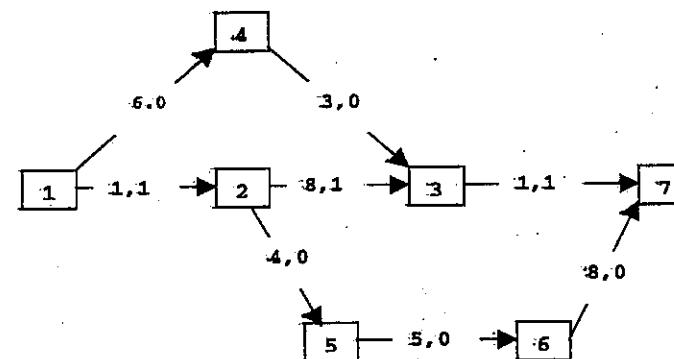
P3. În cazul în care acesta a fost găsit, ceea ce este echivalent cu introducerea în coadă a nodului **fin**, se actualizează valorile fluxului pe drumul găsit și se trece la **P2**. Altfel, se trece la **P4**.

P4. Se afisează fluxul maxim.

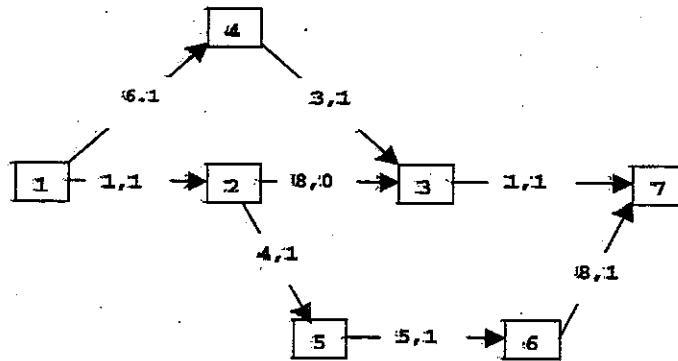
Exemplu. Fie rețeaua de transport următoare, la care am asociat fluxul nul.



Drumul în creștere cu număr minim de muchii este: 1 → 2 → 3 → 7. El are valoarea reziduală 1. După actualizare, se obține un flux, mai mare cu 1:



Se caută un alt drum în creștere. El este 1 → 4 → 3 → 2 → 5 → 6 → 7. Valoarea sa reziduală este 1. După actualizare se obține:



Întrucât nu mai există drume în creștere, am obținut fluxul de valoare maximă și aceasta este 2.

Iată programul:

```
#include "grafuri.cpp"
#include <math.h>
int A[50][50][2], n, st, fin, gasit,
    coada[50], s[50], f[50], i_c, sf_c, i, j, min;

void refac(int Nod)
{
    if (Nod!=st)
        if (s[Nod]>0)
        {
            if (min>A[s[Nod]][Nod][0]-A[s[Nod]][Nod][1])
                min=A[s[Nod]][Nod][0]-A[s[Nod]][Nod][1];
            refac(s[Nod]);
            A[s[Nod]][Nod][1]+=min;
        }
        else
        {
            if (min>A[Nod][s[abs(Nod)]][1])
                min=A[Nod][abs(s[Nod])][1];
            refac(abs(s[Nod]));
            A[Nod][abs(s[Nod])][1]-=min;
        }
    }

void drum_in_crestere()
{
    gasit=0;
    i_c=sf_c=1; coada[i_c]=st;
    while( (i_c<sf_c) && (coada[sf_c]!=fin) )
    {
        if (A[coada[i_c]][i_c][0]-A[coada[i_c]][i_c][1]>0 && (s[i_c]==0))
        {
            s[i_c]=coada[i_c]; coada[++sf_c]=i_c;
        }
        else
            if ((A[i_c][coada[i_c]][1]>0) && (s[i_c]==0) && (i_c==st) )
            {
                s[i_c]=-coada[i_c]; coada[++sf_c]=i_c;
            }
            i_c++;
    }
    i_c++;
}
if (coada[sf_c]==fin) gasit=1;
}

void caut()
{
    do
    {
        for (i=1;i<=n;i++) s[i]=0;
        drum_in_crestere();
        if (s[fin])
        {
            min=32000;
            refac(fin);
        }
    } while (gasit);
}

main()
{
    Citire_Cap("Graf.txt", A, n, st, fin);
    caut();
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=n;j++) cout<<A[i][j][1]<<" ";
        cout<<endl;
    }
}
```

```

    i=1;
    while ( (i<=n) && !gasit )
    {
        if ((A[coada[i_c]][i_c][0]-A[coada[i_c]][i_c][1]>0) && (s[i_c]==0))
        {
            s[i_c]=coada[i_c]; coada[++sf_c]=i_c;
        }
        else
            if ((A[i_c][coada[i_c]][1]>0) && (s[i_c]==0) && (i_c==st) )
            {
                s[i_c]=-coada[i_c]; coada[++sf_c]=i_c;
            }
            i_c++;
    }
    i_c++;
}
if (coada[sf_c]==fin) gasit=1;
}

void caut()
{
    do
    {
        for (i=1;i<=n;i++) s[i]=0;
        drum_in_crestere();
        if (s[fin])
        {
            min=32000;
            refac(fin);
        }
    } while (gasit);
}

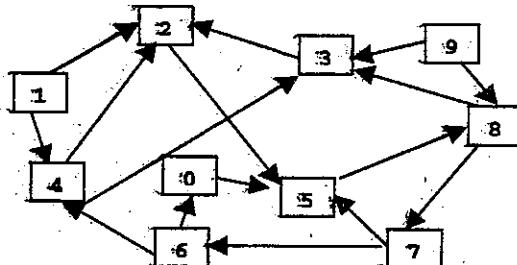
main()
{
    Citire_Cap("Graf.txt", A, n, st, fin);
    caut();
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=n;j++) cout<<A[i][j][1]<<" ";
        cout<<endl;
    }
}
```

- ✓ Întrucât parcurgerea în lățime se face în $O(m)$, unde m este numărul muchiilor, și cum la fiecare pas fluxul crește cu cel puțin 1, dacă s este valoarea sa maximă, atunci complexitatea algoritmului este $O(ms)$.

Probleme propuse

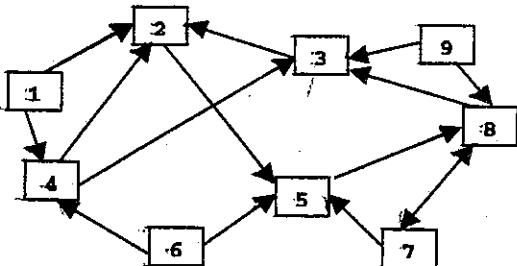
a) Grafuluri orientate (se mai numesc și digrafuri).

1. Pentru graful orientat de mai jos, stabiliți care dintre afirmațiile următoare sunt adevărate?



- a) Graful are 9 noduri;
- b) Graful are 16 arce;
- c) Nodurile 7 și 4 sunt adiacente;
- d) $d^+(7)=2$;
- e) $d^-(7)=3$;
- f) Dacă se elimină nodul 1 și arcele [1,4], [1,2] se obține un subgraf același;
- g) 6,4,2,5,8 este drum elementar de lungime 4;
- h) 2,5,8,3,2 este un circuit elementar;
- i) Nodurile 1 și 4 aparțin aceleiasi componente tare conexă.

2. Pentru graful următor:



- a) Indicați o modalitate de parcurgere în lătime pornind de la nodul 1. Aceeași problemă dacă se pometește de la nodul 7.

- b) La fel ca la a) numai că se cere parcurgerea în adâncime.
- c) Care este matricea de adiacență?
- d) Arătați modul în care se reprezintă graful prin utilizarea listelor de adiacență.
- e) Care este matricea drumurilor?
- f) Care sunt componentele tare conexă?
- g) Este graful tare conexă?

- 3. Se citește un graf orientat. Să se afișeze toate nodurile pentru care $d^+(x)=d^-(x)$ (gradul exterior este egal cu gradul interior).
- 4. Se citește un graf orientat și un anumit nod x . Se cere să se listeze toate nodurile adiacente cu x .
- 5. Se citește un graf și se memorează sub forma listelor de adiacență. Se citește un nod x . Se cere ca programul dv. să afișeze $d^-(x)$.
- 6. Să se scrie o funcție care, pentru un graf dat, transformă matricea de adiacență a unui graf în liste de adiacente.
- 7. Să se scrie o funcție care, pentru un graf dat, transformă listele de adiacență în matrice de adiacență.
- 8. Se dă un graf memorat prin matricea de adiacență și un nod al său, x . Se cere să se parcurgă graful în lătime, pornind de la nodul x . Algoritmul va utiliza coada creată ca listă liniară simplu înlăntuită.
- 9. Se dă un graf memorat prin liste de adiacență și un nod al său, x . Se cere să se parcurgă graful în lătime, pornind de la nodul x . Algoritmul va utiliza coada creată ca listă liniară simplu înlăntuită.
- 10. Se dă un graf memorat sub forma matricei de adiacență. Se cere să se afișeze matricea drumurilor. Algoritmul va utiliza parcurgerea în adâncime.
- 11. Se dă un graf memorat sub forma listelor de adiacență. Se cere să se afișeze matricea drumurilor. Algoritmul va utiliza parcurgerea în adâncime.
- 12. Scrieți un program care citește un graf orientat dintr-un fișier text și îl parcurge în adâncime. Programul nu va utiliza recursivitatea.

- Indicație: se va utiliza stiva.
- 13. Scrieți un program care citește un graf orientat dintr-un fișier text și afișează "Da" dacă graful are cel puțin un circuit și "Nu" în caz contrar.

14. Fie grafurile $G_1 = (X_1, T_1)$ și $G_2 = (X_2, T_2)$, citite din două fișiere text. Să se scrie un program care să verifice dacă G_2 este subgraf pentru G_1 .

15. La fel ca la problema anterioară numai că se va verifica dacă G_2 este graf parțial G_1 .

16. Fie graful orientat $G = (X, T)$ dat prin matricea de adiacență. Fie $\alpha \in X$. Se cere să se listeze multimea arcelor care au o extremitate într-un nod din $X - \alpha$ și altă extremitate într-un nod din $X - \alpha$.

17. Fie graful orientat $G = (X, T)$ dat prin liste de adiacență. Fie $\alpha \in X$. Se cere să se listeze multimea arcelor care au o extremitate într-un nod din $X - \alpha$ și altă extremitate într-un nod din α .

18. Mai multe orașe sunt legate prin autostrăzi cu sens unic. Nu toate orașele sunt legate între ele prin legătură directă. Fiind dat orașul în care se află un turist cu mașina sa, se cer următoarele:

a) Lista orașelor unde turistul poate ajunge;

b) Lista orașelor unde turistul poate ajunge, dar se poate și întoarce.

Datele se citesc dintr-un fișier text, organizat astfel:

- Pe prima linie se găseste numărul orașelor n și orașul în care se află turistul, α ;
- Pe fiecare linie i , din următoarele m linii, se găsesc orașele la care se poate ajunge din α printr-o autostradă care nu mai trece prin alt oraș.

19*. Generarea fișierelor text. În general, problemele cu grafuri au un număr mare de date de intrare. De regulă, datele de intrare se găsesc în fișiere text. Crearea fișierelor text nu se face de la tastatură ci aleator, cu ajutorul unor programe. Programul dv. va crea un fișier text, numit **Grafuri.txt**, pentru un graf orientat cu n noduri în care m este pe prima linie, iar fiecare dintre următoarele m linii, conține câte un arc sub forma $i-j$. Valorile n și m sunt citite de la tastatură și un arc este scris o singură dată. Programul nu va genera arce de tip $i-i$. Este sarcina celui care utilizează programul să introducă pe n și m astfel încât $m \leq n^2 - n$. De ce?

20**. Scrieți un program care generează un graf orientat cu n noduri și muchii și care nu conține circuite.

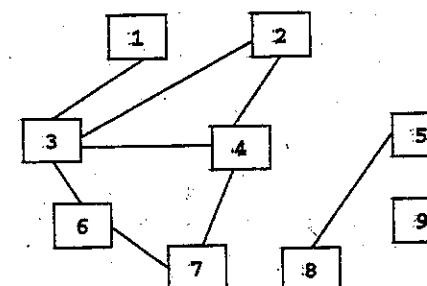
21**. Într-o fabrică există n secții: o secție de plecare, $n-2$ secții intermediare și o secție unde sosesc produsele finite. Fabricii își se poate

asocia un graf orientat în care nodurile sunt secțiile fabricii, iar muchiile sunt traseele pe care produsele intermediare circulă prin secții. O secție nu poate furniza produsul intermediar în care este specializată până când nu îl sosesc toate "ingredientele" de la secțiile de care ea depinde. De asemenea, drumul între două secții durează un anumit interval de timp. Cînd începe i (numărul de secții) și o mulțime de tripleți (i, j, k) unde j este timpul necesar produselor pentru a ajunge din secția i în secția k , să se tipărească timpul necesar pentru fabricarea produsului finit din momentul în care secția de plecare începe să funcționeze. Notă. Datele se introduc în mod corect (nu este necesară validarea lor).

Indicație. Graful nu prezintă circuite. Dacă, prin absurd, o secție ar trimite "ingrediente" către o altă de la care a primit "ingrediente", înțînd cont de faptul că întreaga fabrică începe să funcționeze cu secția 1, fabricația n-ar putea să aibă loc: secția va aștepta "ingrediente" care nu sosesc pentru că ea n-a trimis înapoi "ingrediente" necesare la secția de unde le așteaptă. În aceste condiții se poate aplica Roy Floyd pentru a afla drumul maxim de la secția 1 la secția n .

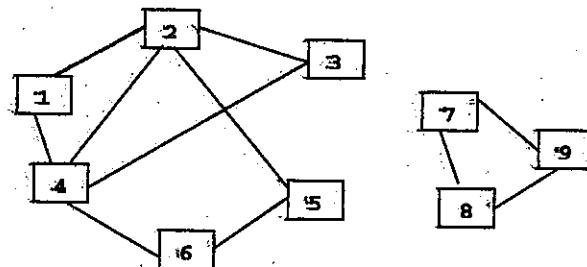
b) Grafuri neorientate

1. Pentru graful neorientat de mai jos, stabiliți care dintre afirmațiile următoare sunt adevărate?



- Graful are 9 noduri;
- Graful are 6 muchii;
- Nodurile 6 și 4 sunt adiacente;
- $d(7)=2$;
- gradul nodului 7 este 6;
- Nodurile 5 și 8 sunt izolate;
- $1, 2, 3$ este lanț;
- $1, 3, 4, 2, 3, 6$ este un lanț elementar;
- Graful are 2 componente conexe și un nod izolat.

2. Pentru graful următor:



- a) Indicați o modalitate de parcursare în lătime pornind de la nodul 4. Aceeași problemă dacă se pomeneste de la nodul 7.
 - b) La fel ca la a) numai că se cere parcursarea în adâncime.
 - c) Care este matricea de adiacență? Este ea simetrică?
 - d) Arătați modul în care se reprezintă graful prin utilizarea listelor de adiacență.
 - e) Care este matricea drumurilor?
 - f) Care sunt componentele conexe?
 - g) Care este matricea drumurilor?
 - h) Este graful conex?
3. Se citește un graf. Să se afiseze toate nodurile care au gradul maxim.
4. Să se scrie o funcție care, pentru un graf dat, transformă matricea de adiacență a unui graf în liste de adiacente.
5. Să se scrie o funcție care, pentru un graf dat, transformă listele de adiacență în matrice de adiacență.
6. Se dă un graf memorat prin matricea de adiacență și un nod al său, x . Se cere să se parcurgă graful în lătime, pornind de la nodul x . Algoritmul va utiliza coada creată ca listă liniară simplu înălțuită.
7. Se dă un graf memorat prin liste de adiacență și un nod al său, x . Se cere să se parcurgă graful în adâncime, pornind de la nodul x . Algoritmul va utiliza stiva creată ca listă liniară simplu înălțuită.
8. Se dă un graf memorat sub forma matricei de adiacență. Se cere să se afiseze matricea drumurilor. Algoritmul va utiliza parcursarea în adâncime.
9. Fieind dată matricea drumurilor unui graf, se cere să se scrie programul care afisează componentele convexe.

10. Scrieți un program care citește un graf dintr-un fișier text și afisează "Da" dacă graful are cel puțin un ciclu și "Nu" în caz contrar.

11. Fie graful orientat $G = (X, T)$ dat prin matricea de adiacență. Fie $A, B \subseteq X$, $A \cap B = \emptyset$. Se cere să se listeze multimea muchilor care au o extremitate în A și alta în B .

12. Partiția determinată de o relație de echivalență. Se consideră o mulțime A . O relație \sim care către elementele acestei mulțimi este o relație de echivalență dacă respectă următoarele trei condiții:

- oricare ar fi $x \in A$, $x \sim x$ (x este echivalent cu x), proprietate numită reflexivitate;
- oricare ar fi $x, y \in A$, din $x \sim y$, rezultă $y \sim x$, proprietate numită simetrie;
- oricare ar fi $x, y, z \in A$, din $x \sim y$ și $y \sim z$, rezultă $x \sim z$, proprietate numită tranzitivitate.

Se citește o mulțime de numere între 0 și 255 prin citirea sa a n perechi (x, y) de numere de acest tip. Printr-o astfel de pereche se înțelege că x este echivalent cu y . Se cere să se determine partitia generată de relația de echivalență considerată pe mulțime.

Exemplu: citim $(1, 2)$, $(4, 5)$, $(2, 3)$, $(6, 7)$, $(7, 1)$.

Se obține partitia: $\{1, 2, 3, 6, 7\}$ $\{4, 5\}$ a mulțimii $\{1, 2, \dots, 7\}$.

Indicație: Componente conexe într-un graf neorientat.

13. Se dau n puncte distințe în plan: $P_i(x_i, y_i)$ cu $0 \leq x_i, y_i \leq 200$, pentru orice $i=1, 2, \dots, n$. Considerăm că fiecare punct este unit cu cel mai apropiat punct diferit de el (dacă există mai multe puncte la distanță minimă, se unește cu fiecare dintre acestea). Numim regiune o mulțime maximală de puncte cu proprietatea că oricare dintre ele sunt unite printr-un lanț. Să se determine numărul de regiuni și să se vizualizeze regiunile (punctele și legăturile dintre ele). (O.N.I. 1993)

14*. Se dă harta legăturilor directe între n orașe, în care se cunoaște distanța între ele și p orașe dintre cele n sub forma unui fișier text, organizat astfel:

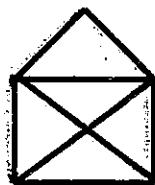
- pe prima linie se găsește numărul oraselor n și un număr natural, k .
- fiecare linie din următoarele k linii, conține un triplet de forma: i, j, d , cu semnificația orașele i și j sunt unite printr-o sosea cu d kilometri.

- pe ultima linie se găsește un număr natural n și $\omega_1, \omega_2, \dots, \omega_n$, numere naturale cuprinse între 1 și m .

O editură are depozite de carte în orașele $\omega_1, \omega_2, \dots, \omega_n$. Un difuzor de carte aflat în orașul C , (C este citit de la tastatură) solicită un număr de cărți. Care este orașul în care există depozitul cel mai apropiat de C , de unde acestea se pot expedia? În plus, se cere să precizezi prin ce orașe vor trece cărțile în drumul lor către difuzor?

Indicație: Se poate utiliza algoritmul lui Dijkstra pentru a afla distanțele minime de la C la toate celelalte orașe. Orașul din care se expediază cărțile va fi unul dintre cele care au depozite, aflat la distanță minimă de C .

- 15*. Se consideră figura alăturată. Se cere să se traseze figura neridicând creionul de pe hârtie (este posibil).



- 16*. Problema anterioară contravine teoremei studiate referitoare la grafuri eulieriene?

- 17*. Plecați la munte și ajungeti cu trenul la o cabană aflată la baza muntelui. Fie ea cabana c_1 . Pe munte sunt n cabane dintre care unele sunt unite prin trasee directe ($c_i \in \{1, 2, \dots, n\}$). Dv. dorîți să parcurgeți toate traseele dintre cabane în astă fel încât fiecare traseu să fie parcurs o singură dată și să vă vă întoarceti în cabana de unde ati plecat. Dacă este posibil, indicați o modalitate de efectuare a traseelor. Datele de intrare se citesc din fisierul text **cabane.txt**, care este organizat astfel:

-linia 1: $n c_1$

-pe fiecare linie i , dintre următoarele, se indică un drum direct între două cabane sub forma **cabana_i cabana_j**.

18*. Autoinstruire

- 18-1. Fiind dată o matrice cu m linii și n coloane, numerotăm, pe linii elementele matricei, așa cum se vede în exemplul următor, unde $m=2$ și $n=3$.

1	2	3
4	5	6

Puteti găsi o formulă prin care, pornind de la linia și coloana unui element, să obtineți numărul său de ordine? Invers, dacă se pornește de la numărul de ordine al unui element, să se obțină linia și coloana în care se găsește elementul respectiv.

Răspuns. Pornind de la linia i și coloana j se obține numărul de ordine prin relația $(i, j) \rightarrow j + (i-1)*n$. Invers, Pornind de la numărul de ordine k vom avea: $i = 1 + (k-1) \text{ div } n$, $j = 1 + (k-1) \text{ mod } n$.

Exemplu: $(2, 3) \rightarrow 3 + (2-1)*3 = 6$. Linia în care se găsește elementul cu numărul de ordine 6 este $1 + (6-1) \text{ div } 3 = 2$, iar coloana este $1 + (6-1) \text{ mod } 3 = 3$.

- 18-2. Labirint. Se consideră un labirint, sub formă de matrice cu m linii și n coloane, citit de program dintr-un fisier text:

```

18
*oooooooo
**o**o**
*ooo**o**
ooooo*** 
*****oooo
oo*o**** 
ooooooooo
o*o**o*o*

```

Prima linie reține numărul de linii (coloane). O cameră este reprezentată prin:

* - camera este ocupată;
o - camera este liberă.

Două camere ale labirintului sunt învecinate dacă amândouă rețin o și au o muchie comună. Se cere ca labirintul să fie transformat într-un graf neorientat prin respectarea următoarelor condiții:

- Nodurile sunt numerotate cu numere de ordine, 1, 2, ..., $m \cdot n$ exact ca în problema anterioară.
- Dacă două camere sunt învecinate, atunci cele două noduri sunt adiacente.
- Graful va fi obținut prin *matricea ponderilor forma 1*, în care orice muchie are ponderea 1.

Indicație. Matricea ponderilor va avea m^2 linii și n^2 coloane.

- 18-3. Pentru problema 18-2 se consideră o persoană care se găsește într-o din camerele labirintului de coordonate l_1, c_1 . Se cere cel mai scurt drum al persoanei respective până în camera de coordonate l_f, c_f . Coordonatele celor două camere se citesc de la tastatură.

Indicație. Labirintul se transformă în graf. Vom utiliza matricea ponderilor pentru a afla drumurile minime de la camera de coordonate l_1, c_1 (de fapt nodul corespunzător) la toate celelalte noduri. Se poate utiliza Roy Floyd sau Dijkstra. După care se listează drumul obținut astă cum am învățat.

18-4. Pe o tablă de sah de dimensiuni $m \times n$ se poate deplasa un nebun conform regulilor obisnuite ale sahului. În plus, pe tablă se pot afla obstacole la diferite coordonate; nebunul nu poate trece peste aceste obstacole. Să se indice dacă există un drum între două puncte $A(x_1, y_1)$ și $B(x_2, y_2)$ de pe tablă și, în caz afirmativ, să se tipărească numărul minim de mutări necesare. Se citesc: n, x_1, y_1, x_2, y_2 , apoi perechi de coordonate ale obstacolelor.

Indicatie. La fel ca la 18-3. Diferența este în construcția matricei ponderelor. Vecinii unui nod sunt toate nodurile în care poate ajunge un nebun care se află în nodul respectiv.

18-5. La fel ca la 18-4, numai că în loc de nebun avem un cal.

18-6. La fel ca la 18-4, numai că în loc de nebun avem o tură.

18-7. La fel ca la 18-4, numai că în loc de nebun avem o regină.

18-8. La fel ca la problema 18-3, numai că se cer coordonatele tuturor camerelor în care poate ajunge persoana respectivă.

18-9. Algoritmul lui Lee. Problema 18-3 se poate rezolva și într-o altă manieră, utilizând algoritmul lui Lee. Ideea este simplă: se marchează cu 1 toti vecinii nodului initial, apoi cu 2 toti vecinii nemarcați ai vecinilor marcați cu 1 și.m.d. În final se listează drumul cu ajutorul funcției `Drum`. Nodul destinație este marcat cu o anumită valoare, `val`, care reprezintă distanța, în noduri, de la nodul initial la el. Se căută primul nod pentru care există drum de la el la nodul final și este marcat cu `val-1`. Procedeul se teia pentru acesta.

18-10. Doi îndrăgostiți numiți R și S sunt pedepsiți pentru faptul că nu au obținut rezultate satisfăcătoare în procesul de învățământ și în activitatea practică, rezultate atât de necesare unei societăți aflate în plin proces de tranziție către Economia de Piată. Prin urmare, sunt închisi într-un labirint L codificat ca o matrice binară cu m linii și n coloane. Elementele $L(i,j)$ sunt camere. Codificarea labirintului se face în felul următor: $L(i,j)=1$ dacă acea cameră este accesibilă și 0 în caz contrar. Dacă unul dintre îndrăgostiți se află într-o cameră, acesta poate intra în oricare dintre camerele situate la nord, sud, est, vest, cu condiția ca aceasta să fie accesibilă. Fiind date camerele în care se găsesc R și S , se cere să se determine timpul minim și traseele urmate de cei doi astfel încât, în final, amândoi să se găsească în aceeași cameră. În fiecare unitate de timp, oricare din îndrăgostiți poate intra într-o cameră învecinată accesibilă sau poate rămâne pe loc.

O.N.I 96

Indicatie. Se poate utiliza algoritmul lui Lee (18-9) pentru găsirea drumului minim, apoi, cei doi se deplasează unul către altul pe drumul minim.

19.** Sortare în limita posibilităților. Se consideră că într-un vector V cu m componente se pot inversa numai conținuturile anumitor componente dintre cele m . Opareche de componente de indice i și j ale căror conținuturi se pot inversa este dată de perechea i și j . Fiind date m astfel de perechi și știind că vectorul conține numerele $1, 2, \dots, m$ într-o ordine oarecare, se cere să se sorteze înversează numai conținuturile componentelor care se pot inversa (care sunt perechi dintre cele m). Dacă sortarea este posibilă se vor afisa indicii componentelor care se inversează, iar dacă sortarea nu este posibilă se afisează `Nu`. Datele de intrare se găsesc în fisierul text `date.in` astfel:

Linia 1: n

Linia 2: $1, \dots, n$ într-o ordine oarecare.

Linia 3: m

urmatoarele m linii conțin fiecare căte o pereche de indici i, j .

Exemplu:

Programul va afisa:

1 2

2 3

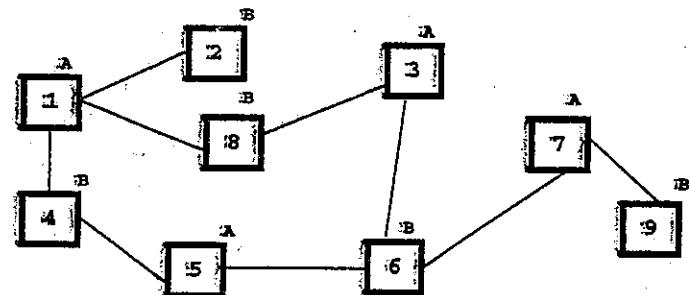
Lugoj 2000

Indicatie. Asociem problemei un graf neorientat. Nodurile sunt indicii elementelor vectorului, de la 1 la n . Când conținuturile a două elemente se pot inversa, nodurile corespunzătoare sunt unite printr-o muchie. Dacă nodurile i_1, i_2, \dots, i_k sunt unite printr-un drum: atunci interschimbările $[i_1, i_2], [i_2, i_3], \dots, [i_{k-1}, i_k], [i_{k-1}, i_k], \dots, [i_2, i_1]$ inversează conținuturile elementelor de indice i_1 și i_k , lăsând conținuturile celorlalte elemente de indici i_2, \dots, i_{k-1} nemodificate. Algoritmul este următorul:

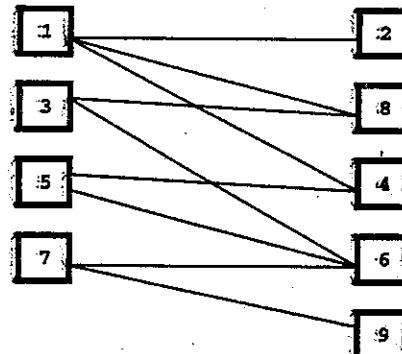
- Se determină pozitia pe care se găsește elementul 1. Dacă este diferită de 1, fie ea i , se căută un drum de la 1 la i . Dacă nu există, sortarea nu se poate efectua, iar dacă există, se inversează astă cum am arătat anterior.
- La fel pentru 2, 3, ..., n .

20.** Graful neorientat $G=(X, \Gamma)$ este bipartit dacă există o parte a mulțimii X în două clase A și B astfel încât două vîrfuri din aceeași clasă să nu fie vecinătate. Să se scrie un program care verifică dacă un graf este bipartit și în caz afirmativ, să tipărească mulțimile A și B . Un graf bipartit se mai notează și prin $G=(A, B, \Gamma)$.

Indicație. Vom exemplifica algoritmul pentru o componentă conexă maximală și să, după care vom generaliza algoritmul pentru tot graful.



În componentă conexă de mai sus - a fost reprezentată doar ea - pornim de la un nod oarecare și marcăm nodurile cu **A** și **B**. Dacă se poate efectua un astfel de marcată încât orice nod să fie marcată fie cu **A** fie cu **B** înseamnă că am reușit să descompunem componentă conexă respectivă ca un graf bipartit, așa cum se vede alăturat.



Evident, o astfel de marcare este posibilă dacă graful nu conține cicluri care trec printr-un număr impar de muchii!

- ✓ Dacă există cicluri care trec printr-un număr impar de muchii, graful nu este bipartit. Retineți acest lucru!
- ✓ Pentru celelalte componente conexe se procedează în mod analog!

. Lucrare în echipă. Se dorește scrierea unei aplicații de informare a călătorilor privind transportul în comun într-un oraș. Se cunosc cele **n stații de autobuz din orașul respectiv. De asemenea, se știe traseul a **k** linii de autobuz (stațiile prin care acestea trec). Se cere ca aplicația să furnizeze modul în care o persoană se poate deplasa cu autobuzul între două stații date, în ipotezele:

- În număr minim de stații;
- Prin utilizarea unui număr minim de linii de autobuz.

Este sarcina dv. să organizați intrările și ieșirile de date.

Capitolul 4

Arbore și arborescente

4.1 Arboori

4.1.1 Noțiunea de arbore

Definiție. Se numește arbore un graf neorientat care este conex și nu conține cicluri.

Problema. Se citește un graf. Să se scrie un program care verifică dacă este arbore.

1. Mai întâi, trebuie văzut dacă graful este conex. În cazul grafurilor orientate această problemă se rezolvă printr-o simplă parcurgere în adâncime (DF). Dar în cazul grafurilor neorientate? Aici apare o problemă: De la nodul **i** la nodul **j** există două arce, de la **i** la **j** și de la **j** la **i**. Aceasta conduce la semnalarea unui ciclu fals. Pentru rezolvare, după alegerea unei muchii, de exemplu de la **i** la **j** se elimină muchia de la **j** la **i**. În final, dacă au fost atinse toate nodurile (adică dacă orice componentă a vectorului **s** refițe numai 1) înseamnă că graful este conex.

2. Trebuie analizat dacă graful nu are cicluri. Această problemă se rezolvă tot cu ajutorul parcurgerii în adâncime. Parcurgerea asigură selectia unei muchii o singură dată. Dacă graful are cel puțin un ciclu, atunci un nod este atins de două ori. Cum ne putem da seama dacă aceasta este loc? Simplu, dacă a fost atins un nod **i**, pentru care **s[i]=1**.

În aceste condiții, nu se face decât o simplă parcurgere în adâncime. Iată programul:

```
#include "grafuri.cpp"
int s[50], A[50][50], gasit, n, i, suma;

void df_x(int nod)
{
    int k;
    s[nod]=1;
    for(k=1; k<=n; k++)
        if (A[nod][k]==1)
    {
        A[k][nod]=0;
        if (s[k]==0) df_x(k);
        else gasit=1;
    }
}
```

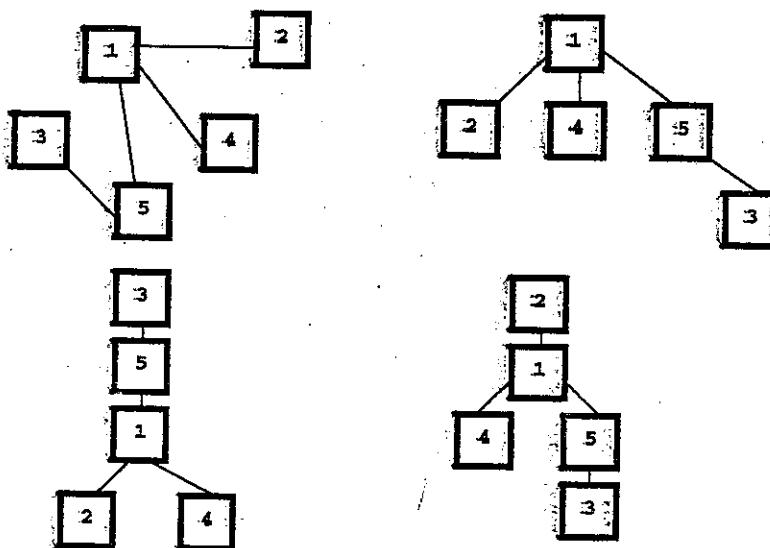
```

main()
{
    CitireG("Graf.txt", A, n);
    df_x(1);
    suma=0;
    for (i=1; i<=n; i++) suma+=s[i];
    if (suma!=n) cout << "Graful nu este conex" << endl;
    else
        if (gasit) cout << "Graful are cel putin un ciclu" << endl;
        else cout << "este arbore";
}

```

- ✓ Complexitatea algoritmului este cea a parcurgerii în adâncime: $O(m)$, unde m reprezintă numărul de muchii.

Mai jos este reprezentat grafic același arbore în mai multe feluri. Ultimele trei reprezentări sunt făcute considerând, pe rând, ca vârf al arborelui nodurile 1, 3, 2. De asemenea, ultimele trei reprezentări justifică și denumirea dată grafurilor conexe și fără cicluri, cea de arbori.



Teoremă. Fie G un graf neorientat cu n noduri. G este arbore dacă și numai dacă are $n-1$ muchii și nu contine cicluri.

Demonstratie:

⇒ Vom demonstra prin inducție. Dacă $n=1$, numărul muchiilor este 0 (se verifică). Vom presupune proprietatea adevărată pentru arbori cu n noduri

(vor avea $n-1$ muchii). Fie un arbore cu $n+1$ noduri. Există cel puțin un nod terminal (nod care are o singură muchie incidentă). Dacă nu ar exista un astfel de nod, arborele ar contine cicluri (se contrazice definiția). Eliminăm nodul terminal și muchia care îl este incidentă. Obținem un arbore cu n noduri. Conform ipotezelii făcute, acesta va avea $n-1$ muchii. Înseamnă că arborele cu $n+1$ noduri va avea n muchii ($n-1+1$)..

⇐ Fie G un graf cu $n-1$ muchii, care nu contine cicluri. Rămâne de dovedit că G este conex. Vom demonstra prin reducere la absurd. Presupunem că G nu este conex. Fie G_1, G_2, \dots, G_p componentele conexe. Fiecare dintre ele îndeplinește condițiile:

- 1) este conexă (așa a fost aleasă);
- 2) nu contine cicluri (pentru că G nu contine cicluri).

Rezultă că fiecare dintre ele este arbore. Fie m_1 numărul muchiilor și n_1 numărul nodurilor fiecărui arbore G_1 . Avem $m_1=n_1-1$. Dar $m_1+m_2+\dots+m_p=n-1$. Rezultă: $n_1-1+n_2-1+\dots+n_p-1=n-1$, deci $n_1+n_2+\dots+n_p=n+p-1$. Dar G are n noduri. Rezultă: $n+p-1=n$, deci $p=1$. În concluzie, există o singură componentă conexă, care nu contine cicluri. Deci G este arbore.

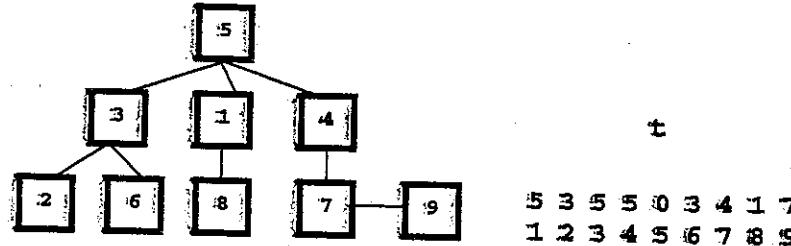
- ✓ Din demonstrație s-a văzut că un graf neorientat fără cicluri, dar neconex este alcătuit, din mai mulți arbori. Din acest motiv, un astfel de graf se numește pădure!
- ✓ Tot în această demonstrație, a fost introdus termenul de nod terminal al unui arbore, un nod cu o singură muchie incidentă.
- ✓ Ati reținut faptul că arboarele poate fi privit ca având un anumit vârf! Vârf al arborelui poate fi oricare nod al său, deci și unul terminal!

4.1.2 Metode de memorare a arborilor

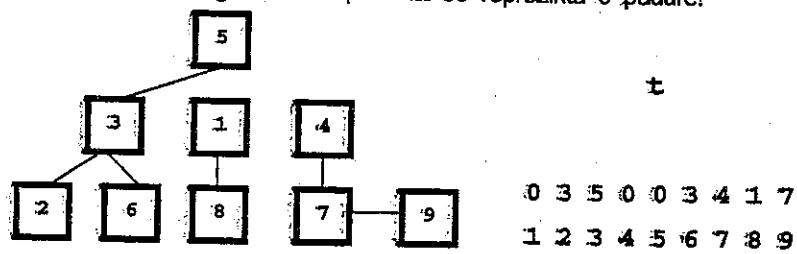
1. Aşa cum am arătat, un arbore este un graf neorientat, cu anumite proprietăți. Aceasta înseamnă că el poate fi reprezentat ca un graf. De aici rezultă că pentru reprezentarea unui arbore se pot utiliza:
 - matricea de adiacență;
 - liste de adiacențe.
2. O a doua formă de reprezentare a arborilor este legătura de tip TATA. Arboarele se reprezintă sub forma unui vector t cu n componente: dacă

$t[i]=k$, atunci nodul i este descendenter al nodului k . Dacă nodul i este vârf, $t[i]=0$.

Exemplu: arborele de mai jos este reprezentat alăturat:



- ✓ Legătura de tip TATA se mai numește și legătură cu referințe ascendentice.
- ✓ În mod evident, legătura de tip TATA este determinată și de nodul ales ca vârf. Dacă, de exemplu, pentru același arbore vârful este 9, reprezentarea este alta. Este ca și cum "apucăm" arborele de un nod și celelalte cad!
- ✓ De fapt, prin legătura de tip TATA se reprezintă o pădure!



✓ Pe această reprezentare am mai întâlnit-o! De exemplu, la algoritmul lui Dijkstra! Dar ce altceva, decât arbore, erau drumurile de la un nod la toate celelalte?

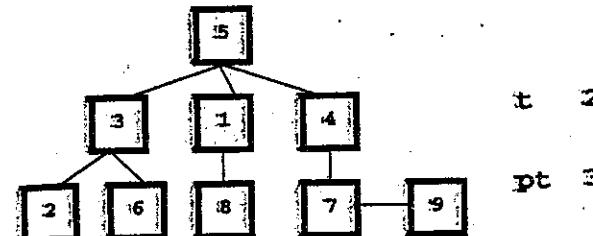
3. Codul lui Prüffer*. Pentru a reprezenta un arbore carecăre se pot utiliza doi vectori, pe care-i vom numi t - de la nod terminal - și pt - de la părinte nod terminal. Modul în care aceștia rețin datele este explicitat mai jos:

- se caută cel mai mic nod terminal, acesta este reținut în vectorul t , iar părintele său este memorat în vectorul pt .
- se obține un nou arbore, renunțând la nodul terminal și la muchia care-l unește de părinte.

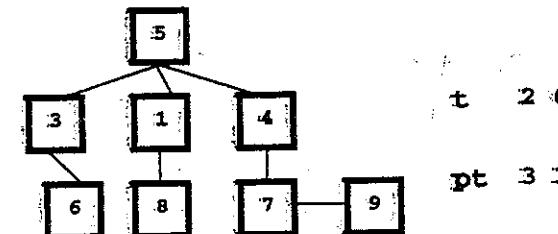
➤ Procedeu se reia până când rămâne un arbore cu un singur nod.

Exemplu:

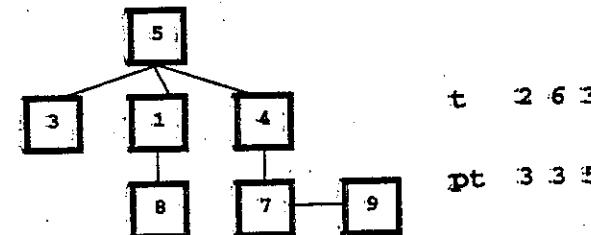
Pentru arborele alăturat cel mai mic nod terminal este 2, iar părintele este 3.



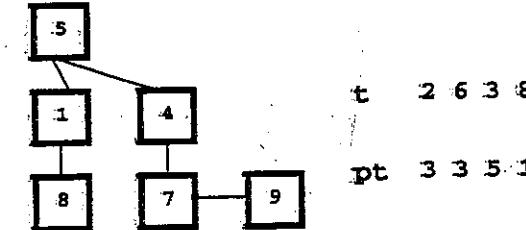
În nouă arbore, cel mai mic nod terminal este 6, iar părintele lui 6 este 3.



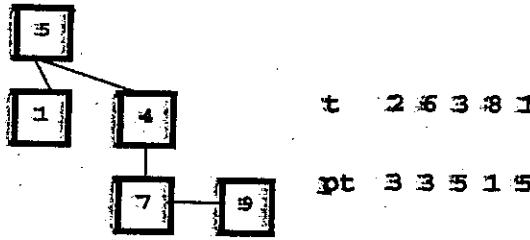
Cel mai mic nod terminal este 3, iar părintele 5.



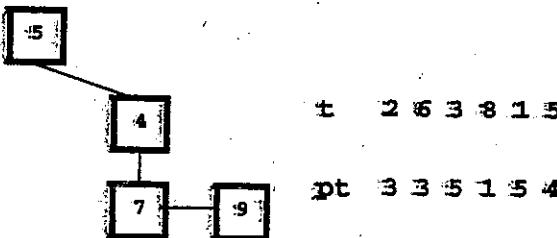
Cel mai mic nod terminal este 8, iar părintele său este 1.



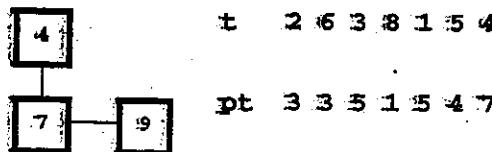
Cel mai mic nod terminal este 1, iar părintele său este 5.



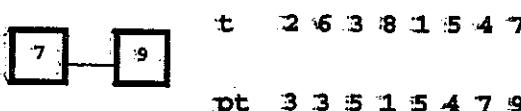
Cel mai mic nod terminal este 5, iar părintele său 4.



Cel mai mic nod terminal este 4, iar părintele său 7.



Cel mai mic nod terminal este 7, iar părintele său este 9.



Rămâne numai un arbore cu un singur nod, 9.

\Rightarrow Până în acest moment am memorat arborele prin utilizarea a doi vectori t și pt . Fiecare dintre ei, are $n-1$ componente!

\Rightarrow Prin logica algoritmului, nodul care rămâne este cel mai mare, adică nodul n . Aceasta este ultimul memorat în vectorul pt . Prin urmare, dacă se cunoaște n , sunt suficiente $n-2$ componente pentru vectorul pt .

\Rightarrow Foarte important! Vectorul t se poate reconstitui doar dacă cunoaștem vectorul pt . Aceasta înseamnă că un arbore cu n noduri a fost memorat doar printr-un vector cu $n-2$ componente!

Pentru reconstituirea vectorului t , pornind de la pt , vom începe prin să memorați în pt , numărul n , pentru componenta $n-1$. Vectorul t se reconstituie prin formula:

$$\forall i \in \{1, 2, \dots, n-1\}$$

$$t[i] = \min_{\substack{k | k \notin \{t[1], t[2], \dots, t[i-1], pt[i], pt[i+1], \dots, pt[n-1]\} \\ k \in \{1, 2, \dots, n\}}}$$

Cum s-a dedus formula de mai sus?

- dacă numărul se găsește în $pt[i]$, $pt[i+1]$... $pt[n-1]$ înseamnă că nodul respectiv nu a fost eliminat până la pasul i ; pentru că este părinte.
- dacă numărul se găsește în $t[1]$, $t[2]$, ..., $t[i-1]$ înseamnă că a fost extras la unul din pașii anterioari.
- dintre nodurile care nu se găsesc ca mai sus, la pasul i a fost extras nodul minim.

Exemplu. Plecăm de la $n=9$, și $pt=\{3 \ 3 \ 5 \ 1 \ 5 \ 4 \ 7\}$

Pasul 1.

```
t={}
pt={3 3 5 1 5 4 7,9}
```

Cel mai mic număr care nu se găsește în pt și este între 1 și 9 este 2.

Pasul 2.

```
t={2}
pt={3 3 5 1 5 4 7,9}
```

Se alege 6.

Pasul 3.

```
t={2 6}
pt={3 3 5 1 5 4 7,9}
```

Se alege 3.

...

Programul care urmează să citește de la tastatură n și PT , după care tipărește vectorul T .

```
#include "grafuri.cpp"
int T[50], PT[50], i, j, k, n, gasit;
main()
{
    cout << "n="; cin >> n;
    for (i=1; i<=n-2; i++)
    {
        cout << "PT[" << i << "]=";
        cin >> PT[i];
    }
    PT[n-1] = n;
    k = 1;
    for (i=1; i<=n-1; i++)
    {
        k = 1;
        do
        {
            gasit = 0;
            for (j=1; j<=i-1; j++)
            if (T[j] == k) gasit = 1;
            if (!gasit)
            for (j=i; j<=n-1; j++)
            if (PT[j] == k) gasit = 1;
            if (gasit) k++;
        } while (gasit);
        T[i] = k;
    }
    for (i=1; i<=n-1; i++) cout << T[i] << " ";
}
```

- ✓ Foarte important. Câtă arbori cu n noduri există? Vectorul PT are $n-2$ componente și fiecare componentă poate retine valori între 1 și n . Deci există n^{n-2} arbori cu n noduri!
- ✓ Pominde la un arbore, i se asociază prin codul lui Pruffer un vector cu $n-2$ componente. Se arată ușor că dacă arborii sunt diferiți, codificarea lor este diferită. Tot așa, pominde la un vector cu $n-2$ componente, unde fiecare la valori între 1 și n se ajunge la un arbore. Cu alte cuvinte am definit o funcție f pe multimea arborilor care ia valori în produsul cartezian:

$$(1,2,\dots,n) \times (1,2,\dots,n) \times \dots (1,2,\dots,n)$$

de $n-2$ ori

- ✓ Din cele arătate rezultă că f este bijectivă. Prin urmare, ținând cont de faptul că funcția este definită pe o mulțime finită, numărul de elemente al codomeniului este egal cu numărul de elemente al domeniului de definiție. Aceasta explică modul în care a fost determinat numărul de arbori.
- ✓ Codul lui Pruffer asociază același cod arborelui indiferent de reprezentarea grafică folosită.

PRIM

4.1.3 Arbore parțial de cost minim

Fie $G=(X,T)$ un graf neorientat și conex. Graful este dat prin matricea ponderilor. Prin eliminarea unei muchii din G se obține un graf parțial al lui G . Dacă acesta este arbore, se va numi arbore parțial.

Problema. Definim costul unui arbore parțial ca suma costurilor muchiilor sale. Se cere să se obțină un arbore parțial de cost minim, adică:

- între oricare două noduri să existe un drum;
- suma muchiilor să fie minimă.

O problemă concretă, în care intervine problema enunțată, este cea de conectare orașelor cu cost minim:

- Se dau n orașe precum și costul conectării anumitor perechi de orașe. Se cere să se aleagă acele muchii care asigură existența unui drum între oricare două orașe astfel încât costul total să fie minim.

Pentru rezolvare se folosesc 2 vectori:

⇒ vectorul S cu următoarea semnificație:

$$S[i] = \begin{cases} 0, & \text{daca nodul } i \text{ apartine arborelui deja construit} \\ k, & \text{daca sunt respectate cele două conditii de mai jos:} \end{cases}$$

1. Nodul i nu aparține arborelui deja construit;
2. Muchia de cost minim care unește pe i cu unul din nodurile grafului deja construit este $[i,k]$.

⇒ vectorul T retine pentru fiecare nod care se adaugă arborelui nodul părinte al acestuia (legătura de tip **TATA**, cu ajutorul acestui vector se poate reconstituî arborele);

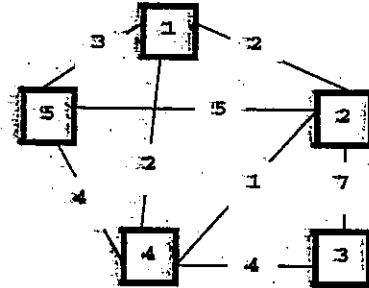
Prezentarea algoritmului:

- 1) Se citește n (numărul nodurilor), matricea ponderilor A și nodul de pornire v . $S[v]=0$; și $S[i]=v$, pentru $i \neq v$
- 2) Se alege muchia de cost minim (i,j) care are o extremitate într-unul din nodurile arborelui deja construit, iar cealaltă într-un nod care nu aparține arborelui ($S[i]=0$, $S(j) \neq 0$). Se pune $T(j)=S[i]$ și se actualizează vectorul S pentru nodul j .

3) Dacă nu au fost alese $n-1$ muchii, se reia pasul 2.

Exemplu. Fie graful din figura următoare. Se porneste cu nodul 1.

$$S=0 \ 1 \ 1 \ 1 \ 1$$



Alegem muchia de cost minim (1,2).

$$S=0 \ 0 \ 2 \ 2 \ 1$$

Pentru nodurile 1 și 2, ca puncte de plecare, se alege muchia de cost minim (2,4).

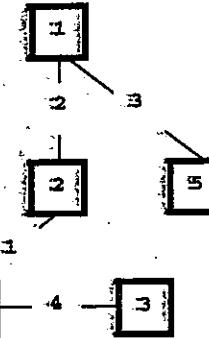
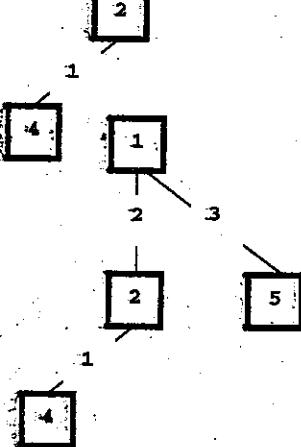
$$S=0 \ 0 \ 4 \ 0 \ 1$$

Pentru nodurile 1, 2 și 4 se alege muchia de cost minim (1,5).

$$S=0 \ 0 \ 4 \ 0 \ 0$$

Pentru nodurile 1, 2, 4 și 5 se alege muchia de cost minim (4,3).

$$S=0 \ 0 \ 0 \ 0 \ 0$$



Am ales 4 muchii.

Continutul final al vectorului S este:

$$0 \ 1 \ 4 \ 2 \ 1$$

✓ Arborele are costul minim 10.

✓ Dacă pornim dintr-un alt nod, este posibil să obținem un alt arbore dar și el va avea costul minim.

Demonstrație. În urma aplicării algoritmului se obține un arbore (vom avea n noduri și $n-1$ muchii).

Fie $G=(x, T)$ arborele parțial obținut în urma aplicării algoritmului și $G_{\min}=(x, T')$ un arbore parțial de cost minim.

Atât G cât și G_{\min} au $n-1$ muchii. Fie $M_1, M_2, \dots, M_{k-1}, M_k$ primele $k \geq 1$ muchii generate de algoritm atunci când a fost obținut G . Fie $\{x_1, x_2, \dots, x_k, x_{k+1}\} \subseteq x$ nodurile din G unite de cele k muchii. Presupunem că M_1, M_2, \dots, M_{k-1} se găsesc și în G_{\min} , și că muchia $M_k \in T$ este prima care nu aparține lui G_{\min} ($M_k \notin T'$). Fie x_1 și x_{k+1} cele două noduri ale lui G unite de muchia M_k . Este evident că, în G_{\min} , cele două noduri nu sunt unite direct (în caz contrar, muchia M_k ar face parte din arborele G_{\min}). Aceasta înseamnă că în G_{\min} nodurile x_1 și x_{k+1} sunt unite printr-un lanț. Acest lanț unește în G_{\min} două noduri, P' și Q' , cu $P' \in \{x_1, x_2, \dots, x_k, x_{k+1}\}$ și $Q' \notin \{x_1, x_2, \dots, x_k, x_{k+1}\}$. În caz contrar, dacă ar uni numai noduri din $\{x_1, x_2, \dots, x_k, x_{k+1}\}$, înseamnă că G ar contine un ciclu (conține muchia M_k , care unește x_1 și x_{k+1} , conține și restul ciclului din G_{\min} , pentru că primele k noduri în G și G_{\min} sunt legate identic). Presupunem nodurile P' și Q' unite prin muchia M . Înlocuim în G_{\min} pe M cu M_k . În urma acestei înlocuiri, G_{\min} se transformă într-un arbore G' . Dar $A(P, Q) \leq A(P', Q')$ pentru că în caz contrar, prin logica algoritmului, s-ar fi ales muchia M . Aceasta înseamnă că arborele parțial G' rămâne tot de cost minim. În plus, G și G' au k muchii care coincid și are costul muchiilor minim. Repetăm raționamentul până se ajunge la coincidența a $n-1$ muchii, suma costului muchiilor fiind aceeași. În concluzie G este arbore parțial de cost minim.

```

#include "grafuri.cpp"
float A[50][50],min,cost;
int s[50],t[50],n,i,j,k,v;
main()
{
    cost=0;
    Citire_cost_N("Graf.txt",A,n);
    cout<<"nodul de pornire ";cin>>v;
    for (i=1;i<=n;i++)
        if (i==v) s[i]=0;
        else      s[i]=v;

    for (k=1;k<=n-1;k++)
    {
        min=PInfinit;
        for (i=1;i<=n;i++)
            if (s[i])
                if (A[s[i]][i]<min)
                    {
                        min=A[s[i]][i];
                        j=i;
                    }
        t[j]=s[j];
        cost+=A[s[j]][j];s[j]=0;
        for (i=1;i<=n;i++)
            if (s[i] && A[i][s[i]]>A[i][j]) s[i]=j;
    }
    cout<<"cost="<<cost<<endl;
    for(i=1;i<=n;i++) cout<<t[i]<<" ";
}

```

✓ Algoritmul are complexitatea $O(n^2)$.

4.2 Arborescente

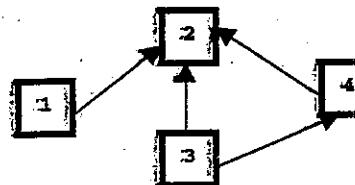
4.2.1 Noțiunea de arborescentă

Noțiunea de arbore poate fi extinsă și pentru grafuri orientate.

Definiție. Fiind date două noduri i și j ale unui graf orientat, se spune că există un lanț de la i la j dacă există o succesiune de arce, indiferent de sensul lor prin care cele două noduri sunt unite.

✓ Dacă există un lanț de la i la j , există și unul de la j la i .

Exemplu:



În graful alăturat există un lanț de la 1 la 4:

[1,2], [4,2]

Definiție. Un graf orientat $G=(X,T)$ este conex dacă $\forall i \neq j \in X$, există un lanț de la i la j .

- ✓ Dacă un graf orientat este **tare conex**, atunci el este **conex**, dar reciproca nu este adevărată. Graful de mai sus este conex.

Definiție. Un graf orientat admite un **ciclu** dacă există un nod i , pentru care există un lanț de la i la i .

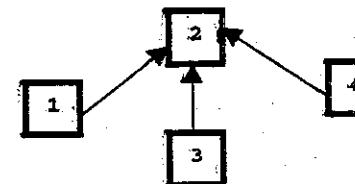
Exemplu. În graful anterior nodurile 4 - 2 - 3 sunt unite printr-un ciclu.

Definiție. Un graf orientat este arbore dacă este conex și nu admite cicluri.

Exemplu:

1. Graful anterior nu este arbore pentru că admite un ciclu.

2.



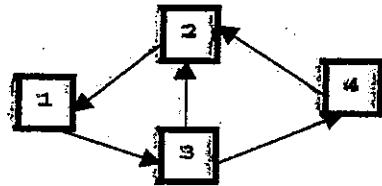
Graful alăturat este un arbore.

Definiție. Un graf orientat $G=(X,T)$ admite o rădăcină $v \in X$ dacă pentru orice $x \in X - \{v\}$ există un drum (atente, nu lanț) de la v la x .

Exemplu:

1. Graful de mai sus nu admite o rădăcină. De exemplu, nu există drum de la 1 la 3 (1 nu este rădăcină), nu există drum de la 3 la 4 (nici 3 nu este rădăcină) etc.

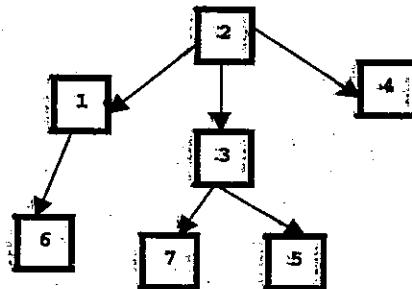
2.



Graful alăturat are rădăcinile 1, 2, 3, 4

Definiție. Se numește arborescentă un graf orientat care îndeplinește simultan două condiții:

1. Este arbore;
 2. Admîne rădăcină.
- ✓ Într-o arborescentă rădăcina este unică. Dacă de exemplu, există două rădăcini v_1 și v_2 atunci există drum de la v_1 la v_2 și de la v_2 la v_1 . Aceasta presupune existența unui ciclu, deci nu este arbore.
 - ✓ De regulă, atunci când se dă o arborescentă se precizează rădăcina sa.



Alăturat este prezentată o arborescentă de rădăcină 2.

- ✓ Se observă că din rădăcină pleacă arce, nu intră. Dacă ar și intra s-ar obține un ciclu.
- ✓ De asemenea, orice nod are un singur predecesor. Dacă, prin absurd, ar avea doi sau mai mulți, s-ar obține un ciclu care include rădăcina.

Problema. Se citește un graf orientat. Acesta este dat prin matricea de adiacență. Se cere ca programul să decidă dacă graful este sau nu o arborescentă. În caz afirmativ, se va tipări rădăcina acesteia.

Rezolvare.

1. Trebuie să decidem mai întâi dacă graful orientat este arbore. Aceasta se face printr-o simplă parcurgere în adâncime, asemănător cu algoritmul aplicat în cazul grafurilor orientate.

2. În cazul când graful este arbore, trebuie să văzut dacă acesta are o rădăcină. Se testează, pe rând, fiecare nod dacă îndeplinește condiția de rădăcină. Fiecare test se face printr-o parcurgere în adâncime pornind de la nodul respectiv.

```
#include "grafuri.cpp"
int A[50][50], B[50][50], s[50], gasit, OK, radacina, m, i, j, suma, x;

void df_r(int nod)
{
    int k; s[nod]=1;
    for (k=1;k<=n;k++)
        if (A[nod][k]==1 || A[k][nod]==1)
            if (A[k][nod]==A[nod][k]==0)
                if (s[k]==0) df_r(k);
            else gasit=1;
    }
}

void df_r1(int nod)
{
    int k; s[nod]=1;
    for (k=1;k<=n;k++)
        if (A[nod][k]==1 && s[k]==0) df_r1(k);
}

main()
{
    Citire("Graf.txt", A, n);
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++) B[i][j]=A[i][j];
    df_r(1);
    for (i=1;i<=n;i++) suma+=s[i];
    if (suma!=n) cout<<"Graful nu este conex "<<endl;
    else cout<<"Graful este conex "<<endl;
    if (gasit) cout<<"Graful are cel putin un ciclu "<<endl;
    else cout<<"Graful nu are cicluri ";
    if (suma==n && !gasit)
        (cout<<"Este arbore "<<endl,
        OK=1);
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++) A[i][j]=B[i][j];
    if (OK)
    {x=1;
    do
        { suma=0; for (i=1;i<=n;i++) s[i]=0;
        df_r1(x);
        for (i=1;i<=n;i++) suma+=s[i];
        if (suma==n)
            { cout<<"radacina este "<<x<<endl<<"este arborescenta";
            radacina=x;
            }
        else x++;
        } while (!radacina && x<=n);
    }
}
```

```

if(!radacina) cout<<"nu are radacina";
}

```

- ✓ Testul de arbore se face în $O(m)$, unde m reprezintă numărul muchiilor. Identificarea rădăcinii se face în $O(n \times m)$, unde n este numărul nodurilor. În concluzie algoritmul are complexitatea $O(n \times m)$.

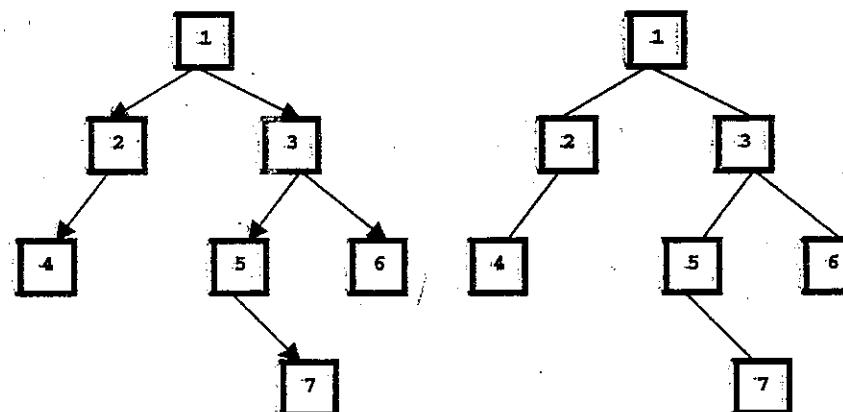
4.2.2 Arborescență binară

4.2.2.1 Noțiunea de arbore binar

Programatorii folosesc, de multe ori, prin abuz de limbaj, termenul de arbore, în loc de arborescență. În astfel de cazuri, se specifică rădăcina, care se mai numește și vârf și se consideră că sensul arcelor este implicit, motiv pentru care nu se mai reprezintă grafic.

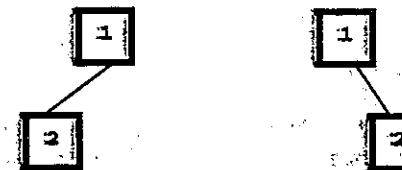
În cazul în care arborescența are cel mult doi descendenti, unul stâng și unul drept, spunem că avem un arbore binar.

Exemplu. Mai jos vedeti o arborescență cu rădăcina 1 care este un arbore binar. Alăturat, observati modul în care se reprezintă grafic, renunțând la sensurile arcelor, care sunt implicate.



- ✓ Se presupune că nodurile sunt așezate pe mai multe niveluri. Radăcina este pe nivelul 0, descendenții direcți ai rădăcinii sunt pe nivelul 1, descendenții direcți ai nodurilor de pe nivelul 1 sunt pe nivelul 2 etc.
✓ Numărul de niveluri mai puțin cel al rădăcinii, ocupate de nodurile grafului, se numește adâncimea arborelui binar.

- ✓ Vom face distincție între "descendentul stâng și cel drept". De exemplu, arborii de mai jos sunt considerați distincți.



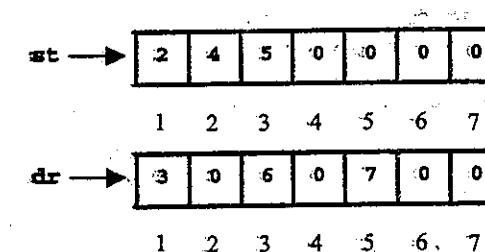
4.2.2.2 Modalități de reprezentare a arborilor binari

Desigur, arborii binari pot fi reprezentați ca orice graf orientat, dar această formă este deosebit de greoale în aplicații. Din acest motiv sunt cu mult mai convenabile reprezentările următoare:

1. Reprezentarea cu ajutorul vectorilor (clasică)

Un arbore binar poate fi reprezentat cu ajutorul a doi vectori, pe care îi vom numi st (de la stânga) și dr (de la dreapta). Pentru fiecare nod i din cele n , $st[i]$ reține numărul de ordine al nodului stâng subordonat de i , iar $dr[i]$ reține numărul de ordine al nodului drept subordonat de i . Dacă nu există nod subordonat, reține 0.

Exemplu. Arborele binar cu 7 noduri și $v=1$ de pe pagina anterioară se reprezintă astfel:



2. Reprezentarea în HEAP (actuală).

Fiecare nod al arborelui este reprezentat în HEAP de o înregistrare cu următoarea structură:

```

struct Nod {
    int info;
    Nod *st, *dr;
}

```

Programul va reține doar adresa rădăcinii (vârfului). În aplicații veți întâlni numeroase exemple în acest sens, motiv pentru care ne oprim aici cu această prezentare.

3. Reprezentarea cu ajutorul legăturii **stata**.

Aceasta va mai fost prezentată în cazul arborilor neorientați. Mentionăm doar faptul că această reprezentare este inconvenientă în majoritatea aplicațiilor.

4.2.2.3 Modalități de parcurgere a arborilor binari

În scopul prelucrării, este necesar ca nodurile arborelui binar să fie vizitate. Există mai multe modalități de parcurgere a arborilor binari care diferă prin ordinea de vizitare a nodurilor.

✓ Putem considera că fiecare nod al arborelui binar subordonează un subarbore binar stâng și un subarbore binar drept. Pe baza acestei observații putem scrie cu ușurință subprograme recursive pentru diversele modalități de parcurgere.

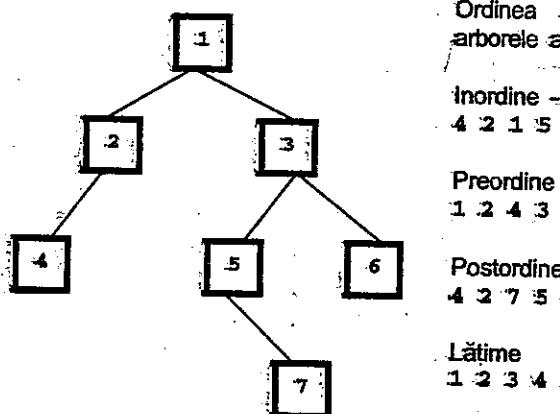
Principalele modalități de parcurgere ale unui arbore binar sunt:

A) Parcurgerea în inordine - **SVD** - se parcurge mai întâi subarborele stâng, apoi vârful, apoi subarborele drept.

B) Parcurgerea în preordine - **VSD** - se parcurge mai întâi vârful, apoi subarborele stâng, apoi subarborele drept.

C) Parcurgerea în postordine - **SDV** - se parcurge mai întâi subarborele stâng, apoi subarborele drept, apoi vârful.

D) Parcurgerea în adâncime - a fost studiată pentru grafuri.



Programul următor citește și parcurge în inordine, postordine și preordine un arbore binar. Arborele este reținut în **HEAP**. Pentru a marca faptul că descendantul unui anumit nod lipseste se introduce **0**.

```

#include <iostream.h>
struct Nod
{
    int nr;
    Nod* st, *dr;
};
Nod* c;
void svd(Nod *c)
{
    if (c)
    {
        svd(c->st);
        cout<<c->nr;
        svd(c->dr);
    }
}
void vsd(Nod *c)
{
    if (c)
    {
        cout<<c->nr;
        vsd(c->st);
        vsd(c->dr);
    }
}
void sdv(Nod *c)
{
    if (c)
    {
        sdv(c->st);
        sdv(c->dr);
        cout<<c->nr;
    }
}
Nod* arb()
{
    int n;
    Nod* c;
    cout<<"n=">>n;
    if (n)
    {
        c= new Nod;
        c->nr=n;
        c->st=arb();
        c->dr=arb();
        return c;
    }
    else return 0;
}
main()
{
    c=arb();
    svd(c);
    cout<<endl;
    svd(c);
    cout<<endl;
    sdv(c);
}
  
```

✓ Nodurile arborelui se introduc în ordinea:

1 2 4 0 0 0 3 5 0 7 0 0 6 0 0

✓ Atât pentru introducerea arborelui, cât și pentru parcurgeri s-au utilizat tehnici recursive care decurg din tehnica **Divide et Impera!**

Programul următor efectuează aceleasi parcurgeri ca și precedentul, numai că graful este reprezentat cu ajutorul vectorilor.

```

#include <iostream.h>
int st[50], dr[50], n, i, v;

void svd(int nod)
{
    if (nod)
    {
        svd(st[nod]);
        svd(dr[nod]);
        cout << nod;
    }
}

void vsd(int nod)
{
    if (nod)
    {
        cout << nod;
        vsd(st[nod]);
        vsd(dr[nod]);
    }
}

void citire()
{
    cout << "n="; cin >> n;
    cout << "v="; cin >> v;
    for (i=1; i<=n; i++)
    {
        cout << "st[" << i << "]=";
        cin >> st[i];
        cout << "dr[" << i << "]=";
        cin >> dr[i];
    }
}

main()
{
    citire();
    cout << "SVD "; svd(v); cout << endl;
    cout << "VSD "; vsd(v); cout << endl;
    cout << "SDV "; svd(v); cout << endl;
}

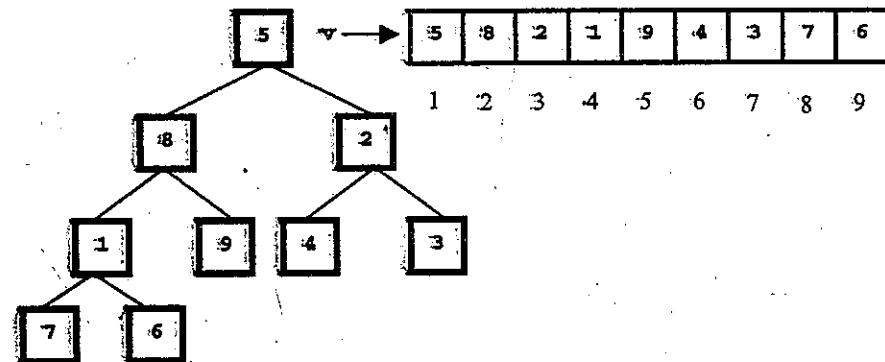
```

4.2.2.4* HeapSort -Facultativ-

Există posibilitatea ca un vector v cu n componente să fie interpretat ca arbore binar. Mecanismul este următorul:

- Rădăcina este etichetată cu $v[1]$.
- nodul etichetat cu $v[i]$ subordonează:
 - la stânga nodul etichetat cu $v[2 \cdot i]$, dacă $2 \cdot i \leq n$;
 - la dreapta nodul etichetat cu $v[2 \cdot i + 1]$, dacă $2 \cdot i + 1 \leq n$;

Mai jos puteți observa un vector și arborele binar reprezentat de el.



✓ Un arbore obținut ca mai sus are $\lceil \log_2 n \rceil$ niveluri.

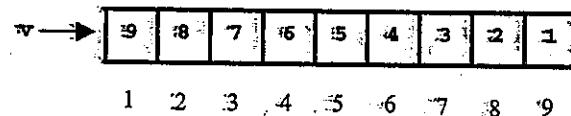
Definiție. Un vector v se numește **MinHeap** dacă:

$$\forall i \in \{1, 2, \dots, \left\lceil \frac{n}{2} \right\rceil\} \Rightarrow \begin{cases} v[i] \leq v[2 \cdot i] \\ v[i] \leq v[2 \cdot i + 1] \text{, dacă } 2 \cdot i + 1 \leq n \end{cases}$$

Definiție. Un vector v se numește **MaxHeap** dacă:

$$\forall i \in \{1, 2, \dots, \left\lceil \frac{n}{2} \right\rceil\} \Rightarrow \begin{cases} v[i] \geq v[2 \cdot i] \\ v[i] \geq v[2 \cdot i + 1] \text{, dacă } 2 \cdot i + 1 \leq n \end{cases}$$

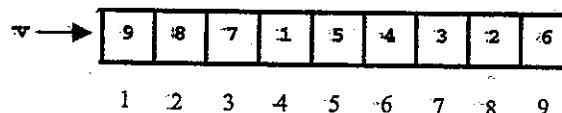
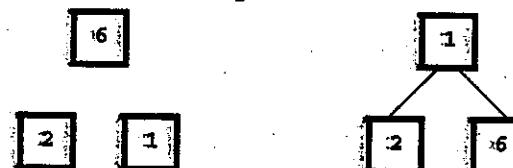
✓ În cazul arborelui reprezentat de un vector de tip **MinHeap** orice nod subordonăza noduri cu etichete mai mari, dar în cazul vectorului de tip **MaxHeap** orice nod subordonăza noduri cu etichete mai mici.
 Problema. Fieind dat un vector v , se cere ca acesta să fie organizat ca **MinHeap**.



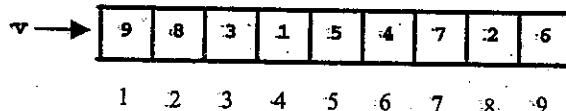
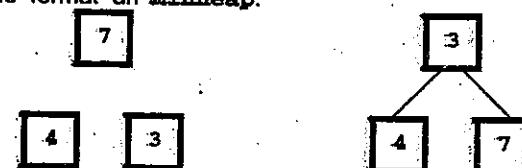
Dacă un vector are o singură componentă, evident, avem un **MinHeap**. O variabilă i va lăua, pe rând, valorile:

$$\left[\frac{n}{2} \right], \left[\frac{n}{2} \right] - 1, \left[\frac{n}{2} \right] - 2, \dots, 1$$

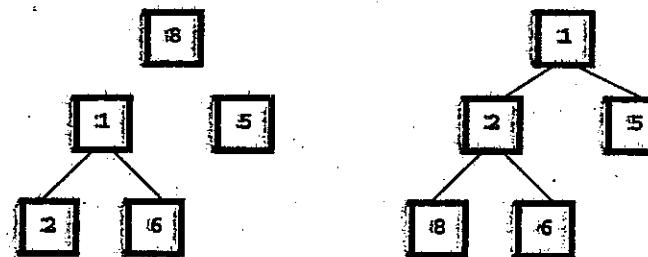
1. $i=4$. $v[4]=6$; $v[8]=2$; $v[9]=1$. Din vârful 16 și **MinHeap-urile** 2 și 1 trebuie format un **Minheap**.



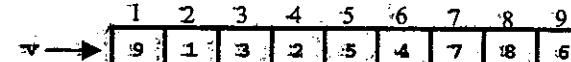
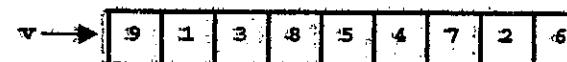
2. $i=3$. $v[3]=7$; $v[6]=4$; $v[4]=3$. Din vârful 7 și **MinHeap-urile** 4 și 3 trebuie format un **Minheap**.



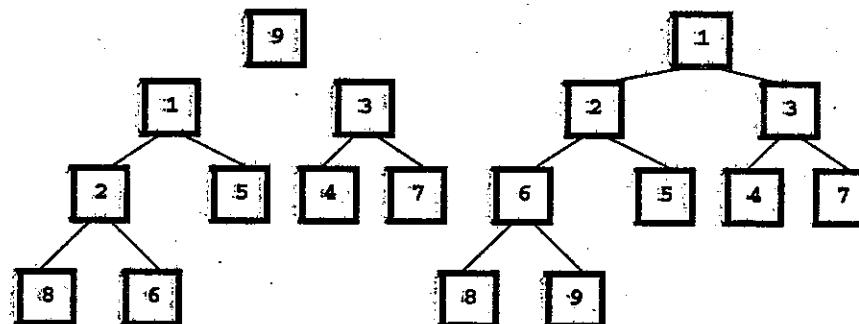
3. $i=2$. $v[2]=8$; $v[4]=1$; $v[5]=5$. Din vârful 8 și **MinHeap-urile** 1 și 5 trebuie format un **Minheap**.



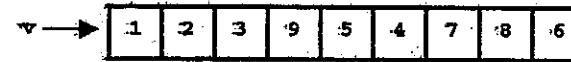
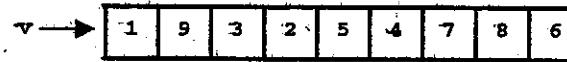
Iată cum se modifică v :



4. $i=1$. $v[1]=9$; $v[2]=1$; $v[3]=3$. Din vârful 8 și **MinHeap-urile** cu vârfurile 1 și 3 trebuie format un **Minheap**.



Iată cum se modifică v :





1 2 3 4 5 6 7 8 9

În acest moment v este organizat ca **MinHeap**.

- ✓ La fiecare pas, trebuie format un **MinHeap** din două **MinHeap**-uri și o valoare. La început valoarea se compară cu vârfurile celor două **MinHeap**-uri, și își schimbă poziția cu cel mai mic dintre ele. Apoi subarboreii subordonati de un vârf al unui **MinHeap** sunt tot **MinHeap**-uri, deci procesul se repetă, fie până la baza arborelui, fie până când valoarea care trebuie inserată este cea mai mică dintre cele cu care se compară.
- ✓ La o astfel de operatie, numărul de comparații este cel mult $\lfloor \log_2 n \rfloor$.
- **HeapSort**. Fiind dat un vector v cu n componente se cere ca acesta să fie sortat descrescător (crescător) prin utilizarea **Heap**-urilor. Vom analiza cazul sortării descrescătoare.
 - ⇒ În cazul în care vectorul se organizează ca **MinHeap**, este evident că prima componentă reține cea mai mică valoare, deci pentru ca vectorul să fie sortat descrescător se organizează ca **MinHeap**.
 - ⇒ Apoi se interschimbă conținuturile componentelor 1 și n . În acest caz, ultima componentă reține valoarea cea mai mare.
 - ⇒ Din prima componentă și **MinHeap**-urile cu vârfurile în 2 și 3 se formează un nou **MinHeap**.
 - ⇒ Se interschimbă conținuturile componentelor 1 și $n-1$.
 - ...

Prezentăm programul care realizează aceasta:

```
#include <iostream.h>
int v[50], n, i;

void Combinare(int vf, int n)
{
    int baza=2*vf, valoare=v[vf], gata=0;
    while (baza<=n && !gata)
```

```

        if (baza<n && v[baza]>v[baza+1]) baza++;
        if (valoare>v[baza])
            { vv[vf]=v[baza]; vf=baza; baza+=2; }
        else gata=1;
    }
    v[vf]=valoare;
}

void Heap()
{
    int i;
    for (i=n/2; i>=1; i--) Combinare(i, n);
}

void Citeste()
{
    cout<<"n"; cin>>n;
    for (i=1; i<=n; i++)
    {
        cout<<"v["<<i<<"]"; cin>>v[i];
    }
}

void HeapSort()
{
    int man;
    Heap();
    for (i=n; i>=2; i--)
    {
        man=v[1]; v[1]=v[i]; v[i]=man;
        Combinare(1, i-1);
    }
}

main()
{
    Citeste();
    HeapSort();
    for (i=1; i<=n; i++) cout<<v[i]<< " ";
}
```

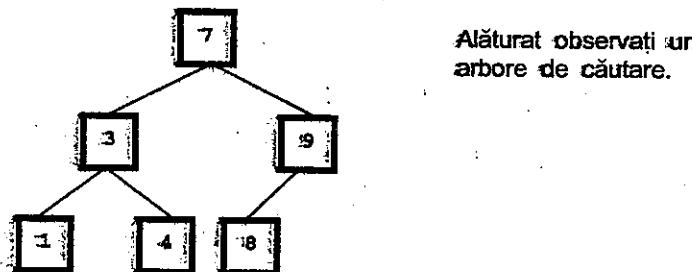
- Funcția **Combinare** - formează un **MinHeap** dintr-un vârf și două **MinHeap**-uri, iar funcția **Heap** - organizează vectorul ca **MinHeap**;
- ✓ Algoritmul are complexitatea $O(n \cdot \log_2 n)$.
- ✓ Pentru sortarea crescătoare se va utiliza **MaxHeap**-ul.

4.2.2.5 Arboare de căutare

Definiție. Se numește arbor de căutare un arbor binar ale cărui noduri au o cheie de identificare, iar pentru fiecare nod avem proprietățile următoare:

- cheia asociată unui nod al subarborelui stâng este mai mică decât cheia asociată nodului;
- cheia asociată unui nod al subarborelui drept este mai mare decât cheia asociată nodului.

Cheile de identificare sunt distincte!



1. Înserarea

Crearea arborilor de căutare se face aplicând de un număr de ori operația de inserare. Regula de inserare este următoarea:

⇒ Se compară cheia asociată unui nod cu cheia înregistrării care se inserează. Avem trei posibilități:

- cheia coincide cu numărul - se renunță la inserarea acelui număr;
- cheia este mai mare decât numărul - se încearcă inserarea în subarborele stâng;
- cheia este mai mică decât numărul - se încearcă inserarea în subarborele drept.

⇒ Inserarea propriu-zisă se realizează atunci când subarborele stâng, respectiv drept, este vid, altfel se reia.

Vezi funcția **Inserare**.

2. Căutarea

Se face în mod asemănător cu inserarea, motiv pentru care nu o mai comentăm.

3. Listarea

Informația se poate lista utilizând oricare din metodele cunoscute pentru parcurgerea arborilor. Dacă dorim listarea informațiilor în ordinea strict crescătoare a cheilor, se utilizează metoda stânga-vârf-dreapta (*inordine*), întrucât pentru orice nod avem următoarele:

- ⇒ cheile nodurilor din subarborele stâng sunt mai mici decât cheia asociată nodului;
- ⇒ cheile nodurilor din subarborele drept sunt mai mari decât cheia asociată nodului.

Vezi funcția **SVD**.

4. Stergerea

După stergerea unui nod care are o anumită cheie, arborele rămas trebuie să fie de căutare.

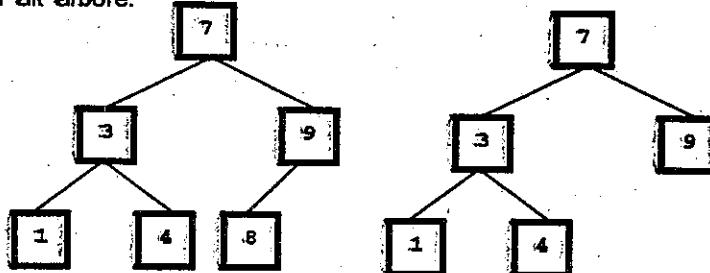
Se disting 4 situații posibile:

- a) nodul care urmează să fie sters este nod terminal - în acest caz se face stergerea având grijă ca la părintele lui să înlocuim adresa către el cu #0;
- b) nodul care urmează să fi sters subordonează un singur subarbore - cel drept - caz în care părintelui i se va înlocui adresa către el cu adresa subarborelui drept, iar nodul respectiv se sterge;
- c) nodul care urmează să fi sters subordonează un singur subarbore - cel stâng - caz în care părintelui i se va înlocui adresa către el cu adresa subarborelui stâng, iar nodul respectiv se sterge;
- d) nodul care urmează să fi sters (după cum vom vedea, acesta se va sterge numai logic) subordonează doi subarbore, caz în care se fac operațiile:
 - se identifică cel mai din dreapta nod al subarborelui stâng corespunzător nodului care urmează să fi sters (acesta va fi sters în mod efectiv, nu înainte de a muta informațiile sale la nodul care se sterge logic);
 - cheia acestuia și alte informații utile continute de el (altele decât cele de adresă) se mută la nodul care urmează să fi sters;
 - subarborele stâng al nodului care se va sterge fizic se leagă;

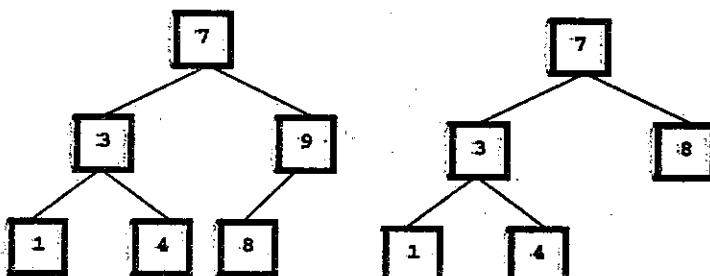
- în stânga nodului care se sterge logic (dacă nodul identificat ca cel mai din dreapta din subarborele stâng este descendent direct al nodului care se sterge logic);
- în dreapta tatălui nodului care se sterge fizic (în caz contrar);

Exemple de stergere:

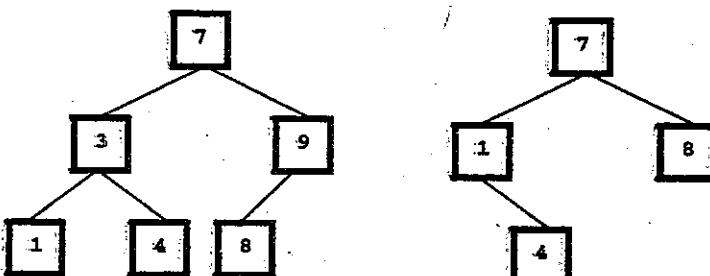
1. Arborelui din stânga i se sterge nodul 8. Acesta nu subordonează nici un alt arbore.



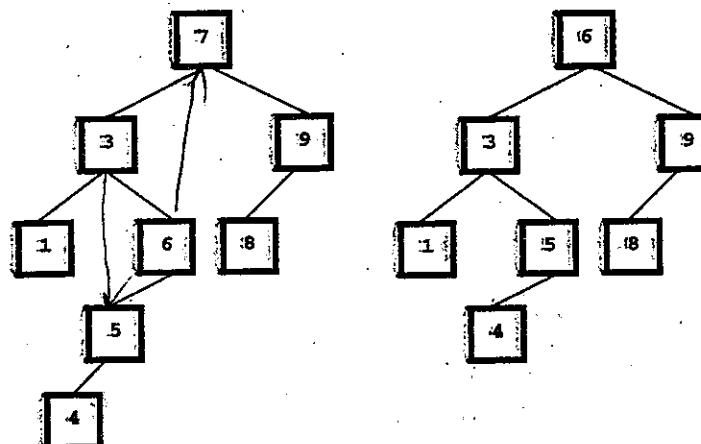
2. Arborelui din stânga i se sterge nodul 9. Acesta subordonează un singur subarbore, cel drept.



2. Arborelui din stânga i se sterge nodul 3. Cel mai din dreapta nod al subarborelui stâng este 1. Se obtine:



3. Arborelui din stânga i se sterge nodul 7. Cel mai din dreapta nod al subarborelui stâng este 6. Tatăl nodului care se sterge fizic este 3. În dreapta sa se șteagă subarborele 5 4. Se obtine:



Vezi funcția `Sterg`. În situația în care un nod care urmează a fi sters subordonează doi arbori, se apelează funcția `CMD`.

- ✓ Întrebarea la care trebuie să răspundem în continuare este următoarea: de ce, dacă se sterge în acest mod un nod, arborele rămâne de căutare? Stergerea unui nod se face în mod distinct pentru fiecare din cele patru cazuri arătate. Datorită simplității prelucrării, primele trei cazuri nu necesită comentarii. În cazul 4 se identifică nodul cel mai din dreapta din arborele stâng (care este cu cheia cea mai mare din acest subarbore). Cheia acestuia trece în locul cheii nodului care se sterge. Aceasta este mai mică decât cheia inițială (pentru că se găsește în subarborele stâng), este în același timp cea mai mare din subarborele stâng, deci este cea mai mare cheie din subarborele stâng care este mai mică decât cheia care se sterge. Iată motivul pentru care aceasta trece în locul cheii sterasse logic.
- ✓ Transmiterea parametrului `c` se face prin referință. Adresa de alocare (obținută prin `new`) va fi trecută automat părintelui, ca adresă de subarbore stâng sau drept (după caz).
- ✓ Algoritmul are complexitatea în cazul cel mai defavorabil $O(n^2)$. În cazul în care, de exemplu, cheile inserate sunt în ordine crescătoare, arborele va degenera într-o listă liniară -orice nod are numai un descendent drept, iar inserarea unui nod înseamnă parcurgerea tuturor nodurilor deja inserate la pasul anterior.

```

#include <iostream.h>
struct Nod
{
    int nr;
    Nod *as, *ad;
};

Nod *v, *man;
char opt;
int k;

void Inserare(Nod*& c, int k)
{
    if (c)
        if (c->nr==k)
            cout<<"nr deja inserat "<<endl;
        else
            if (c->nr<k) Inserare (c->ad,k);
            else Inserare (c->as,k);
    else
    {
        c=new Nod;c->as=c->ad=0;
        c->nr=k;
    }
}

void Cautare(Nod*& c,Nod*& adr,int k)
{
    if (c)
        if (c->nr<k) Cautare(c->ad,adr,k);
        else
            if (c->nr>k) Cautare(c->as,adr,k);
            else adr=c;
        else adr=0;
}

void Parcurg(Nod* c)
{
    if (c)
        {Parcurg(c->as);
        cout<<c->nr<<endl;
        Parcurg(c->ad);}
}

void cmmd(Nod*& c, Nod*& f)
{
    if (f->ad) cmmd(c,f->ad);
    else
    {
        c->nr=f->nr;
        man=f;
        f=f->as;
        delete man;
    }
}

```

```

void Sterg(Nod*& c,int k)
{
    Nod* f;
    if (c)
        if (c->nr==k)
            if( c->as==0 && c->ad==0)
                {delete c;c=0;}
            else
                if (c->as==0)
                    {f=c->ad;delete c; c=f;}
                else
                    if (c->ad==0)
                        {f=c->as;
                        delete c;
                        c=f;}
                    else
                        else cmmd(c,c->as);
        else
            if (c->nr<k)Sterg(c->ad,k);
            else Sterg(c->as,k);
        else cout<<"numar-absent - tentativa esuanta ";
    }

main()
{v=0;
do
    {cout<<"optiunea ";cin>>opt;
    switch (opt)
        {case 'i':
            cout<<"k=";cin>>k;
            Inserare (v,k);
            break;
        case 'l':
            Parcurg(v);
            break;
        case 'c':
            cout<<"k=";cin>>k;
            Cautare(v,man,k);
            if (man) cout<<k<<endl;
            else cout<<"nu exista acest nod"<<endl;
            break;
        case 's':
            cout<<"se va sterge numarul ";
            cin>>k;
            Sterg(v,k);
            break;
        }
    } while (opt!='t');
}

```

Teoria regăsirii informației este deosebit de importantă pentru viața practică. Aceasta conduce la o căutare rapidă (puține comparații).

4.2.2.6 Forma poloneză -aplicații -

Se dă o expresie aritmetică ce folosește operatorii '+', '-', '*', '/' precum și '(', ')'. Fiecare operand este o literă. Se cere să se convertă această expresie într-o expresie în forma poloneză (postfixată).

Definim expresia în forma poloneză (postfixată) recursiv, astfel:

- un operand este o expresie în forma poloneză;
- dacă E_1 și E_2 sunt două expresii în formă poloneză și # un operator, $E_1 E_2 #$ este o expresie în forma poloneză;
- orice expresie în formă poloneză se obține aplicând regulile de mai sus de un număr finit de ori.

Exemple:

EXPRESIE ARITMETICĂ	EXPRESIE ÎN FORMĂ POLONEZĂ
$a+b$	$ab+$
$a * (b+c)$	$abc+*$
$a * (b+c) - e / (a+d) + h$	$abc+*ead+/-h+$

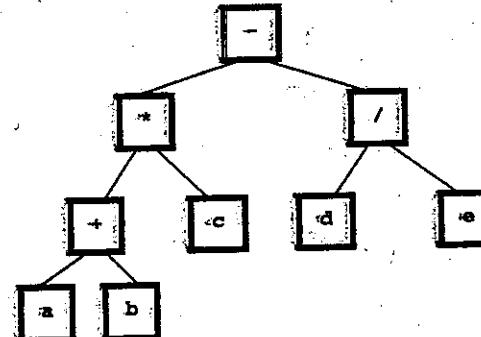
- ✓ Forma poloneză este o formă de scriere a expresiilor aritmetice în care parantezele nu mai sunt puse (din acest motiv o expresie scrisă în forma poloneză se mai numește și expresie scrisă fără paranteze), dar aceasta nu conduce la calculul eronat al expresiei.

Expresiile aritmetice pot fi reprezentate utilizând arbori binari, respectând următoarele reguli:

- fiecare operatie corespunde unui nod neterminal, având ca informație utilă operația respectivă;
- fiecare nod terminal este etichetat cu o variabilă sau cu o constantă;
- pentru fiecare nod neterminal subarboarele din stânga și cel din dreapta reprezintă, în această ordine, cei doi operanzi;
- rădăcina corespunde ultimei operații executate la evaluarea expresiei.

Exemplu: expresiei $(a+b)*c-d/e$ îi se asociază arborele binar din figura de mai jos. Parcurgerea acestui arbore în postordine va da chiar forma poloneză: $ab+c*de/-$. Utilizând cele spuse, pentru rezolvarea problemei, vom proceda în felul următor:

- ⇒ construim arborele binar asociat expresiei aritmetice;
- ⇒ îl parcugem în postordine pentru a obține forma poloneză.



Pentru construcția arborelui vom acorda priorități operatorilor și operanzilor (mai puțin parantezelor), după cum urmează:

- prioritatea initială a operatorilor '+', '-' este 1;
- prioritatea initială a operatorilor '*', '/' este 10;
- la prioritatea unui operator se adună 10 pentru fiecare pereche de paranteze între care se găsește;
- prioritatea unui operand este 1000.

În program acest lucru se realizează astfel:

- se citește expresia aritmetică în variabila e ;
- se utilizează o variabilă p care indică ce număr se adaugă la prioritatea initială a unui operator (la întâlnirea unei paranteze deschise p crește cu 10, iar la întâlnirea unei paranteze închise p scade cu 10);
- se parcurge expresia caracter cu caracter și se pun în vectorul p prioritățile acestor operatori și operanzi (mai puțin ale parantezelor);
- în efp se construiește expresia fără paranteze (la expresia aritmetică inițială lipsesc parantezele), iar în pfp se obține vectorul priorităților, din care, spre deosebire de p , lipsesc componentele corespunzătoare parantezelor (acestea nu aveau nici o valoare).

Utilizând **efp** și **pfp**, cu ajutorul funcției **arb**, se construiește arborele atașat expresiei aritmetice. Un nod al acestui arbore are ca informație utilă un operator sau un operand.

Conform tehnicii DIVIDE ET IMPERA, funcția **arb** procedează în felul următor:

- ⇒ de la limita superioară către limita inferioară (limite corespunzătoare subșirurilor de caractere tratate din **efp**) căută operatorul sau operandul cu prioritate minimă, reținând poziția acestuia. Acesta constituie informația utilă din nod, care va fi completată;
- ⇒ în situația în care limita inferioară este diferită de limita superioară, pentru completarea adresei subarborelui din stânga și a subarborelui din dreapta se repeleză funcția, iar în caz contrar aceste câmpuri capătă valoarea 0.

Pentru listarea în postordine se utilizează funcția **parc**.

```
#include <iostream.h>
struct Nod {
    char op;
    Nod *as,*ad;
};

char e[30],efp[30],a;
int pfp[30],p[30],i,j,n;
Nod *c;

Nod* arb(int li,int ls,char efp[30],int pfp[30])
{
    Nod* c;
    int i,j,min;
    min=pfp[ls];
    i=ls;
    for (j=ls;j>=li;j--)
        if (pfp[j]<min)
            {min=pfp[j]; i=j;}
    c=new Nod; c->op=efp[i];
    if (li==ls) c->as=c->ad=0;
    else
        { c->as=arb(li,i-1,efp,pfp);
          c->ad=arb(i+1,ls,efp,pfp);
        }
    return c;
}

void parc(Nod* c)
{
    if (c)
        {parc(c->as);
         parc(c->ad);
         cout<<(c->op);
        }
}
```

```
main()
{
    j=0;cin>>a;n=1;
    while (a!=='')
    { e[n++]=a;
      cin>>a;
    }
    n--;
    for (i=1;i<=n;i++)
        switch(e[i])
        { case '+': j+=10; break;
          case '-': j+=10; break;
          case '*': p[i]=j+1; break;
          case '/': p[i]=j+1; break;
          case '**': p[i]=j+10; break;
          case '/': p[i]=j+10; break;
          default : p[i]=1000;
        }
    j=1;
    for (i=1;i<=n;i++)
        if (e[i]==')' && e[i-1]=='(')
        { efp[j]=e[i];
          pfp[j]=p[i];
          j++;
        }
    carb(1,j-1,efp,pfp);
    parc(c);
}
```

Aplicatie. Dându-se o expresie în forma poloneză postfixată, să se scrie un program care generează programul de calcul al expresiei utilizând numai instrucțiuni de atribuire cu un singur operator. Se vor introduce variabilele auxiliare x_0, x_1, \dots, x_n .

Exemplu: pentru expresia aritmetică $a*(b+c)-e/(a+d)+h$, forma poloneză este $abc+*ead+/-h+$. Se generează următorul program de calcul:

```

x0=b+c
x1=a*x0
x2=a+d
x3=e/x2
x4=x1-x3
x5=x4+h.

```

Pentru realizarea scopului propus, procedăm astfel:

- folosim o stivă dublă **ST**, care permite memorarea variabilelor $x_0, x_1, \dots, x_n, a, b, \dots, z$.
- forma poloneză se găsește în **FP**.

c) din FP, se citește caracter cu caracter iar rezultatul citirii se tratează diferențiat astfel:

- în cazul în care caracterul citit este operand, se încarcă în stivă;
- în cazul în care caracterul citit este operator, se scoacă din stivă conținuturile ultimelor două niveluri și se face tipărirea sub forma:
- $x_3 = \text{variabila penultimă în stivă, operator, variabilă ultimă în stivă, iar variabila tipărită se reține în stivă.}$

Procedeul se repetă atât timp cât nu au fost citite toate caracterele. Pe exemplul considerat, rularea décerge astfel:

c
b
a

a, b, c se încarcă în stivă;

x_0
a

la citirea operatorului "+" se tipărește $x_0 = b+c$ și x_0 se pune în stivă;

x_1

citirea operatorului "*", determină tipărirea $x_1 = a*x_0$ și x_1 se pune în stivă;

d
a

e, a, d se încarcă în stivă;

x_1

se tipărește $x_2 = a+d$;

x_2
e

se tipărește $x_3 = e/x_2$;

x_3
x_1

se citește "/", se tipărește $x_3 = e/x_2$;

34 se citește "-", se tipărește $x_4 = x_1 - x_3$;

characterul "h" se pune în stivă;

la citirea operatorului "+" se tipărește $x_5 = x_4 + h$.

Întrucât au fost citite toate caracterele, algoritmul se încheie.

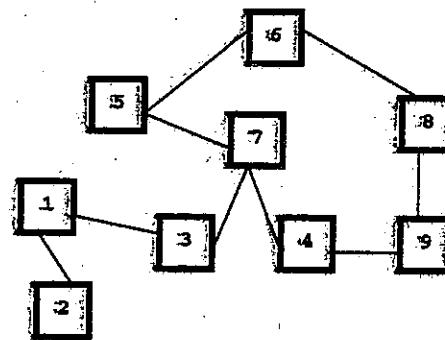
```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
char st[100][2], fp[100], c[2];
int i,j,k;

main()
{ cout<<"Forma polonexă este "; cin>>fp;
  for (i=0;i<=strlen(fp)-1;i++)
    if (fp[i]>='a' && fp[i]<='z')
      { k++;
        st[k][1]=fp[i];
        st[k][0]=' ';
      }
    else
      if ( fp[i]=='+' || fp[i]=='-' || fp[i]=='*' || fp[i]=='/')
        { cout<<"x "<<j<<" = "<<st[k-1][0]<<st[k-1][1];
          <<fp[i]<<st[k][0]<<st[k][1]<<endl;
          k--;
          st[k][0]='x';
          itoa(j,c,10);
          st[k][1]=c[0];
          j++;
        }
  }
}
```

✓ Procedeul indicat este utilizat pentru compilarea expresiilor. Se cunoaște faptul că procesorul are instrucțiuni care permit o singură operare și doi operanzi. O expresie trebuie adusă sub forma prezentată, pentru obținerea codului masină. Algoritmii utilizati sunt de acest tip (evident, aceștia sunt și optimizați).

Probleme propuse

1. Priviți graful de mai jos. Eliminați-o singură muchie astfel încât să rămână un arbore. Desenați arborele presupunnd că vârful său este 16. La fel, dacă vârful este 4.



2. Reprezentați arborele obținut la problema 1 prin:

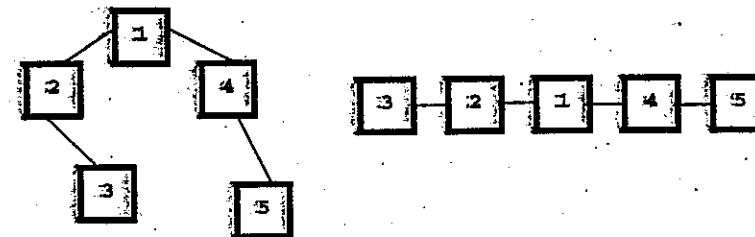
- Matricea de adiacență;
- Liste de adiacență;
- Prin legătura de tip TATA, considerând vârful 4.
- Prin codul lui Prüffer.

3. Un arbore este citit dintr-un fisier text și este memorat sub forma listelor de adiacență (vezi capitolul 3). Reprezentați arborele sub forma legăturii TATA, alegând ca vârf un nod oarecare.

4. Se dau n calculatoare. Se cunoaște distanța între oricare două calculatoare. Se cere să se găsească o soluție astfel încât acestea să fie legate între ele, iar lungimea totală a cablului utilizat să fie minimă. Datele se găsesc în fisierul text `reteza.in` și sunt organizate astfel:

- Linia 1: n
- următoarele $\frac{n(n-1)}{2}$ linii ale fisierului contin fiecare trei valori naturale i , j , k cu semnificația: distanța între calculatorul i și calculatorul j este de k metri.
- ✓ Pentru ca două calculatoare să poată comunica este necesar ca ele să fie legate, dar nu în mod obligatoriu direct.

5. Se citește un arbore dat sub forma legăturii de tip TATA. Nodurile sale sunt $1, 2, \dots, n$. Poate fi el memorat sub forma unei liste liniare? În caz afirmativ, să se creeze lista și să se afișeze. Exemplu: arborele de mai jos este reprezentat sub forma unei liste.



6. Un vector cu n componente reține o pădure cu nodurile $1, 2, \dots, n$. Căți arbori conține pădurea?

7. Se citește un arbore cu nodurile $1, 2, \dots, n$ și este memorat sub forma listelor de adiacență. Se citește, de asemenea un nod, care va fi considerat vârf. Se cere să se afișeze numărul de niveluri ale arborelui.

8. În mod evident, numărul de niveluri ale unui arbore depinde de vârful ales. Findând un arbore, care este numărul minim de niveluri și pentru care vârf se realizează?

9. Se citește un arbore care este introdus prin legătura de tip TATA. Nodurile sale sunt $1, 2, \dots, n$. Se cere să se afișeze nodurile sale terminate.

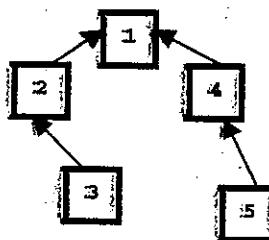
10. Se citesc doi arbori, unde fiecare are nodurile $1, 2, \dots, n$, și sunt reprezentați prin legătura de tip TATA. Se cere să se decidă dacă ei reprezintă unul și același arbore.

11. Se citește un arbore cu nodurile $1, 2, \dots, n$ și se memorează sub forma listelor de adiacență. Afisati parcurgerea în lățime, considerând, pe rând, vârfurile $1, 2, \dots, n$.

12. Se citește un arbore cu nodurile $1, 2, \dots, n$ și se memorează sub forma listelor de adiacență. Afisati parcurgerea în adâncime, considerând, pe rând, vârfurile $1, 2, \dots, n$.

13. Este posibil să considerăm un arbore ca un graf bipartit? Dacă da, pentru un arbore citit, afișați o soluție pentru multimile A și B . (vezi problema 29 capitolul 3) Dar un graf bipartit este întotdeauna arbore?

14. Priviți graful orientat de mai jos. Este o arborescentă? Justificați răspunsul.



15. Un arbore binar cu nodurile $1, 2, \dots, n$ este reprezentat în **HEAP** și adresa rădăcinii este în **v**. Scrieți o funcție care memorază arborele cu ajutorul vectorilor.

16. Un arbore binar cu nodurile $1, 2, \dots, n$ este reprezentat cu ajutorul vectorilor. Scrieți o funcție care îl memorază în **HEAP**.

17. Scrieți o funcție care returnează numărul de niveluri ale unui arbore binar memorat în **HEAP**.

18. Scrieți o funcție care returnează numărul de niveluri ale unui arbore binar memorat cu ajutorul vectorilor.

19. Scrieți o funcție care copiază un arbore binar memorat în **HEAP** și care are adresa vârfului în **v** și obține un altul memorat tot în **HEAP** care are adresa în **v1**.

20. Scrieți o funcție care eliberează memoria din **HEAP** ocupată de un arbore binar.

21. Scrieți o funcție care afișează nodurile unui arbore binar memorat în **HEAP** în ordinea obținută în urma parcurgerii acestuia în lățime.

22. Scrieți o funcție care listează nodurile de pe nivelul k al unui arbore binar memorat în **HEAP**.

23. Scrieți o funcție care listează nodurile de pe nivelul k ale unui arbore binar memorat cu ajutorul vectorilor.

24. Scrieți o funcție care listează nodurile terminale ale unui arbore binar memorat în **HEAP**.

25. Scrieți o funcție care listează nodurile terminale ale unui arbore binar memorat cu ajutorul vectorilor.

26*. Numărarea arborilor binari. Autoinstruire.

26.1 Se citește n , număr natural. În câte feluri se pot așeza în linie n litere **P** și n litere **S**. Exemplu: $n=2$. Avem: **PPSS**, **SSPP**, **SPSP**, **SPPS**, **PSSP**. Se va afișa 6 .

Indicație. Dacă fiecare literă ar fi distinctă am avea $(2n)!$ posibilități de a aseza literele. Întrucât pentru fiecare permutare nu contează dacă s-a inversat **P** cu **P**, $(2n)!$ se împarte la $n!$ și pentru că nu contează dacă s-a inversat **S** cu **S**, împărțim din nou la $n!$. Astfel obținem:

$$\frac{(2n)!}{n!n!} = C_{2n}^n$$

26.2 La fel ca la 26.1 dar se cere numărul secvențelor cu $n+1$ litere **P** și $n-1$ litere **S**.

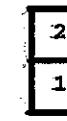
Răspuns. Printr-un rationament asemănător se obține: C_{2n}^{n-1}

26.3 Se citește n , număr natural. Fie permutarea $1, 2, \dots, n$ și o secvență cu n litere **S** și n litere **P**. Se presupune existența unei stive. Secvența de litere o interpretăm ca $2n$ comenzi de scoatere din stivă (**S**) sau punere în stivă (**P**). Exemplu: $n=3$. Permutarea este $1\ 2\ 3$, iar secvența este **PPSPSS**. Executăm comenzi:

P 1 se pune în stivă: rămâne **2 3**.



P 2 se pune în stivă: rămâne **3**.



S 2 se scoate din stivă.



2

P 3 se pune în stivă.



2

S 3 se scoate din stivă.



2

S 1 se scoate din stivă.



0

1 3 2

Astfel, din permutarea initială $1\ 2\ 3$ am obținut permutarea $1\ 3\ 2$ - ca în cazul manevrelor de vagoane.

Se cere ca programul av. să decidă dacă secvența de comenzi este admisibilă, în caz afirmativ să se tipărească permutarea care se obține executând secvența.

Indicație. Ca o secvență să fie admisibilă este necesar ca pentru fiecare i între 1 și $2n$ secvența $1..i..k$ să contină un număr de comenzi s mai mare sau egal decât numărul de comenzi p (astfel, stiva este vidă și nu se poate executa s).

26.4 Se citește n , număr natural. Se cere să se listeze toate permutările multimi $\{1, 2, \dots, n\}$ care se pot obține din permutarea identică ca în problema anterioară.

Indicație. Backtracking, dinând cont de indicația din problema.

26.5 Se citește n . Câte permutări afișează programul anterior?

Rezolvare. În nici un caz nu le numărăm. Am așteptat cam mult... O demonstrație surprinzătoare a fost obținută în 1878 de D. Andree.

\Rightarrow Numărul total de secvențe este: C_{2n}^n

\Rightarrow Fie o secvență inadmisibilă. Fie k , minim astfel încât secvența $1..k..k$ să fie inadmisibilă. Exemplu: $SPSPSP$. Aici $k=5$.

\Rightarrow În secvența $1..k..k$ toate comenziile p devin comenzi s și toate comenziile s devin comenzi p (pentru exemplul dat obținem $SPPSPP$). În acest fel am obținut o secvență cu $n-1$ comenzi s și $n+1$ comenzi p .

\Rightarrow Fie acum o secvență cu $n-1$ comenzi s și $n+1$ comenzi p . Căutăm k minim astfel încât secvența $1..k..k$ să contină mai multe comenzi p decât comenzi s . De exemplu, pentru $SPPSPP$, k este 5. Înversăm p cu s în secvența $1..k..k$. Pentru exemplul dat avem: $SPSPSP$.

\Rightarrow Să observăm că orice secvență cu $n-1$ comenzi s și $n+1$ comenzi p se transformă într-o secvență inadmisibilă prin procedeul indicat (asta pentru că în secvență obținută există mai multe comenzi s decât p).

\Rightarrow Prin urmare, am stabilit o bijectie de la multimea secvențelor inadmisibile la cea a secvențelor care au $n-1$ comenzi s și $n+1$ comenzi p . Deoarece cele două multimi sunt finite înseamnă că au același număr de elemente. Dar numărul secvențelor cu $n-1$ comenzi s și $n+1$ comenzi p il cunoaștem. Vezi 26.2. De aici rezultă că numărul secvențelor admisibile cu n comenzi s și n comenzi p este:

$$C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$$

26.6 Se citește n , număr natural. În câte feluri se pot așeza n perechi de paranteze $()$ într-o expresie aritmetică astfel încât expresia să aibă sens.

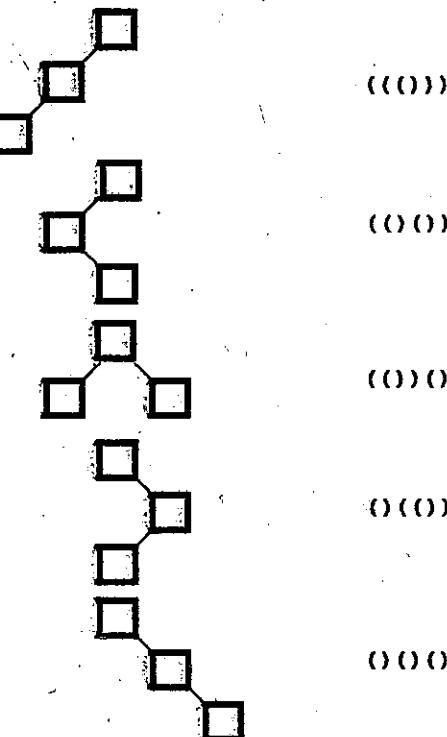
Indicație. Problema este identică cu anterioara dacă înlocuim p cu $($ și s cu $)$. Prin urmare, rezultatul este cel de mai sus.

26.7 Fie A multimea arborilor binari și B multimea secvențelor admisibile de n perechi de paranteze, ca la problema anterioară. Fiecarui arbore binar din A îl atașăm o secvență admisibilă de paranteze după formula:

$$\text{vârf} = ([\text{subarbore stâng}]) (\text{subarbore drept}])$$

Ce este trecut între paranteze drepte este facultativ (pentru cazul când subarborele stâng sau drept lipsesc).

Exemplu: reprezentăm toți arborii binari cu trei noduri:



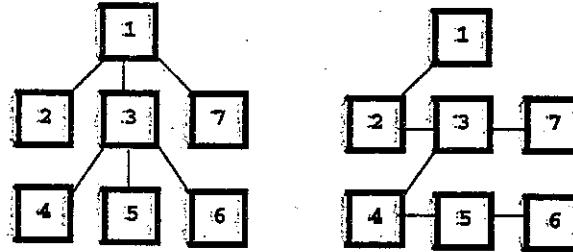
Să se scrie un program care citește un arbore binar cu n noduri reprezentat cu ajutorul vectorilor și tipărește succesiunea admisibilă de paranteze.

26.8 Care este numărul arborilor binari cu n noduri, abstractie făcând de numerotarea acestora?

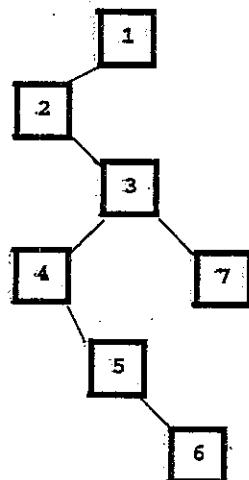
Indicatie. Se demonstrează că la 26.7 am definit o funcție $F: A \rightarrow P$, bijectivă. De aici rezultă că numărul acestora este:

$$\frac{1}{n+1} C_{2n}^n$$

27. Un arbore oarecare, de vîrf dat poate fi reținut ca un arbore binar astfel: pentru fiecare nod îi vârful subarborelui stâng este dat de primul descendenter al său (în ordinea de la stânga la dreapta) iar vârful subarborelui drept de primul nod aflat pe același nivel în aceeași ordine. Exemplu:



Arborele oarecare din stânga sus se reprezintă ca arbore binar, aşa cum rezultă din imaginile următoare.



Citiți un arbore oarecare într-o reprezentare convenabilă și reprezentați-l în **HEAP** ca arbore binar. Invers, pomiți de la un arbore binar memorat în **HEAP** și reprezentați-l ca un arbore oarecare, reprezentat într-o structură convenabilă.

Capitolul 5

Initiere în programarea orientată pe obiecte

5.1 Introducere

Conceptul poate fi întâlnit sub diverse denumiri, cum ar fi: programare orientată pe obiecte, programare obiectuală, programare orientată spre obiect, sau, pe scurt, **OOP - Object Oriented Programming**.

- O dorință mai veche a programatorilor a fost aceea de a utiliza soft deja scris. În acest fel există posibilitatea ca programatori să devină cu mult mai eficienți atunci când au de scris o aplicație.
- ✓ Metoda de a refolosi munca altora nu este nouă, este utilizată aproape peste tot. De exemplu, dacă se dorește să se construiască un calculator, se poate folosi un procesor deja existent. Acesta nu mai este cazul să fie reproiectat! Tot așa se poate folosi o placă de bază existentă, un monitor un hard-disk. Unde s-ar ajunge dacă de fiecare dată toate acestea ar fi reproiectate pornindu-se de la tranzistoare și rezistente?
- ✓ Zeci de ani singura posibilitate de refolosire a soft-ului a fost data de utilizarea subprogramelor. Au apărut colecții de subprograme grupate pe teme. Chiar în C++ există o serie de fisiere anexe care permit apelarea multor funcții specializate.
- ✓ Programarea orientată pe obiecte prezintă avantaje cu mult mai mari decât utilizarea subprogramelor, pe care o include. Cunoștințele dobândite până în acest moment nu ne permit nici măcar o enumerare a acestora.
- ✓ O aplicație a programării orientate pe obiecte este Programarea Vizuală. Orice apare pe ecranul dv. atunci când folosiți **Windows** este realizat cu ajutorul **OOP**. Fereastra este un obiect, meniuul atașat ferestrei este un alt obiect, butoanele sunt obiecte, bara cu instrumente este un obiect s.a.m.d.
- ✓ Astăzi este imposibil să realizăm un program cenuș pe piață fără ajutorul **OOP**. Imaginați-vă ce ar însemna ca programatorul să scrie secvența prin care se desenează o fereastră, de câte ori programul său o va folosi. Ar fi o muncă titanică și inutilă. A fost făcută o dată de Microsoft, n-are rost să-ă reluăm, decât, cel mult, în scop didactic.

5.2 Încapsularea

5.2.1 Definirea claselor

Definiție: Prin **încapsulare** înțelegem mecanismul prin care datele (variabilele) și funcțiile (numite în acest caz și **metode**) sunt plasate împreună, într-o unică structură.

În C++ aceasta se realizează printr-un tip abstract, numit **class**. În C++ se face distincție între clasa, care este tipul abstract, și obiectul propriu-zis care este instantierea clasei respective.

Notă forma simplificată a tipului clasă:

```
class
{
    [private]
    date;
    metode;
    [protected]
    date;
    metode;
    [public]
    date;
    metode;
}
```

Între paranteze drepte sunt trecuți modificatorii **public**, **protected**, **private**. Un modificador are efect până la întâlnirea altuia, sau până la sfârșitul definiției.

- modificadorul **private** are rolul de a interzice accesul la date și metode din afara obiectului;
- modificadorul **public** are rolul de a permite accesul din afara obiectului la date și metode;
- modificadorul **protected** va fi prezentat în alt context.

În absența modificatorilor, este interzis accesul la date și metode din afara obiectului.

Exemplu. La matematică v-ați întâlnit cu numere complexe! Ele sunt de formă: $z=x+iy$, unde $x, y \in \mathbb{R}$. În C++ există mai multe tipuri care rețin numere reale! Prin mecanismul OOP încercăm să construim singuri un tip care permite memorarea și prelucrarea numerelor complexe.

```
class complex
{
    public:
        float x, y;
        float modul();
};
```

Un număr real cuprinde:

- partea reală - x ;
- partea imaginară - y
- o funcție care returnează modulul: $\sqrt{x^2+y^2}$

Descrierea numărului complex îl vom face într-un tip separat, numit **complex**:

```
#include <math.h>
class complex
{
    public:
        float x, y;
        float modul();
};

float complex::modul()
{
    return sqrt (x*x+y*y);
}
```

► Definiția funcției **modul** (metodă a clasei **complex**) s-a făcut în afara clasei respective. Pentru a preciza faptul că funcția este metodă a clasei **complex**, utilizăm în antetul definiției operatorul ":", numit și **operator de rezoluție**:

```
float complex::modul()
```

- ✓ Există posibilitatea ca funcția să fie definită în interiorul clasei. În acest caz ea va fi tratată de compilator ca funcție **inline**. Reamintim că pentru o astfel de funcție orice apel este înlocuit prin codul ei. Astfel se obține o economie de timp - cea necesară saltului la cod și revenirii - dar se mărește codul programului. Din acest motiv vom prefera să definim funcția în afara clasei.
- Programul următor utilizează un obiect **complex** pentru care atribuie valori părții reale și imaginare, apoi le tipărește împreună cu modulul acestuia.

```
#include <iostream.h>
#include "complex.cpp"

main()
{
    complex z;
    z.x=3;z.y=4;
    cout<<z.x<<" "<<z.y<<" "<<z.modul();
}
```

- Obiectul propriu-zis a fost declarat ca o variabilă. El se numește `z`.
- Obiectul este, de fapt, instantierea clasei din care face parte.
- Ca și în cazul tipului `struct` accesul la datele (metodele) membre se realizează prin numele obiectului, urmat de operatorul `"."` și de numele variabilei (metodei): `z.x`, `z.y`, `z.modul()`.
- ✓ Prin utilizarea obiectului de tip `complex`, construit acum, putem lucra "aproape" la fel ca și cum limbajul ar conține tipul `complex`.
- Programul următor citește un vector de numere complexe și tipărește cel mai mare modul al acestora.

```
#include <iostream.h>
#include "complex.cpp"

main()
{ complex v[100];
  int n,i;
  float max;
  cout<<"n=">>n;
  for (i=0;i<n;i++)
  {
    cout<<"Numarul complex "<<i+1<<endl;
    cout<<"partea reala ">>v[i].x;
    cout<<"partea imaginara ">>v[i].y;
  }
  max=v[0].modul();
  for (i=1;i<n;i++)
    if (v[i].modul()>max) max=v[i].modul();
  cout<<max;
}
```

- Programul următor sortează un vector de numere complexe, descrescător, în funcție de modulul lor.

```
#include <iostream.h>
#include "complex.cpp"

main()
{ complex v[100],man;
  int n,i,inversari;
  cout<<"n=">>n;
  for (i=0;i<n;i++)
  {
    cout<<"Numarul complex "<<i+1<<endl;
    cout<<"partea reala ">>v[i].x;
    cout<<"partea imaginara ">>v[i].y;
  }
  do
  { inversari=0;
    for (i=0;i<n-1;i++)
      if(v[i].modul()<v[i+1].modul())
        { man=v[i];
          v[i]=v[i+1];
          v[i+1]=man;
          inversari=1;
        }
    }while (inversari);
  for (i=0;i<n;i++)
    cout<<v[i].x<<" "<<v[i].y<<endl;
}
```

```
{ man=v[i];
  v[i]=v[i+1];
  v[i+1]=man;
  inversari=1;
}
}while (inversari);
for (i=0;i<n;i++)
cout<<v[i].x<<" "<<v[i].y<<endl;
}
```

- ✓ Observați că, în cazul obiectelor, atribuirea este permisă întocmai ca la variabile. În fapt, o atribuire constă într-o copiere a datelor unui obiect, în altul. O astfel de atribuire, se numește în termenii OOP copiere bit cu bit.
- ✓ Atenție! Copierea bit cu bit se poate face doar în anumite cazuri. Cazurile când aceasta nu se poate face vor fi semnalate la momentul potrivit.
- Datele și metodele unui obiect pot fi accesate pornind de la un pointer către acesta. De asemenea, obiectele pot fi alocate în **HEAP**. Întrucât nu există modificări față de ceea ce cunoaștem deja, ne mărginim să dăm un exemplu în care un obiect de tipul `complex` este alocat în **HEAP**.

```
#include <iostream.h>
#include "complex.cpp"

main()
{ complex *adr_comp=new complex;
  adr_comp->x=3;
  adr_comp->y=4;
  cout<<adr_comp->modul();
  delete adr_comp;
}
```

- Obiectele pot fi descrise și cu `struct`, la fel cum am facut-o acum cu ajutorul lui `class`. Diferența este, în acest caz, că datele și metodele sunt implicit publice, față de `class`, unde acestea sunt implicit private.

5.2.2 Constructori, destructori

Revenim asupra declaratiilor obiectelor. Fie declaratia: `complex a,b;` Au fost declarate două obiecte numite `a` și `b`.

⇒ Întrebarea 1: care este mecanismul ce permite declararea (instantierea) obiectelor?

În absența altor definiții, clasei i se atasează în mod implicit o metodă specială, numită, constructor, care are rolul de a permite declarația obiectelor. În acest fel se alocă spațiul de memorie necesar obiectelor declarate. De asemenea, numele lor pot fi folosite pentru accesarea datelor și metodelor. Un astfel de constructor îl numim constructor generat implicit și are următoarele caracteristici:

- nu are parametri formali;
- consecința absenței parametrilor formali este faptul că nu este permisă initializarea la declarare a datelor membru;
- nu se generează, în cazul în care clasa are atașat un alt constructor, fără parametri.

⇒ Întrebarea 2: programatorul poate înzestră clasa cu o metodă constructor proprie?

Răspunsul este afirmativ, cu mențiunea că este necesar să respectăm anumite reguli suplimentare, comparativ cu o metodă obișnuită.

- O metodă constructor are întotdeauna numele clasei din care face parte. Ea este apelată automat la declararea obiectelor.
- Un constructor este o funcție fără tip. Cu toate acestea, în dreptul tipului nu se trece cuvântul cheie `void`.

Exemple:

1. Clasa `intreg` este înzestrată cu un constructor implicit. Cu ajutorul lui instantiem obiectele `x, y, z`.

```
class intreg
{ public:
    int a;
};

main(){ intreg x,y,z; }
```

2. Atașăm clasei `intreg` un constructor fără parametri. În acest caz nu se atasează implicit alt constructor. Programul tipărește de 4 ori sirul "trece" pentru că au fost declarate (instantiate) 4 obiecte. Aceasta înseamnă că s-a apelat constructorul de 4 ori.

```
#include <iostream.h>
class intreg
{ public:
    int a;
    intreg();
};

intreg::intreg()
{
    cout<<"trece "<<endl;
}

intreg k;

main()
{ intreg x,y,z; }
```

3. Programul următor prezintă un exemplu de utilizare a unui constructor cu parametru implicit. La fiecare apel se afisează valoarea de initializare.

```
#include <iostream.h>
class intreg
{ public:
    int a;
    intreg(int v=0);
};

intreg::intreg(int v)
{ cout<<"trece "<< v << endl;
}

intreg k;
main()
{ intreg x(3),y(4),z(); }
```

După cum observați, obiectele se declară în trei moduri:

- a) Prin utilizarea operatorului de atribuire "`=`". În acest caz, valoarea aflată în dreapta operatorului este convertită către tipul `int` (tipul parametrului formal al constructorului);
- b) Prin utilizarea perechii de paranteze rotunde. Efectul este același, dar apelul constructorului este făcut întocmai ca apelul unei metode.
- c) Obiectul se declară fără a fi inițializat explicit (se inițializează cu valoarea implicită).

- ✓ Analiza exemplului de mai sus conduce la observația: constructorii obiectelor globale se apelează primii.
- ✓ Știați că și variabilele clasice se pot declara așa? Ce concluzie trageți de aici?

```
int x=3, y(4);
cout<<x<<" "<<y;
```

⇒ Întrebarea 3: O clasă poate avea mai mulți constructori?

Răspunsul este afirmativ, dar există o restricție: metodele constructor trebuie să difere prin numărul și tipul parametrilor formalii. Problema se reduce la apelul mai multor funcții (metode) care au același nume, dar parametri formalii diferiți, adică la supraîncărcării funcțiilor!

Exemplu: vom înzestră clasa **complex** cu doi constructori, unul cu doi parametri, altul cu un singur parametru:

```
class complex
{
public:
    float x, y;
    float modul();
    complex(int v1, int v2);
    complex(int v=0);
};

complex::complex(int v1, int v2)
{ x=v1; y=v2; }

complex::complex (int v)
{ x=1; y=v; }
...
```

Iată și un exemplu de utilizare:

```
#include <iostream.h>
#include "complex.cpp"

main()
{ complex z1(3,4), z2(5);
  cout<<z1.x<<" "<<z1.y<<" "<<z2.x<<" "<<z2.y;
}
```

În condițiile de mai sus, priviți declaratiile:

```
complex z1(3,4), z2(5), z3=z1;
```

⇒ Întrebarea 4: Primele două declarații au fost explicate. În schimb, ultima **z3=z1**, ar părea fără sens. După cum am văzut, parametrul constructorului este de tip **int**. Aceasta înseamnă că lui **z1** nu îl se poate atribui parametru. Atunci?

În realitate există două tipuri de constructor. Un tip de constructor care asigură alocarea și initializarea obiectelor, altul de copiere, care permite atribuirea datelor unui obiect altuia.

- Orice obiect este înzestrat cu un constructor implicit de copiere. În caz contrar, atribuirea de mai sus ar fi semnalată ca eroare.
- Constructorul implicit de copiere realizează o copiere "bit cu bit" a datelor membru.
- Există posibilitatea ca programatorul să scrie și constructorul de copiere. Iată, de exemplu, cum arată acesta în cazul clasei **complex**:

```
class complex
{
public:
    float x, y;
    float modul();
    complex(int v1, int v2);
    complex(int v=0);
    complex (complex& v);
};

...
complex::complex(complex& v)
{
    cout<<"copiez"<<endl;
    x=v.x;
    y=v.y;
}
```

- ✓ La atribuire se afișează sirul "copiez". Aceasta are rolul de a testa apelul constructorului: **complex z3=z1;**
- ✓ Observați că parametrul unui constructor de copiere este chiar de tipul clasei din care face parte constructorul respectiv.
- ✓ Transmiterea prin referință este preferată transmiterii prin valoare. În practică obiectele pot conține multe date și transmiterea prin valoare ar încărca stiva inutil.

Dacă alocarea obiectelor se face prin constructor, dezalocarea vor se face prin destructor. În cazul în care nu s-a definit un destructor se generează automat unul.

Pentru a defini un destructor trebuie să ținem cont de următoarele:

- numele destructorului coincide cu numele clasei căreia îl aparține și este precedat de "~-";
- destructorul nu are parametri formali;
- o clasă are un singur destructor;
- destructorul se apelează automat atunci când viața unui obiect începează (la sfârșitul executării programului, pentru obiectele globale, la ieșirea din blocul respectiv, pentru obiectele automate);
- destructorul poate fi apelat explicit. Întrucât apelul explicit al destructorului se realizează, de regulă, în legătură cu dezalocarea din HEAP a obiectelor create dinamic, vom reveni asupra sa, atunci când vom realiza primele aplicații.

5.2.3 Supraîncărcarea operatorilor

Ne propunem să adăugăm clasei `complex` metode care permit operații aritmetice cu numere complexe. De exemplu, să adăugăm o metodă `adun` care are rolul de a aduna două numere complexe. Unul dintre ele este obiectul curent (cel prin care se apelează funcția), iar celălalt este transmis prin valoare. Iată cum arată clasa `complex` după adăugarea metodelor `adun`:

```
class complex
{
public:
    float x, y;
    float modul();
    complex(float v1, float v2);
    complex(float v=0);
    complex (complex v);
    complex adun(complex z);
};

complex complex::adun(complex z)
{ complex t;
    t.x=x+z.x; t.y=y+z.y;
    return t;
}
```

Programul următor adună două numere complexe, `z1` - obiectul curent - și `z2` - obiectul transmis ca parametru - iar rezultatul este atribuit lui `z1`.

```
main()
{
    complex z1(3,4),z2(7,5);
    z1=z1.adun(z2);
    cout<<z1.x<<" "<<z1.y<<" "<<z2.x<<" "<<z2.y;
}
```

După cum observăm, apelul metodei nu este elegant, fiind departat de cel cunoscut în matematică. Există și o altă posibilitate și anume utilizarea funcțiilor operator.

Funcția (metoda) de mai sus (`adun`) se va numi de această dată `operator+`. Cuvântul operator este cuvânt cheie. Prin urmare, prototipul și definiția devin:

```
complex operator+(complex z);

complex complex::operator+(complex z)
{
    complex t;
    t.x=x+z.x; t.y=y+z.y;
    return t;
}
```

În aceste condiții, funcția poate fi apelată în două moduri:

1. `z1=z1+operator+(z2);`
2. `z1=z1+z2;`

Forma 2 este ideală. Ea face ca lucrul cu obiectele de tip `complex` să fie atât de ușoară, ca și cum acestea ar fi continue de limbaj.

În mod practic, operatorul `+` este folosit și în alt scop decât cel folosit de limbaj. Cu alte cuvinte a fost supraîncărat. Tot așa, am supraîncărat și operatorii `"-"`, `"**"`, `"/"` pentru scăderea, înmulțirea și împărțirea numerelor complexe. Pentru aceasta, am utilizat formulele următoare:

$$z_1 + z_2 = x_1 + i \cdot y_1 + x_2 + i \cdot y_2 = x_1 + x_2 + i \cdot (y_1 + y_2).$$

$$z_1 - z_2 = x_1 + i \cdot y_1 - x_2 - i \cdot y_2 = x_1 - x_2 + i \cdot (y_1 - y_2).$$

$$z_1 \cdot z_2 = (x_1 + i \cdot y_1)(x_2 + i \cdot y_2) = x_1 \cdot x_2 - y_1 y_2 + i \cdot (x_1 y_2 + x_2 y_1).$$

$$\frac{z_1}{z_2} = \frac{x_1 + i \cdot y_1}{x_2 + i \cdot y_2} = \frac{(x_1 + i \cdot y_1) \cdot (x_2 - i \cdot y_2)}{(x_2 + i \cdot y_2) \cdot (x_2 - i \cdot y_2)} = \frac{x_1 x_2 + y_1 y_2 + i(x_2 y_1 - x_1 y_2)}{x_2^2 + y_2^2}$$

Mai jos, este prezentată noua clasă **complex**:

```
#include <math.h>
class complex
{ public:
    float x, y;
    float modul();
    complex(float v1, float v2);
    complex();
    complex operator+(complex z);
    complex operator-(complex z);
    complex operator*(complex z);
    complex operator/(complex z);
};

complex::complex()
{ x=0; y=0; }

complex::complex(float v1, float v2)
{ x=v1; y=v2; }

complex complex::operator+(complex z)
{ complex t;
    t.x=x+z.x; t.y=y+z.y;
    return t;
}

complex complex::operator-(complex z)
{ complex t;
    t.x=x-z.x; t.y=y-z.y;
    return t;
}

complex complex::operator*(complex z)
{ complex t;
    t.x=x*z.x-y*z.y;
    t.y=x*z.y+z.x*y;
    return t;
}

complex complex::operator/(complex z)
{ complex t;
    float num=z.x*z.x+z.y*z.y;
    if (num)
        { t.x=(x*z.x+y*z.y)/num;
          t.y=(z.x*y-x*z.y)/num;
        }
    else cout<<"impartire la 0";
    return t;
}

float complex::modul()
{ return sqrt (x*x+y*y);
}
```

- Se citesc z_1 , z_2 numere complexe și n număr natural. Se cere să se scrie un program care calculează $(z_1/z_2)^n$.

```
#include <iostream.h>
#include "complex.cpp"

main()
{ complex z1,z2,z3,z4(1,0);
    int n,i;
    cout<<"n=">>n;
    cout<<"z1"<<endl; cin>>z1.x>>z1.y;
    cout<<"z2"<<endl; cin>>z2.x>>z2.y;
    z3=z1/z2;
    for (i=1;i<=n;i++) z4=z4*z3;
    cout<<z4.x<<" "<<z4.y;
}
```

Observații importante:

- Funcțiile obținute prin supraîncărcarea operatorilor au aceeași prioritate și asociativitate cu operatorii respectivi;
- Funcțiile obținute prin supraîncărcarea operatorilor au aceeași n-aritate ca și operatorii respectivi. Astfel:
 - Dacă operatorul este unar și funcția este unară, deci nu are parametri;
 - Dacă operatorul este binar și funcția este binară. Aceasta înseamnă că are un singur parametru.
- Operatorul ":" nu poate fi supraîncărcat.
- În cazul supraîncărcării operatorilor "+" și "--" nu se face distincție între aplicarea postfixată și prefixată.

Anumiți operatori prezintă particularități în legătură cu supraîncărcarea. Vom prezenta câțiva dintre aceștia.

□ Supraîncărcarea operatorului [].

Acest operator este binar. De exemplu, $a[3]$ se poate scrie ca $*(&a+3)$.

Vom lua un exemplu, simplu. Dorim să reținem un vector, ca tip **class**. Astfel, o funcție va putea întoarce un vector, ceea ce constituie un avantaj. De asemenea, dorim ca data membru să fie privată și accesul la ea să aibă loc exclusiv prin intermediul unor metode. În aceste condiții obținem programul următor:

```

#include <iostream.h>
class vector
{ int v[100];
public:
    void in(int,int);
    int out(int);
};

void vector::in(int i, int x)
{ v[i]=x; }

int vector::out(int i)
{ return v[i]; }

main()
{ vector a;
    a.in(3,5);
    cout<<a.out(3);
}

```

Observați modul de accesare la unei componente:

- `a.in(3,5);` componenta 3 la valoarea 5.
- `a.out(3);` valoarea reținută de componenta 3.

Orice să spune, accesul la componentele vectorului este deosebit de greoi. Suntem tentați să avem un acces de genul `a[3]`. Aceasta conduce la ideea supraîncărcării operatorului `[]`. Pomin de la forma sa generală:

`expresie1[expresie2]`

Funcția corespondentă ei este `operator[]`. Vom înlocui `expresie1` cu obiectul curent, iar `expresie2` cu indicele de adresare. Metoda `operator[]` va întoarce un sinonim al componentei accesate. Acestui sinonim îl se poate atribui orice valoare, sau poate fi citit. Exemplul de mai jos este semnificativ:

```

#include <iostream.h>
class vector
{ int v[100];
public:
    int& operator[](int i);
};

int& vector::operator[](int i)
{ return v[i]; }

main()
{ vector a;
    a[3]=5;
    cout<<a[3];
}

```

□ Supraîncărcarea operatorului `"()"`.

De regulă, operatorul `"()"` este supraîncărcat în scopul permiterii accesului simplificat la datele unui obiect. În exemplul următor, supraîncărcăm operatorul `"()"` pentru a avea acces simplificat la elementele unei matrice. Exemplul următor este semnificativ. Se accesează o matrice astăzi cum suntem obișnuiți de la matematică.

```

#include <iostream.h>
class matrice
{ int v[10][5];
public:
    int& operator()(int i, int j);
};

int& matrice::operator()(int i, int j)
{ return v[i][j]; }

main()
{ matrice a;
    a(3,2)=71;
    cout<<a(3,2);
}

```

□ Supraîncărcarea operatorului `"=`.

Se pot face atribuiri între obiecte, astăzi cum suntem obișnuiți. În astfel de cazuri avem o copiere bit de bit. Mecanismul este simplu: fiecare dată că obiectul afiat în dreapta operatorului este copiată către data corespunzătoare afiată în stânga.

Nu întotdeauna o astfel de atribuire este satisfăcătoare. Pentru a demonstra aceasta vom pomi de la un exemplu. Programul următor utilizează o clasă numită `string`. Clasa va fi construită în scop efectuării anumitor operații cu siruri de caractere. Din acest motiv, fiecare obiect conține adresa din `HEAP` a sirului atașat, dar nu conține sirul.

Fie două obiecte `a` și `b`. Presupunem că `a` reține adresa sirului "un text", iar `b` reține adresa sirului "alt text". Scriem expresia `a=b`. În ipoteza în care operatorul `'='` nu este supraîncărcat obiectul `a` va reține adresa sirului atașat lui `b`. Pe de o parte, sirul care avea adresa în obiectul `a` se pierde, dar rămâne alocat în `HEAP`, chiar dacă nu mai avem adresa lui. Pe de altă parte, `a` și `b` vor reține adresa unui singur sir. Dacă, de exemplu, stergem pe `b`, `a` va reține adresa unui sir inexistent.

Datorită celor arătate, înzestrăm clasa cu un constructor de copiere care efectuează următoarele:

- alocă spațiu în HEAP -un număr de octeți care să permită copierea;
- copiază sirul.

Tot așa, supraîncărcăm operatorul “=” cu o metodă care efectuează următoarele:

- eliberează spațiu ocupat de sirul a;
- alocă spațiu în HEAP -un număr de octeți care să permită copierea;
- efectuează copierea.

```
#include <iostream.h>
#include <string.h>

class string
{
    char* adr;
public:

    // constructor
    string(char sir[])
    {
        adr=new char[strlen(sir)+1];
        strcpy(adr,sir);
    }

    // constructor de copiere
    string(string& x)
    {
        adr=new char[strlen(x.adr)+1];
        strcpy(adr,x.adr);
    }

    void operator=(string& x)
    {
        delete adr;
        adr=new char[strlen(x.adr)+1];
        strcpy(adr,x.adr);
    }

    void tip()
    {
        cout<<adr<<endl;
    }
};

main()
{
    string a("mama"),b("bunica"),c=a;
    a.tip();b.tip();c.tip();
    a=b;a.tip(); b.tip()
}
```

5.2.4 Funcții prieten

Există posibilitatea ca anumite funcții, care nu aparțin unei clase să poată accesa datele private ale acesteia, caz în care funcțiile respective se numesc “funcții prieten”.

Pentru a ataşa unei clase o funcție prieten introducem în interiorul definirii sale prototipul funcției prieten, precedat de cuvântul cheie **friend**.

O funcție prieten nu poate accesa datele direct, așa cum poate o metodă a clasei respective, ci prin intermediul parametrului de tip obiect transmis.

Exemplu. Clasa **complex**, definită mai jos, are funcția prieten **modul()**, care returnează modulul numărului **complex**.

```
#include <iostream.h>
#include <math.h>

class complex
{
    float x,y;
public:
    complex(float, float);
    friend float modul(complex& z);
};

complex::complex(float v1, float v2)
{ x=v1;y=v2; }

float modul(complex& z)
{ return sqrt(z.x*z.x+z.y*z.y); }

main()
{
    complex z(3,4);
    cout<<modul(z);
}
```

În C++ operatorii pot fi supraîncărajați prin funcții prieten.

Exemplu. Dorim să efectuăm suma dintre un număr complex și unul real. Pentru aceasta vom folosi o funcție prieten care supraîncarcă operatorul **+**. Pentru ca operația să fie comutativă, adică să se poată face și suma dintre un număr real și un număr complex, vom folosi o altă funcție prieten, în care ordinea parametrilor este inversă față de cea anterioară. Vezi exemplul următor:

```
#include <iostream.h>
#include <math.h>
```

```

class complex
{
    float x,y;
public:
    complex(float, float);
    friend float modul(complex& z);
    friend complex operator+(complex& z, float nr);
    friend complex operator+(float nr, complex& z);
};

complex::complex(float v1, float v2)
{ x=v1; y=v2; }

float modul(complex& z)
{ return sqrt(z.x*z.x+z.y*z.y); }

complex operator+(complex& z, float nr)
{ complex t(z.x+nr, z.y);
    return t;
}

complex operator+(float nr, complex& z)
{ complex t(z.x+nr, z.y);
    return t;
}

main()
{
    complex z(3,4);
    z=z+7; z=3+z;
    cout<<modul(z);
}

```

5.2.5 Aplicații ale încapsulării

5.2.5.1 Multimi

Limbajul C++ nu are un tip special destinat lucrului cu multimi.

În acest paragraf vom crea un tip obiect, numit **Multime** cu ajutorul căruia putem lucra cu multimi de numere naturale între 0 și 799!

Cum gândim crearea unui astfel de tip?

- Pentru a memora dacă un anumit element aparține sau nu unei multimi este suficient un bit. Cum un octet are 8 biți, pentru a putea reține o multime cu 800 elemente sunt necesari 100 de octeti. Prin urmare, un astfel de obiect contine o variabilă, numită **Mult** cu 100 de octeti (componente de tip **char**). Pentru memorare utilizăm mecanismul prezentat în manualul pentru clasa **a X-a**, dar poate fi cu ușurință dedus din codul sursă al metodelor utilizate.

- Orice lucru cu o mulțime presupune posibilitatea de adăugare a unui anumit element mulțimii (de a face reuniunea dintre o mulțime dată și una care conține un singur element). Aceasta este făcută în mod direct prin intermediul funcției de tip **operator+** rezultată prin supraîncărcarea operatorului **+** cu doi parametri, unul de tip **Multime**, transmis prin referință, și altul de tip **int**. Întrucât operația este comutativă, vom utiliza, de fapt, două funcții care diferă doar prin ordinea parametrilor.
- Pentru a putea testa dacă un anumit element aparține sau nu mulțimii considerate, este necesară existența unei metode, numită **in**, cu un singur parametru, care conține elementul ce se testează dacă aparține sau nu mulțimii.
- Pentru a putea inițializa o mulțime cu mulțimea vidă vom utiliza un constructor. El are rolul de a inițializa obiectul cu mulțimea vidă.
- Pentru a putea efectua operațiile cunoscute cu mulțimi vom utiliza metode rezultate prin supraîncărcarea operatorilor **+** pentru reuniune, - pentru diferență, și ***** pentru intersecție.

```

class Multime
{
    char Mult[100];
public:
    Multime();
    int in (int n);
    Multime operator+(Multime& M);
    Multime operator-(Multime& M);
    Multime operator*(Multime& M);
    friend Multime operator+(Multime& M, int n);
    friend Multime operator+(int n, Multime& M);
};

Multime::Multime()
{
    for (int i=0;i<100;i++) Mult[i]=0;
}

int Multime::in(int n)
{
    unsigned char Nr_Bit=7-n%8,masca=1;
    int Nr_Octet=n/8;
    masca=masca<<Nr_Bit;
    if (Mult[Nr_Octet] & masca) return 1;
    else return 0;
}

Multime operator+(Multime& M, int n)
{
    Multime M1=M;
    unsigned char Nr_Bit=7-n%8,masca=1;
    int Nr_Octet=n/8;
    masca=masca<<Nr_Bit;
    M1.Mult[Nr_Octet]=M1.Mult[Nr_Octet] | masca;
    return M1;
}

```

```

Multime operator+(int n,Multime& M)
{
    Multime M1=M;
    unsigned char Nr_Bit=7-n%8,masca=1;
    int Nr_Octet=n/8;
    masca=masca<<Nr_Bit;
    M1.Mult[Nr_Octet]=M1.Mult[Nr_Octet] | masca;
    return M1;
}

Multime Multime::operator+(Multime& M)
{
    Multime M1;
    int i;
    for (i=0;i<100;i++) M1.Mult[i]=Mult[i] | M.Mult[i];
    return M1;
}

Multime Multime::operator-(Multime& M)
{
    Multime M1;
    int i;
    for (i=0;i<100;i++)
        M1.Mult[i]=Mult[i] & (Mult[i] ^ M.Mult[i]);
    return M1;
}

Multime Multime::operator*(Multime& M)
{
    Multime M1;
    int i;
    for (i=0;i<100;i++) M1.Mult[i]=Mult[i] & M.Mult[i];
    return M1;
}

```

- Aplicația 1. Se generează două multimi și se afișează reuniunea lor.

```

#include <iostream.h>
#include "multime.cpp"
main()
{
    Multime M,M1,M2;
    int i;
    M=M+15; M=M+7; M=M+23; M=17+M; M=M+799;
    M1=M1+11; M1=7+M1; M1=0+M1; M1=1+M1;
    M=M+M1;
    for (i=0;i<800;i++)
        if (M.in(i)) cout<<i<<" ";
}

```

- ✓ Există și dezavantaje ale lucrului cu multimi de până la 800 elemente. Spațiul necesar memorării este mai mare, ca, de altfel, și efortul de calcul!
- Aplicația 2. Se citește un sir de caractere. Se cere să se tipărească literele distincte care apar în sir.

De această dată este necesar să memorăm caractere, nu numere. Dar, acesta nu-i o problemă. Pentru fiecare caracter se memorează codul său, fără nici o altă modificare, iar la afișare se tipărește caracterul (se convertește codul în caracter prin utilizarea operatorului de conversie explicită).

```

#include "multime.cpp"
main()
{
    Multime M;
    int i;
    char sir[20];
    cin.get(sir,20);
    i=0;
    while (sir[i])
        if (sir[i]!=' ') M=M+sir[i];
        i++;
    for (i=0;i<256;i++)
        if (M.in(i)) cout<<(char)i;
}

```

5.2.5.2* Matrice rare

Este cunoscut faptul că matricele ocupă mult spațiu în memorie. Uneori, spațiul ocupat de o matrice poate fi micșorat. O astfel de situație este aceea în care matricea are majoritatea elementelor nule.

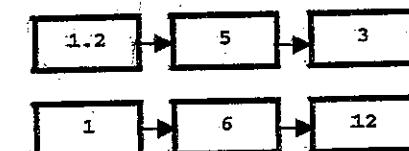
Definție. Se numește matrice rară o matrice în care majoritatea elementelor sunt nule.

Vă dați seama că dacă am memoria matricea astă cum suntem obișnuiți, spațiul ocupat de ea în memorie este mare și, în plus, în cazul elementelor nule este ocupat degeaba. Din acest motiv vom utiliza un alt mod de memorare a matricelor rare.

➤ O matrice rară va fi memorată cu ajutorul a două liste liniare simplu înlántuite, una conținând valorile nenule, alta numărul de ordine al lor, număr rezultat prin parcursarea pe linii a matricei.

Exemplu: Matricea următoare se memorează astă cum se vede:

$$A = \begin{pmatrix} 1.2 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$



- ✓ Deși avantajoasă din punctul de vedere al memoriei ocupate, în astfel de memorare prezintă două dezavantaje: prelucrarea necesită un timp mare și manipularea matricelor este greoie pentru utilizator.

Ultimul dezavantaj poate fi eliminat dacă se apelează la mecanismul **copy**. Pentru aceasta vom crea un obiect de tipul **Matrice**, care reține matricele, memorate astă cum am arătat. Obiectele de acest tip sunt descrise în modulul **Matrice**. Cum gândim crearea unui astfel de obiect?

1. În primul rând, obiectul trebuie să rețină numărul de linii (**m**) și de coloane (**n**) ale matricei.
2. De asemenea, obiectul trebuie să rețină adresele de început ale celor două liste liniare. Listele sunt reținute în **HEAP**, deci nu în cadrul obiectului.
3. Constructorul **Matrice(int lin, int col)** are rolul de a inițializa o matrice rară în care toate elementele sunt nule. De asemenea, ea este utilizată pentru a memora numărul de linii (**m**) și coloane (**n**) ale matricei.
4. Constructorul **Matrice()** se utilizează pentru declararea unui obiect **Matrice** care urmează să fie citit dintr-un fișier text.
5. Metoda **operator()(i,j,elm)** adaugă unei matrice un element **elm** (nenul) care se găsește în linia **i** și coloana **j**.
6. Metoda **operator()(i,j)** returnează valoarea elementului aflat pe linia **i** și coloana **j** a matricei. Mai întâi funcția calculează poziția elementului **n*(i-1)+j** și o cauță în prima listă. Dacă este găsită, înseamnă că elementul este nenul, prin urmare returnează valoarea aflată în cealaltă listă în aceeași poziție. Dacă nu este găsită, înseamnă că elementul respectiv este nul, deci returnează 0.
7. Metoda **Citesc(Nume_fis)** citește matricea dintr-un fișier text. Prima linie va conține numărul de linii și numărul de coloane ale matricei și fiecare din următoarele linii reține linia, coloana și valoarea unui element nenul. Pentru a memora elementul citit se apelează metoda **operator()(int i,j,elm)**.

Prezentăm modulul care conține clasa **Matrice**:

```
#include<fstream.h>
struct Nod
{
    int val;
    Nod* adr_urm;
};
```

```
struct Nod1
{
    float val;
    Nod1* adr_urm;
};

class Matrice
{
    Nod* v;
    Nod1* v1;
public:
    int m,n;
    Matrice(int lin, int col);
    Matrice();
    void operator()(int i,int j, float elm);
    float operator()(int i,int j);
    void Citesc(char Nume_fis[20]);
};

Matrice::Matrice(int lin, int col)
{
    m=lin;n=col;
    v=0;v1=0;
}

Matrice::Matrice()
{
    v=0;v1=0;
}

void Matrice::operator()(int i,int j, float elm)
{
    Nod* man=v, *c;
    Nod1* man1=v1,*cl;
    int Poz=n*(i-1)+j;
    while (man && man->val!=Poz)
    {
        man=man->adr_urm;
        man1=man1->adr_urm;
    }
    // pana aici am vazut daca elementul este in lista;
    if (man->val==Poz) // daca este in lista
        if (elm==0) // daca trebuie sters
            if (man==v) // daca este primul
                { c=man; v=man->adr_urm; delete c;
                  c1=man1; v1=man1->adr_urm; delete c1;
                }
            else // daca nu este primul
                { man=man->adr_urm;
                  man1=man1->adr_urm;
                }
        c=man->adr_urm; c1=man1->adr_urm;
        man->adr_urm=man->adr_urm->adr_urm;
        man1->adr_urm=man1->adr_urm->adr_urm;
        delete c; delete c1;
    }
    else // daca nu trebuie sters;
        man1->val=elm;
    else //daca este diferit de 0 il adaug in lista
}
```

```

if (elm)
{
    man=v; v->val=Poz; v->adr_urm=man;
    man1=v1; v1=new Nod1; v1->val=elm; v1->adr_urm=man1;
}

float Matrice::operator()(int i,int j)
{
    Nod* man=v;
    Nod1* man1=v1;
    int Poz=n*(i-1)+j;
    while (man && man->val!=Poz)
    {
        man=man->adr_urm;
        man1=man1->adr_urm;
    }
    if (man->val==Poz) return man1->val;
    else return 0;
}

void Matrice::Citesc(char Nume_fis[20])
{
    int i,j;
    float t;
    fstream f(Nume_fis,ios::in);
    f>>n;
    while (f>>i>>j>>t) operator()(i,j,t);
    f.close();
}

```

- Aplicația 1. Să se citească o matrice rară dintr-un fișier text și să se afișeze pe monitor.

```

#include "matrice.cpp"

main()
{
    Matrice A;
    int i,j;
    A.Citesc("Mat.dat");
    for (i=1;i<=A.m;i++)
    {
        for (j=1;j<=A.n;j++) cout<<A(i,j);
        cout<<endl;
    }
}

```

- Aplicația 2. Să se citească o matrice și să se inverseze două coloane de indice dat:

```

#include "matrice.cpp"
Matrice A;
main()
{
    int i,j,c1,c2;
    float man;
    A.Citesc("Mat.dat");
    cout<<"c1=";cin>>c1;
    cout<<"c2=";cin>>c2;
    cout<<endl;
    for (i=1;i<=A.m;i++)

```

```

    {
        for (j=1;j<=A.n;j++) cout<<A(i,j);
        cout<<endl;
    }

    for (j=1;j<=A.n;j++)
    {
        man=A(j,c1);A(j,c1,A(j,c2));A(j,c2,man);
    }

    for (i=1;i<=A.m;i++)
    {
        for (j=1;j<=A.n;j++) cout<<A(i,j);
        cout<<endl;
    }
}

```

- ✓ Observați că dacă se utilizează obiectul **Matrice** lucrul cu matricele devine foarte ușor pentru utilizator, iar mecanismul de memorare și regăsire a matricei devine "transparent" pentru programatorul utilizator.

5.3 Mostenirea

5.3.1 Noțiuni de bază care privesc mostenirea

Definiție. Prin mostenire se înțelege acea proprietate a claselor prin care un tip nou construit poate prelua datele și metodele unui tip mai vechi. În C++ acest mecanism mai este cunoscut sub numele de "derivare a claselor".

- ✓ Evident, clasei nou construite î se pot adăuga noi date și metode!
- ✓ Prin acest procedeu se preia soft deja făcut care se dezvoltă. Avantajul urias al acestui procedeu este că persoana care preia un anumit soft trebuie să cunoască doar documentația de utilizare a soft-ului preluat.

Ideea de bază este următoarea: fiind date clasele x_1, x_2, \dots, x_n , putem construi clasa x_{n+1} care are, pe lângă datele și metodele claselor enumerate, date și metode proprii. Clasele x_1, x_2, \dots, x_n se numesc clase de bază, iar clasa x_{n+1} clasă derivată.

Exemplul 1. Clasa **numar** conține o dată membru **n** și o metodă membru **tip()**. Clasa **numar_d** este derivată din clasa **numar**. Prin urmare, un obiect obținut prin instantierea acesteia, are data membru **n** și metoda **tip()**. Pe lângă acestea, clasa **numar_d** are o nouă metodă **tip()**. În paragraful următor vom analiza rolul modificatorului **public** din:

```
class numar_d : public numar
```

```

#include <iostream.h>

class numar
{
public:
    int n;
    void tip()
    {
        cout<<n<<endl;
    }
};

class numar_d : public numar
{
public:
    void cit()
    {
        cout<<"n=">>n;
    }
};

main()
{
    numar_d a;
    a.cit(); a.tip();
}

```

Exemplul 2. Clasa `Complex` ne permite să lucrăm ușor cu numere complexe. În timpul utilizării acestei clase ne-am dat seama că ar fi bine să conțină o metodă booleană numită `real()`, care ne permite să testăm dacă numărul `complex` este `real` sau nu. Dispunem de documentația clasei `Complex`. Nu-i o problemă să definim o clasă nouă, numită `Complex_d`, care conține datele și metodele clasei `Complex`, dar care include și noua metodă.

```

#include <iostream.h>
#include "complex.cpp"

class complex_d : public complex
{
public:
    int real();
};

int complex_d::real()
{
    return y==0;
}

main()
{
    complex_d z;
    z.x=3; z.y=1;
    if (z.real()) cout<<"numarul este real";
    else cout<<"numarul nu este real";
}

```

Întrebarea 1. Se pot face atribuiri între obiectele unei clase derivate și obiectele clasei de bază?

⇒ Unui obiect al clasei de bază î se poate atribui un obiect al unei clase care este derivată din ea.

⇒ Unui obiect al unei clase derivate î se poate atribui un obiect al clasei de bază.

Motivul? Se presupune că o clasă derivată are date în plus. Ce valori ar primi acestea în cazul unei astfel de atribuiri?

✓ Nimeni nu ne opreste să înzestrăm clasa derivată cu o metodă obținută prin supraîncărcarea operatorului `=`. Dar și datele care aparțin exclusiv clasei derivate îau valori arbitrate (în exemplul următor ①).

Exemplu:

```

#include <iostream.h>

class numar
{
public:
    int m;
};

class numar_d : public numar
{
public:
    int n;
    void operator=(numar& x)
    {
        m=x.m; n=0;
    }
};

main()
{
    numar a;
    numar_d b;
    b.m=3;b.n=4;
    a=b;
    cout<<a.m<<endl;
    b=a; //nu ar fi permis
    cout<<b.m<<" "<<b.n;
}

```

Întrebarea 2. Se stie că funcțiile pot fi supraîncărcate. Dar o clasă derivată poate conține o funcție cu același nume și cu aceeași listă de parametri formali ca una a clasei de bază?

Răspunsul este afirmativ. În exemplul următor clasa de bază (`numar`) are o metodă numită `tip()`, în care lista parametrilor formali este vidă. Clasa derivată are pe lângă metoda `tip()`, moștenită, o alta numită tot `tip()`, cu aceeași listă a parametrilor formali (tot vidă). Prin `a.tip()` se apelează metoda `tip()` proprie clasei derivate, în cazul în care dorim să apelăm metoda `tip()` a clasei de bază scriem: `a.numar::tip()`. Cele două metode cu același nume diferă doar prin mesajul afișat înaintea tipăririi valorii reținute de variabilă.

```

#include <iostream.h>

class numar
{ public:
    int i;
    void tip()
    { cout<<"numar "<<i<<endl; }
};

class numar_d :public numar
{ public:
    void tip()
    { cout<<"numar_d "<<i<<endl; }
};

main()
{ numar_d a;
    a.i=3;
    a.tip();
    a.numar::tip();
}

```

Întrebarea 3. Ce se întâmplă în situația în care în clasa derivată se definește o nouă dată cu același nume ca una a clasei de bază?

În acest caz nu se semnalează eroare. Un obiect rezultat ca instanțiere a unei clase derivate va avea două date cu același nume. Implicit se adresează variabila definită în clasa derivată, dar este posibil să adresăm și data clasei de bază la fel ca în cazul funcțiilor. Analizați exemplul următor:

```

#include <iostream.h>
class numar
{ public:
    int i;
};

class numar_d :public numar
{ public:
    int i;
};

main()
{ numar_d a;
    a.i=2; a.numar::i=3;
    cout<<a.i<< " " <<a.numar::i;
}

```

5.3.2* Accesul la membri unei clase

Membrii unei clase (date și metode) sunt de 3 feluri:

1. Publici - precedați de cuvântul cheie **public**: Aceștia pot fi accesati din exteriorul clasei.
2. Privati - precedați de cuvântul cheie **private**: Să nu uităm că, implicit, membrii unei clase sunt privați. Membrii privați nu pot fi accesati decât din interiorul clasei respective.
3. Protejați - precedați de cuvântul cheie **protected**: Până în prezent aceștia nu au fost prezențați. Rolul lor apare în momentul în care o clasă este derivată. Ei nu pot fi accesati din exterior (întocmai ca membrii privați) dar pot fi accesati din clasele deriveate.

Există 3 modalități prin care o clasă poate fi moștenită (derivată):

1. class numar_d :public numar
 {
 ...
 };
2. class numar_d :private numar
 {
 ...
 };
3. class numar_d :protected numar
 {
 ...
 };

1. A fost moștenită precedată de cuvântul cheie **public**:
 - membrii publici ai clasei de bază rămân publici pentru clasa derivată;
 - membrii privați ai clasei de bază rămân privați pentru clasa derivată;
 - membrii protejați ai clasei de bază rămân protejați pentru clasa derivată.
2. A fost moștenită precedată de cuvântul cheie **private**. În această situație clasa derivată are toți membrii privați. În schimb, din interiorul clasei deriveate putem accesa membrii publici și protejați ai clasei de bază.

3. A fost moștenită precedată de cuvântul cheie `protected`. În această situație, membrii publici și protejați ai clasei de bază sunt protejați pentru nouă clasă, iar cei privați ai clasei de bază rămân privați și pentru clasa derivată. Din interiorul clasei derivate putem accesa membrii publici și privați ai clasei de bază.

✓ Nu este obligatoriu să retineți aceste notiuni. Ele au fost introduse în dorința de a vă pune la dispoziție o documentație.

Dăm forma generală prin care se obține o clasă derivată care are la bază n clase.

```
class nume : (public sau protected sau private) nume_1  
           : (public sau protected sau private) nume_2,  
...  
           : (public sau protected sau private) nume_n  
{  
---  
}
```

5.3.3 Din nou despre constructori și destrutori*

Întrebarea la care răspundem pe parcursul acestui paragraf este următoarea: care este comportamentul constructorilor și ai destruitorilor în cazul în care clasa este moștenită (derivată)?

Priviți exemplul următor. Clasa `numar_d` moștenește clasa `numar`. Fiecare dintre ele este înzestrată cu câte un constructor. Cei doi constructori diferă doar prin sirul afisat. La declararea unui obiect care instantiază clasa derivată se tipărește "numar" apoi "numar_d".

```
#include <iostream.h>  
  
class numar  
{ public:  
    numar()  
    {cout<<"numar "<<endl;}  
};  
  
class numar_d : public numar  
{ public:  
    numar_d()  
    {cout<<"numar_d";}  
};  
  
main()  
{ numar_d a; }
```

⇒ La declararea unui obiect se apelează mai întâi constructorul clasei de bază, apoi cel al clasei derivate.

Problema nu este asa de simplă. Să ne gândim la faptul că o clasă poate fi înzestrată cu mai mulți constructori. Dacă ne gândim la faptul că și clasa derivată poate fi înzestrată cu mai mulți constructori, lucrurile se complică. Pentru a ne îmuri, trebuie analizat exemplul următor.

```
#include <iostream.h>  
  
class numar  
{ public:  
    numar()  
    {cout<<"numar "<<endl;}  
    numar(int k)  
    {cout<<"numar "<<k<<endl;}  
};  
  
class numar_d : public numar  
{ public:  
    numar_d()  
    {cout<<"numar_d "<<endl;}  
    numar_d(int k) : numar(k)  
    {cout<<"numar_d "<<k<<endl;}  
};  
  
main()  
{ numar_d a,b(1); }
```

Clasa `numar` are doi constructori, unul fără parametri, al doilea cu un parametru implicit. În mod asemănător, clasa derivată, `numar_d` are, de asemenea, doi constructori, unul fără parametri și unul cu un parametru implicit.

Un constructor al clasei derivate, mai întâi apelează constructorul fără parametri al clasei de bază, apoi efectuează propriile operații. Apelul constructorului clasei de bază se face automat, fără ca programatorul să scrie ceva. În cazul în care acesta dorește să fie apelat un anumit constructor, altul decât cel fără parametri, se procedează ca în exemplul precedent, unde constructorul cu un parametru al clasei derivate apelează pe cel cu un parametru al clasei de bază:

```
numar_d(int k) : numar(k)  
{cout<<"numar_d "<<k<<endl;}
```

Programul tipărește: numar, numar_d, numar 1, numar_d 1.

În ce privește destrutorii, acestia se apelează în ordinea inversă în care au fost apelati constructorii. Programul următor tipărește `numar`, `numar_d`, `numar_d destructor`, `numar destructor`.

```
#include <iostream.h>

class numar
{ public:
    numar()
    {cout<<"numar "<<endl;}
    ~numar()
    {cout<<"numar destructor "<<endl;}
};

class numar_d :public numar
{ public:
    numar_d()
    {cout<<"numar_d "<<endl;}
    ~numar_d()
    {cout<<"numar_d destructor "<<endl;}
};

main()
{ numar_d b; }
```

5.3.4 O aplicație a moștenirii

Înainte de a proiecta o clasă trebuie făcută o analiză atentă asupra folosirii ei, pentru ca aceasta să fie concepută de la început aşa cum trebuie.

Analiza presupune multă experiență din partea celui care o face. Persoana care efectuează analiza trebuie să cunoască bine domeniul unde se va aplica clasa respectivă, dar și programarea. Trebuie să aibă capacitatea de a sinteza a necesităților. Din acest motiv ea este cu mult mai bine plătită decât un programator! În informatică o astfel de persoană se numește analist de sistem.

În realitate, cu toate măsurile luate, de multe ori se impun anumite îmbunătățiri ale tipului considerat.

În astfel de cazuri avem două posibilități:

1. Se reconcepe de la bază totul. Această soluție, desigur, este mai avantajoasă din punct de vedere al exploatarii, este costisitoare, pentru că presupune efort finanic și consum de timp.

2. O altă soluție este de a utiliza mecanismul moștenirii (evidenț, dacă am utilizat tehnica OOP). Această soluție este avantajoasă în ce privește banii și timpul, dar, de multe ori conține și anumite neajunsuri, ca:
 - în urma aplicării repetate a moștenirii se obțin obiecte care ocupă multă memorie, dar nu orice contin ele este util în totalitate;
 - timpul de calcul este mai mare, pentru că anumite secvențe se apelează în mod inutile;
 - de multe ori chiar proiectantii "scapă de sub control" propriul produs, iar acesta devine un obiect "nesigur", un "monstru cu picioare de lut".

Din cele prezentate n-ar trebui să se înțeleagă că OOP este mai degrabă dăunătoare decât utilă. Tehnica poate fi utilizată și pentru dezvoltarea programelor, de la simplu la complex. Nimic în lumea asta nu este perfect, nici chiar OOP.

În urma utilizării obiectelor `Multime` s-a constatat că ar mai fi nevoie de două metode:

1. Un constructor care să permită citirea unei multimi dintr-un fisier text, în care elementele se găsesc toate pe o linie.
2. O metodă care să returneze numărul de elemente ale multimii, numită `Card()`.

Pentru aceasta vom construi o nouă clasă numită `Multime_g`, care moștenește clasa `Multime`. La noul tip de obiect se adaugă metodele de mai sus și se obține:

```
#include <fstream.h>
#include "multime.cpp"
class Multime_g:public Multime
{ public:
    Multime_g(char Nume_fis[20]);
    Multime_g();
    int Card();
};

Multime_g::Multime_g(char Nume_fis[20])
{ int n,Nr_octet;
  unsigned char Nr_bit,masca;
  ifstream f(Nume_fis, ios::in);
  while (f>>n)
  { Nr_bit=7-n%8;masca=1; Nr_octet=n/8; masca=masca<<Nr_bit;
    Mult[Nr_octet]=Mult[Nr_octet] | masca;
  }
}
```

```

int Multime::Card()
{
    int s=0,i;
    for (i=0;i<800;i++)
        if (A.in(i)) s++;
    return s;
}

```

- Aplicatie. Programul următor citeste o mulțime dintr-un fișier text, și afisează și tipărăște numărul de elemente pe care le conține mulțimea.

```

#include "multimeg.cpp"
Multime A("Int.dat");

main()
{
    for (int i=0;i<800;i++)
        if (A.in(i)) cout<<i<<endl;
    cout<<A.Card();
}

```

- ✓ Nu era mai bine dacă obiectul de tip mulțime ar fi reținut ca dată numărul de elemente ale mulțimii? În acest caz tipărarea ar fi fost imediată.
- ✓ Desigur, în astfel de cără ar fi putut fi introdusă la acest pas. Totuși apare o problemă: numărul de elemente ale unei mulțimi se modifică în urma operațiilor de reuniune, intersecție etc., operații care se pot efectua. În acest caz, ar trebui să modificăm toate acele funcții. Dar operația necesită timp și este posibil, ca în practică să nu disponem de sursa lor. Prin urmare am ales soluția cea mai comodă de a lăta numărul, dar această soluție nu este cea mai eficientă. Ce contează, veți spune "Merge" și aşa repede... Așa este, dar în practică, dacă se recurge tot timpul la astfel de soluții, programele devin ineficiente.
- ✓ Dacă, prin absurd, cineva ar prelua acest tip și îl ar dezvolta, ar obține un altul. Dar, îl preia cu tot ce conține el înșicent, iar prin ce face să ar putea să introducă alte elemente de înșicentă. Si tot aşa...

5.4 Polimorfism

Polimorfismul este legat de moștenire. Să vedem în ce constă problema.

Fie o clasă **O1** care conține o metodă **M1** care o apelează pe alta **M2** din aceeași clasă. Până aici nimic deosebit, este un caz pe care l-am întâlnit. O altă clasă **O2** o moștenește pe prima. În cadrul acesteia redefinim metoda **M2**, dar nu redefinim metoda **M1**. Declaram un obiect de tip **O2** și apelăm metoda **M1**. Problema este următoarea: care va fi metoda apelată de **M1**, **M2** din **O1** sau **M2** din **O2**?

- În situația în care procedăm ca până acum va fi apelată metoda **M2** din **O1**.

Exemplu: fie modulul următor, care exemplifică situația prezentată: metoda **care()** apelează metoda **tip()**, redefinită.

```

#include <iostream.h>
class unu
{
public:
    void care()
    {
        tip();
    }
    void tip()
    {
        cout<<"unu"<<endl;
    }
};

class doi: public unu
{
public:
    void tip()
    {
        cout<<"doi"<<endl;
    }
};

unu x;
doi y;

main()
{
    x.care();
    y.care();
}

```

- ✓ Evident, în astfel de situație nu convine. De ce am redefinit metoda **tip()** dacă este apelată tot cea veche?

- O astfel de metodă de selecție a metodei prin obiectul care o apelează se numește selecția statică. Codul de apel al funcției este obținut la compilare și rămâne nemodificat pe parcursul executării programului.
- ✓ Atunci când a fost compilat obiectul număr, deci și metoda **care()**, ea apela metoda **tip()** acolo definită. Prin urmare, aceasta explică situația neplăcută.

Cum se poate remedia situația, adică să fie apelată metoda **tip()** redefinită?

- Există posibilitatea ca selecția să se facă în momentul apelului (nu la compilare). În astfel de cazuri programatorul poate decide care metodă să fie selectată. O selecție de acest tip se numește selecție dinamică sau virtuală.

Pentru a realiza selecția dinamică trebuie să procedăm astfel:

- ⇒ când se declară, în cadrul clasei de bază, o metodă ce va fi redefinită, se adaugă după declaratiele cuvântul **VIRTUAL**;
- ⇒ orice redefinire a metodelor, în cadrul claselor derivate (descendente), se va face utilizând același cuvânt cheie **VIRTUAL**;

```
#include <iostream.h>
class unu
{
public:
void care()
{ tip(); }
virtual void tip()
{ cout<<"unu"<<endl; }
};

class doi: public unu
{
virtual void tip()
{ cout<<"doi"<<endl; }
};

unu x;
doi y;

main()
{
x.care();
y.care();
}
```

De această dată, selectia se va face corect, adică se apelează metoda **tip()** redefinită în clasa **doi**.

5.4.1 O aplicatie a polimorfismului

Anul trecut am studiat backtracking nerecursiv standardizat. Am văzut acolo că o unică secvență, apela anumite funcții care întotdeauna aveau același nume, dar care se modificau de la un program la altul.

- Problema este următoarea: cum să facem ca rutina unică să fie scrisă o singură dată, iar funcțiile **init()**, **succesor()**... să fie redefinite pentru fiecare caz în parte. Evident, toate acestea prin utilizarea mecanismului OOP.

Care este dificultatea? Ea constă în faptul că funcția de bază, pe care o vom numi **Run()** apelează rutinele **succesor()**, **init()** etc. Aceasta

înseamnă că obiectul care îl reține trebuie să contină și aceste subprograme. Apoi, acestea vor fi redefinite. După redefinire, trebuie ca funcția de bază să le apeleze pe cele redefinite. Prin urmare, este necesar să se aplică polimorfismul.

Initial, declarăm clasa **bkt**, în care funcțiile **init()**, **succesor()** etc. sunt declarate ca virtuale. Întrucât ele nu vor fi apelate niciodată din cadrul acestei clase, au "corpu" vid. De asemenea, variabilele **st**, **As**, **E** care se găsesc în orice program backtracking, vor fi declarate aici.

```
class bkt
{ public:
int st[10],n,k;

virtual void Init(){} //init()

virtual int Am_Succesor ()
{ return 0; }

virtual int E_Valid()
{ return 0; }

virtual int Solutie()
{ return 0; }

virtual void Tipar(){}
void Run();
};

void bkt::Run()
{int As;
k=1; Init();
while (k>0)
{ do {} while ((As=Am_Succesor()) && !E_Valid());
if (As)
if (Solutie()) Tipar();
else {k++;Init();}
else k--;
}
}
```

- Se cere ca, prin utilizarea clasei **bkt** să se genereze toate permutările multimi 1, 2...n.
- Pentru a folosi clasa **bkt** vom defini o alta, care o moștenește și care redefineste funcțiile specifice. Ea va fi înzestrată cu un constructor. Constructorul are și rolul de a-l citi pe n. Astfel se obține tipul următor:

```
#include <iostream.h>
#include "bkt.cpp"
```

```

class permut : public bkt
{
public:
    permut (int v) : n(v) {}
    virtual void Init();
    virtual int Am_Succesor();
    virtual int E_Valid();
    virtual int Solutie();
    virtual void Tipar();
};

void permut::Init()
{
    st[0]=0;
}

int permut::Am_Succesor()
{
    if (st[k]<n)
    {
        st[k]++;
        return 1;
    }
    else return 0;
}

int permut::E_Valid()
{
    for (int i=1;i<k;i++)
        if (st[i]==st[k]) return 0;
    return 1;
}

int permut::Solutie()
{
    return k==n;
}

void permut::Tipar()
{
    for (int i=1;i<=n;i++)
        cout<<st[i];
    cout<<endl;
}

```

De acum, scrierea programului devine o simplă formalitate...

```

#include "grafuri.cpp"
#include "per.cpp"
main()
{
    permut x(3);
    x.run();
}

```

- Prin utilizarea claselor deja create, se cere să se rezolve problema celor n dame.

După cum știm, rezolvarea acestei probleme diferă fără de generarea permutărilor doar printr-o singură metodă: **valid()**. Atunci aceasta va fi redefinită și se obține:

```

#include "per.cpp"
#include <cmath.h>
class dame: public permut
{
public:
    dame( int v): permut(v) {};
    virtual int dame::E_Valid();
};

int dame::E_Valid()
{
    for (int i=1;i<k;i++)
        if (st[i]==st[k] || abs(st[k]-st[i])==abs(k-i)) return 0;
    return 1;
}

```

Scrierea programului de generare a damelor este o simplă formalitate.

```

#include "grafuri.cpp"
#include "dame.cpp"
main()
{
    dame x(8);
    x.run();
}

```

- ✓ Atenție la continutul acestui paragraf. Aceste cunoștințe sunt fundamentale, inclusiv în Borland C++ Builder.

Probleme propuse

1. Descrieți o clasă numită **Rational** care permite lucrul cu numere rationale. Un număr rational este de forma:

$$r = \frac{p}{q}, p, q \in \mathbb{Z}, q \neq 0$$

2. Scrieți funcții de tip prieten care permit efectuarea operațiilor aritmetice cu numere rationale prin utilizarea clasei **Rational** creată anterior.

3. Cititi și sortați descrescător un vector ale cărui componente au tipul **Rational**.

4. Sortați crescător mai multe numere de tip **Rational** prin utilizarea sortării prin inserție.

5. Creati un tip **Object**, numit **Stiva**. Acesta va conține două metode **Push** și **Pop**. Tipul de bază al elementelor va fi de tip **Rational**, iar stiva va fi implementată ca listă liniară simplu înălțuită.

6. Creati o clasă, numită **Coada**. Aceasta va conține metode pentru adăugare și extragere din structură. Tipul de bază al elementelor va fi **Rational**, iar coada va fi implementată ca listă liniară simplu înălțuită.

- Parcurgeti în întreaga un graf, în care fiecare nod reține un obiect Rational. În vederea parcurgerii se va utiliza un obiect de tip Coada.
- Scripti două subprograme care efectuează adunarea, respectiv înmulțirea a două obiecte de tip Matrice (prezentate în curs).
- Prin utilizarea subprogramelor de mai sus și a tipului Matrice_g (prezentat în curs) calculați:

$$I + A + A^2 + A^3 \dots + A^n$$

unde A este o matrice patratică citită de la tastatură cu elemente din \mathbb{R} , iar I este matricea unitate.

- Lucrare în echipă***. Creați o clasă numită Numar_Mare, care permite reținerea unui număr natural cu un număr mare de cifre. De asemenea, se vor scrie funcțiile care efectuează adunarea, scăderea, înmulțirea și împărțirea întreagă cu astfel de numere, precum și comparațiile între ele. Numerele vor fi reținute sub forma de liste liniare simplu înăntăuite.

- Calculați $n!$, unde n este tip Numar_Mare.
- Creați o clasă numit Lista, care va implementa o listă liniară simplu înăntăuită în HEAP.
- Scripti o funcție care permite copierea a două obiecte de tip Lista. Informația, alta decât cea de adresă, va reține un număr real.
- Scripti o funcție care permite eliberarea memoriei reținută de un obiect de tip Lista.
- Realizați sortarea prin inserție prin utilizarea unui obiect de tip Lista.
- Interclasati două liste liniare cu elementele sortate crescător. Listele vor fi reținute prin obiecte de tip Lista.
- Adăugați clasei bkt o metodă numită Run1, care să permită căutarea unei soluții într-un anumit număr de secunde, dat. La expirarea timpului se va tipări conținutul stivei.
- Lucrare în echipă***. Realizați o clasă numită Polinom care permite lucrul cu polinoame de forma $P(x)$. Polinoamele vor fi reținute sub forma unor liste liniare simplu înăntăuite. Clasa va conține și o metodă care permite citirea polinomului dintr-un fișier text. Scripti, de asemenea, funcții de tip prieten care adună, scad., înmulțesc sau împart polinoame.

Capitolul 6

Proiectarea aplicațiilor

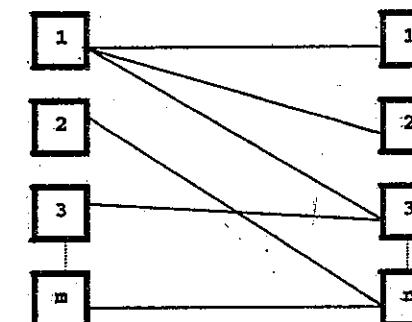
6.1 O problemă din practică

Sunteți analist programator la Oficiul de plasare a forței de muncă. Există un mare număr de persoane care solicită un loc de muncă. Notăm acest număr cu m . Firmele interesate au depus la Oficiu ofertele lor de angajare. În acest sens, s-au centralizat n locuri de muncă. Fiecare dintre cele n locuri de muncă presupune o anumită specializare a celui care îl ocupă. Fiecare dintre cele m persoane care solicită locuri de muncă este calificată într-o sau mai multe meserii. Sarcina dv. este de a repartiza cât mai multe persoane pe locuri de muncă, astfel încât, dacă este posibil, să fie ocupate toate locurile de muncă.

Faptul că sunteți analist programator, vă oferă posibilitatea de a fi cu adevărat util în ceea ce faceti, cu mult mai mult decât o persoană care nu cunoaște decât programare.

6.2 Analiza problemei

Incepeti prin să aseza "fată în fată" cele m persoane și cele n locuri de muncă. Cu alte cuvinte, analizăm cererea comparativ cu oferta. De asemenea, pentru fiecare persoană se trasează căte o linie către fiecare dintre locurile de muncă pentru care este calificată. Obtineti o schemă ca următoarea:



În situația finală de la fiecare persoană trebuie să plece cel mult o linie (pentru că ea poate fi repartizată doar într-un loc de muncă). De asemenea, la fiecare loc de muncă ar trebui să sosească cel mult o

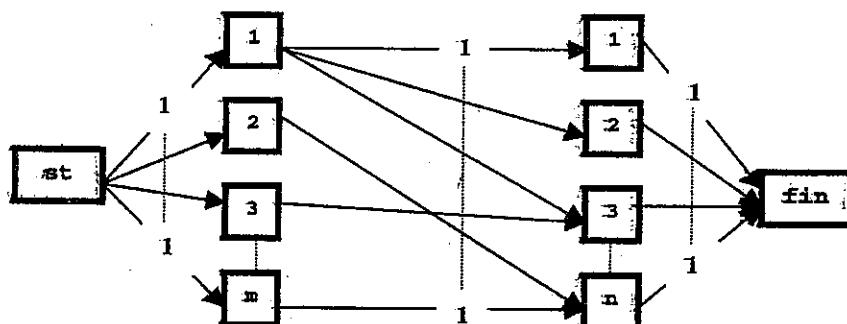
singură linie (pentru că un loc de muncă nu poate fi ocupat de mai multe persoane).

6.3 Conceptia

Cum am putea rezolva problema?

O primă idee este de a o rezolva prin utilizarea tehnicii backtracking. Dupa cum știm, datorită faptului că această tehnică necesită timp exponential, o evităm dacă este posibil. Există probleme pentru care acest lucru nu este posibil, pentru că pentru ele nu se cunosc algoritmi polinomiali. Din fericire, problema actuală nu intră în această categorie.

De la început observăm că schema noastră este, de fapt, un graf (bipartit). De vreme ce este graf, încercăm să inventariem algoritmii pe care i-am studiat pentru grafuri. De componentă conexă nu poate fi vorba, nici de drumuri în grafuri. Atunci? Am studiat un caz particular de grafuri, și anume "rețele de transport". Pentru rețelele de transport am studiat un algoritm pentru aflarea unui flux maxim. Există oare o legătură între acest algoritm și problema noastră? Cu puțină imaginatie, răspunsul este afirmativ. Pentru a transforma în rețea graful nostru, se adaugă un nod de start (**st**) și altul final (**fin**). Nodul **st** este unit printr-un arc de capacitate 1 cu fiecare dintre cele m muncitori. Fiecare dintre cele m locuri de muncă este unit cu nodul **fin** tot printr-un arc de capacitate 1. De asemenea, fiecare arc care unește o persoană cu un loc de muncă va avea tot capacitatea 1.



Observați că fiecare lucrător poate fi asociat unui singur loc de muncă (capacitatea arcului care-l unește cu locul de muncă este 1). Fiecare loc de muncă poate fi ocupat doar de un lucrător (capacitatea arcului care unește locul de muncă cu nodul final este 1). În concluzie, dacă fluxul este maxim, cei m muncitori vor fi repartizați în mod optim.

Programul care rezolvă problema a fost prezentat. Rămâne ca dv. să-l adaptați pentru grafuri de acest tip. În final veți elabora documentația așa cum ați învățat anul trecut în capitolul cu același nume.

- ✓ Aceleași etape, mai puțin cele care privesc elaborarea documentației, deputenți aplica atunci când aveți de rezolvat o problemă la un concurs de informatică.

Probleme propuse

Manualul conține mai multe probleme propuse care pot fi tratate așa cum am învățat aici. Am preferat includerea lor în capitolele ale căror cunoștințe se aplică. Ele pot fi recunoscute sub numele "Lucrare în echipă".

Acum vom propune două probleme de concurs.

1. Sistem de reprezentanți distincți

Se dau m multimi A_1, A_2, \dots, A_m , cu $A_i \subseteq \{1, 2, \dots, p\}$, pentru i de la 1 la m . Se cere, dacă există, un vector cu m componente distințe (a_1, a_2, \dots, a_m) astfel încât:

- a_1, a_2, \dots, a_m sunt distințe;
- $a_1 \in A_1, a_2 \in A_2, \dots, a_m \in A_m$.

- ✓ Ordinea elementelor în vector prezintă importanță!

Exemplu: $m=4, p=6; A_1=\{3, 5\}, A_2=\{3, 5, 6\}, A_3=\{1, 2\}, A_4=\{4\}$.

O soluție este: $(3, 6, 1, 4)$.

Limită: valorile date pentru m și p respectă relația: $m+p \leq 150$.

Datele de intrare se găsesc în fișierul **Date.txt** și sunt organizate astfel:

linia 1: $m \ p$

fiecare dintre următoarele m linii conține date referitoare la o multime sub forma:

$k \ a_1 \ a_2 \ \dots \ a_k$

unde k reprezintă numărul de elemente ale multimii, iar a_1, a_2, \dots, a_k reprezintă elementele multimii.

Exemplu: Pentru mulțimile de mai sus fișierul **Data.txt** este:

4 16
2 3 5
3 3 5 6
2 1 2
1 4

Programul are ca ieșire fișierul **Rez.txt**, unde pe o linie se găsesc, în ordinea cerută, elementele vectorului:

Exemplu: pentru intrarea de mai sus fișierul **Rez.txt** poate conține:

3 6 1 4

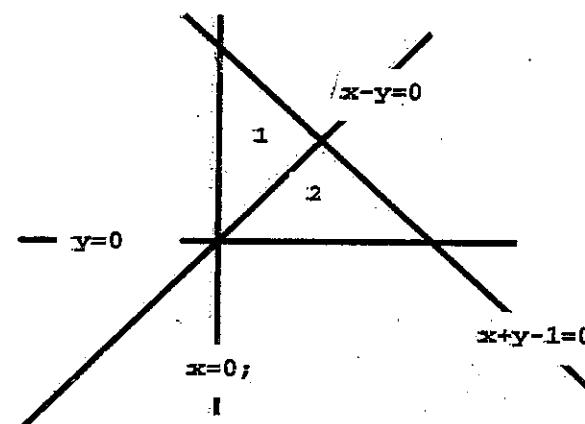
- ✓ Dacă problema nu admite soluție, programul av. va afisa pe monitor: "Nu are solutie".
- ✓ Timp de executare pentru un test: 1 secundă.

Olimpiada de informatică, faza județeană , București 2001

Indicatie: problema de cuplaj, pe care o rezolvăm printr-un algoritm de tip flux maxim într-o rețea de transport.

2. Drepte. Se citesc $N \leq 20$ ecuații de drepte date sub forma $Ax+By+C=0$. Se cere să se determine numărul de poligoane rezultate prin intersecția dreptelor. Se numără numai poligoanele care nu se descompun în altele.

Exemplu: se citesc ecuațiile a 4 drepte: $x=0$; $y=0$; $x+y-1=0$; $x-y=0$; Se obțin 2 poligoane (triunghiuri în acest caz):



Datele de intrare se găsesc în fișierul **drepte.in** în formatul:

- Linia 1 și -numărul dreptelor;
- următoarele n linii conțin fiecare coeficienții unei drepte sub forma A , B , C .

Numărul de poligoane se va tipări în fișierul **rez.out**. Pentru exemplul de mai sus, fișierele trebuie să aibă continutul:

rez.in

4
1 0 0
0 1 0
1 1 -1
1 -1 0

rez.out

2

Timp de lucru: 1 secundă pe test.

Anexa

Modulul "Grafuri"

```
#include <iostream.h>
const float PInfin = 1.e20;
float MInfin = -1.e20;
struct Nod
{
    int nd;
    Nod* adr_urm;
};

void Citire(char Nume_fis[20],int A[50][50],int& n)
{
    ifstream f(Nume_fis,ios::in);
    int i,j;
    f>>n;
    while (f>>i>>j) A[i][j]=1;
    f.close();
}
```

```

void Citire_L(char Nume_fis[20], Nod* L[50], int& n)
{
    Nod* p;
    int i,j;
    fstream f(Nume_fis,ios::in);
    f>>n;
    for (i=1;i<=n;i++) L[i]=0;
    while (f>>i>>j)
    {
        p=new Nod;
        p->adr_urm=L[i];
        p->nd=j;
        L[i]=p;
    }
    f.close();
}

void Citire_cost (char Nume_fis[20], float A[50][50], int& n)
{
    int i,j;
    float c;
    fstream f(Nume_fis,ios::in);
    f>>n;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (i==j) A[i][j]=0;
            else A[i][j]=PInfinite;
    while (f>>i>>j>>c) A[i][j]=c;
    f.close();
}

void Citire_cost1(char Nume_fis[20], float A[50][50], int& n)
{
    int i,j;
    float c;
    fstream f(Nume_fis,ios::in);
    f>>n;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (i==j) A[i][j]=0;
            else A[i][j]=MInfinite;
    while (f>>i>>j>>c) A[i][j]=c;
    f.close();
}

void CitireN(char Nume_fis[20], int A[50][50], int& n)
{
    int i,j;
    fstream f(Nume_fis,ios::in);
    f>>n;
    while (f>>i>>j) A[i][j]=A[j][i]=1;
    f.close();
}

```

```

void Citire_L_N(char Nume_fis[20], Nod* L[50], int& n)
{
    Nod* p;
    int i,j;
    fstream f(Nume_fis,ios::in);
    f>>n;
    while (f>>i>>j)
    {
        p=new Nod; p->adr_urm=L[i]; p->nd=j; L[i]=p;
        p=new Nod; p->adr_urm=L[j]; p->nd=i; L[j]=p;
    }
    f.close();
}

void Citire_cost_N(char Nume_fis[20],float A[50][50],int &n)
{
    int i,j;
    float c;
    fstream f(Nume_fis,ios::in);
    f>>n;
    for (i=1;i<=n;i++) A[i][i]=0;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++) A[i][j]=A[j][i]=PInfinite;
    while(f>>i>>j>>c) A[i][j]=A[j][i]=c;
    f.close();
}

void Citire_cost_N1(char Nume_fis[20],float A[50][50],int &n)
{
    int i,j;
    float c;
    fstream f(Nume_fis,ios::in);
    f>>n;
    for (i=1;i<=n;i++) A[i][i]=0;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++) A[i][j]=A[j][i]=MInfinite;
    while(f>>i>>j>>c) A[i][j]=A[j][i]=c;
    f.close();
}

void Citire_Cap(char Nume_fis[20], int A[50][50][2], int& n, int& st, int& fin)
{
    int i,j,cost;
    fstream f(Nume_fis,ios::in);
    f>>n>>st>>fin;
    while (f>>i>>j>>cost) A[i][j][0]=cost;
    f.close();
}

```

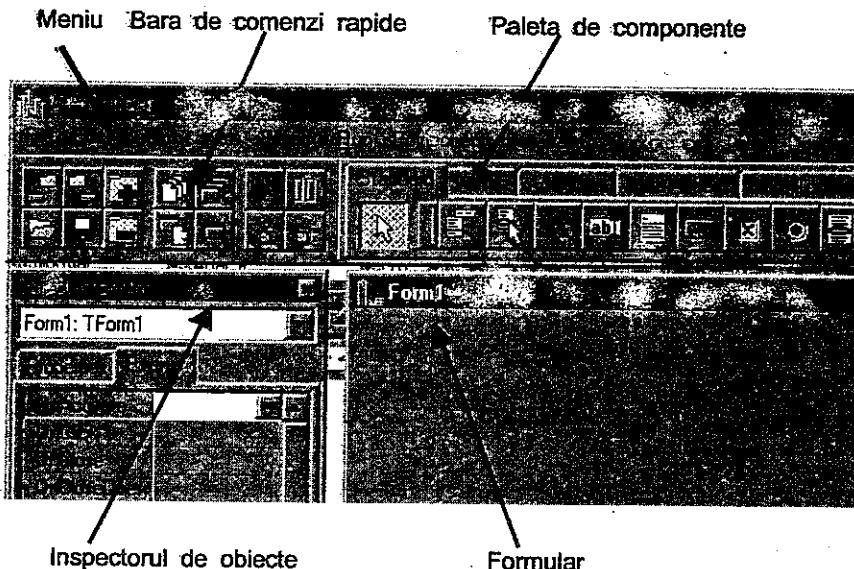
Capitolul 7

Initiere în Programarea Vizuală, principala aplicatie a programării orientate pe obiecte.

7.1 Primii pasi în C++ Builder

7.1.1 Primul program

⇒ Apelati mediul Borland C++ Builder. De fapt, trebuie apelat programul **Bcb.exe**. Iată ce apare:



Înainte de a prezenta tot ce afisează mediul, vom rula deja un prim program, și anume programul implicit. Pentru aceasta comandăm compilarea și executarea. Aceasta se poate face în trei feluri:

1. Executăm click asupra butonului **Run** aflat pe bara de comenzi rapide;
2. Selectăm din meniu **Run**;
3. Tastăm **F9**.

Iată ce apare:

Prin urmare, programul implicit afisează o fereastră. Fereastra este înzestrată cu butoanele care asigură maximizarea, minimizarea sau închiderea ei. De asemenea, ea este înzestrată cu un meniu implicit care se apelează executând click pe icon-ul care reprezintă "Clădirea".



Să vedem acum semnificația elementelor afisate:

- **Meniul** - Servește pentru a da anumite comenzi.
- **Formularul (forma)**. Ați văzut faptul că, în timpul executării, programul afisează o fereastră. În fază de realizare a programului, lucrăm asupra formularului, iar la rulare, formularul devine fereastră.
- **Paleta de componente**. În general, programele realizate sub Windows afisează o fereastră. De obicei, aceasta conține mai multe componente vizuale (butoane, meniuri, etc.). Pentru ca fereastra să le contină, este necesar să adăugăm formularului componentele respective. Colectia de astfel de componente se găseste pe Paleta de componente. Astfel putem adăuga o componentă de tip buton, o componentă de tip meniu, etc.
- **Inspectorul de obiecte**. O componentă este, de fapt, un obiect. Stîm că un obiect are propriile date, care în programarea vizuală se numesc proprietăți. Exemple de proprietăți: culoarea de afisare, font-ul de scriere, coordonatele colțului din stânga sus, înălțimea, lățimea etc. Valorile inițiale ale acestora se stabilesc, de obicei, cu ajutorul inspectorului de obiecte, deși există posibilitatea ca, în anumite cazuri, să fie stabilite cu ajutorul mouse-ului. Modul în care se lucrează cu inspectorul de obiecte îl veți înțelege pe parcurs.

7.1.2 Proprietățile formularului

Cu ajutorul proprietăților stabilim modul în care fereastra apare pe ecran în timpul executării programului. Multe dintre ele le veți regăsi și la alte obiecte (pentru că și fereastra este un obiect). Valorile lor le vom stabili cu ajutorul inspectorului de obiecte.

⇒ **Caption** - reține un sir de caractere care este afișat de fereastra. Respectivul sir constituie, de fapt, titlul ferestrei.

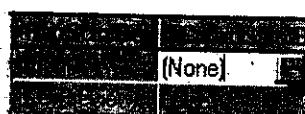
Să presupunem că avem ca fereastra să se numească "Fereastra mea". Stabilirea titlului se face scriind în dreptul proprietății respective, afisată de inspectorul de obiecte, și următorul.



Aici scriem titlul

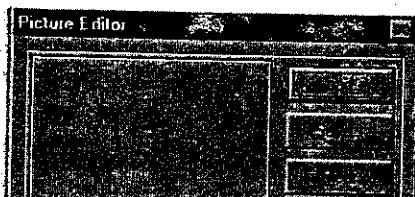
Automat se schimbă titlul ferestrei

⇒ **Icon** - Stabilește icon-ul de identificare a ferestrei. Implicit acesta reprezintă o clădire. În mod practic se încarcă un fișier cu extensia .ico (adică icon-ul).

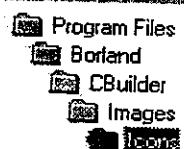


Observăm că în dreptul proprietății respective se găsește un semn distinct (trei puncte).

Aceasta înseamnă că la stabilirea valorilor reținute de proprietatea respectivă se utilizează anumite cutii de dialog. Aici este vorba despre încărcarea unui fișier (cel cu extensia .ico). Executăm click asupra celor trei puncte:



Apare o cutie de dialog, la care selectăm Load.

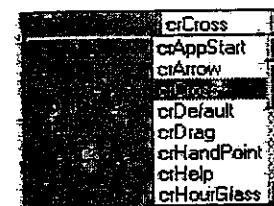


Builder-ul este livrat împreună cu mai multe resurse de acest tip. Alăturat observați folder-ul Icons, de unde vom lua un astfel de fișier.



De acum fereastra va afisa noul icon.

⇒ **Cursor** - stabilește forma cursorului grafic (cel care afișează poziția mouse-ului).



Utilizatorul are de ales forma cursorului dintr-o listă.

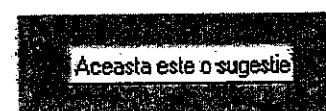
✓ Atunci când cursorul se deplasează deasupra ferestrei, el are forma cerută.

✓ Cursorul implicit este cel CrDefault și este simbolizat de o săgetăță.

Exercitiu. "Încercați" mai multe cursoare din listă!

⇒ **Hint** - Retine un sir de caractere!

⇒ **ShowHint** - Proprietate de tip bool. Poate retine una dintre cele două valori (true, false) selectate dintr-o listă. Dacă reține true și dacă proprietatea Hint reține un sir de caractere, atunci sirul de caractere este afisat pe formular în cazul în care mouse-ul staționează asupra sa.



Hint - "Aceasta este o sugestie".

ShowHint - reține true.

Cursorul grafic al mouse-ului staționează pe fereastră.

✓ După cum știm, în C++ se consideră ca adevărată orice valoare diferită de 0, și falsă o valoare egală cu 0. Pentru a lucra mai ușor cu aceste valori, s-a definit o constantă true, egală cu 1 și o alta false, egală cu 0. De asemenea, s-a definit un tip numit bool, unde o variabilă de acest tip poate reține una dintre aceste două valori.

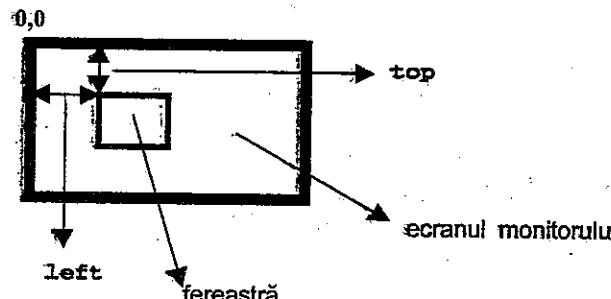
✓ Procedeul este împrumutat din limbajul Pascal!

⇒ **Width, Height** - lățimea și înălțimea ferestrei, în pixeli.

⇒ **ClientWidth, ClientHeight** - lățimea și înălțimea ferestrei, abstractie făcând de bara de titlu și marginile ferestrei, adică suprafața afisată pe care se pot amplasa diverse componente.

✓ În practică, dimensiunile ferestrei se stabilesc cu ajutorul mouse-ului, iar conținutul acestor proprietăți este modificat automat de către sistem.

⇒ **Left, Top** - coordonatele colțului din stânga sus al ferestrei, referitoare la locul de afisare pe monitor a ferestrei. Colțul din stânga sus al ecranului monitorului are coordonatele (0,0).



✓ Atenție la modul în care s-a stabilit locul în care fereastra a fost afișată pe ecran și la stabilirea dimensiunilor ei! Același mecanism există și la afișarea oricărei componente pe fereastră.

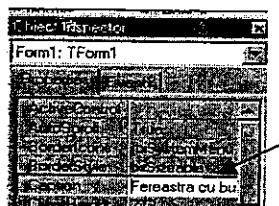
⇒ Color - culoarea ferestrei. Se selectează dintr-o listă.

7.1.3 O fereastră și un buton

□ Vom scrie un program care afișează o fereastră cu un buton. La apăsarea lui, programul afișează un mesaj.

Pasul 1. Apelăm Builder-ul. După cum stim, avem deja creat programul implicit, cel care afișează o fereastră.

Pasul 2. Atașăm ferestrei numele: "Fereastra cu buton".



Scriem sirul respectiv în dreptul proprietății **Caption**.

Pasul 3. Selectăm din paleta de componente un obiect de tip TButton.



Aceasta este componenta selectată. Pentru selecție se execută click asupra sa.

Pasul 4. Asezăm butonul pe formular.

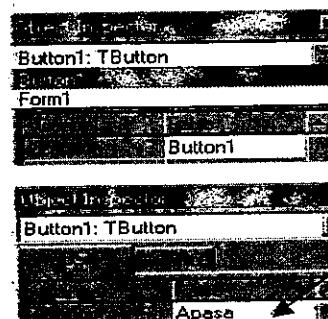


Se execută click asupra formularului. Apoi, pentru ca butonul să ocupe poziția dorită "lă tragem" cu mouse-ul, sau pentru deplasări mici utilizăm **CTRL+săgețiile**.

Pasul 5. Afisăm pe buton un sir dorit de noi, de exemplu: "Apasă!".

Pentru a scrie în dreptul proprietății respective (tot **Caption**) a butonului, este necesar ca inspectorul de obiecte să afișeze proprietățile butonului și nu ale formularului. Există două moduri de a obține aceasta:

1. Dacă butonul nu este selectat (incadrat de un chenar, că mai sus) acesta se selectează (click asupra sa). Selectarea conduce automat la efectul amintit.
2. În partea superioară, inspectorul de obiecte afișează o listă a tuturor obiectelor (în cazul nostru atât butonul (**Button1**) cât și formularul (**Form1**)).



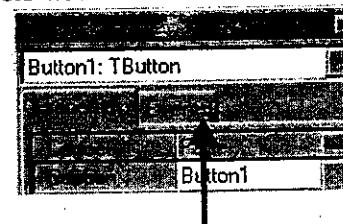
Pentru a obține lista, execută click pe săgeată, apoi operați selecția.

Acesta este locul unde scrie sirul considerat. Automat, butonul va afișa sirul.

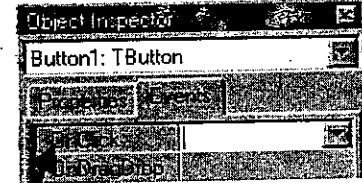
Pasul 7. Trebuie ca, la apăsarea butonului, să apară un mesaj, de exemplu "Merge!". Utilizăm o funcție a limbajului C++ Builder și anume: `ShowMessage(): ShowMessage("Merge!");`

Functia respectivă se apelează dintr-o altă funcție, care răspunde apăsării butonului. Cum o obținem? Inspectorul de obiecte poate afisa, pentru fiecare obiect în parte, atât proprietățile sale cât și evenimentele la care răspunde.

1. Pentru ca inspectorul să afișeze evenimentele la care răspunde butonul, este necesar să executăm click asupra opțiunii respective:

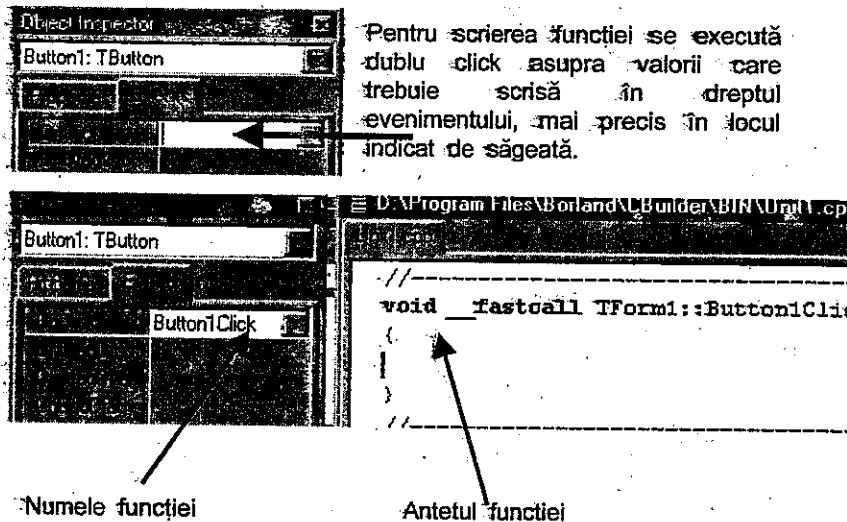


Aici se execută click



Inspectorul va afișa evenimentele la care răspunde butonul

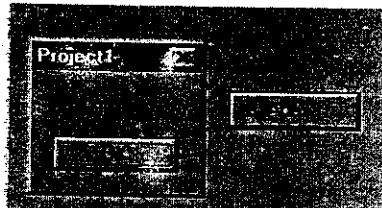
2. Apăsarea unui buton corespunde evenimentului **OnClick**. Acesta este evenimentul care, dacă se produce, trebuie să se ruleze funcția care afișează mesajul cerut. Pe scurt, vom spune că funcția răspunde evenimentului **OnClick**. Prin urmare, ea intră în acțiune la apăsarea butonului.



Automat inspectorul de obiecte va afisa numele implicit al functiei, iar editorul mediului va afisa antetul acesteia. Sarcina dv. este să completati functia. În cazul de față vom adăuga numai **ShowMessage()**:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ShowMessage("Merge!");
}
```

Pasul 8. Testăm programul. Iată ce apare, după executarea unui click asupra butonului:



Programul așteaptă ca să fie apăsat butonul **OK** al cutiei de dialog. După apăsarea acestuia se poate închide fereastra principală.

7.1.4 Ce fișiere alcătuiesc programul sursă C++ Builder

Majoritatea programelor în C++ scrise de noi până acum utilizează un fișier care conține programul și, eventual, acesta conține câteva directive prin care se includ alte module (de exemplu definiția mai multor funcții pe care programul le utilizează).

În C++ Builder un program sursă ocupă mai multe fișiere. El este realizat sub forma unui proiect. Majoritatea fișierelor sunt generate automat, motiv pentru care ne vom limita la o prezentare sumară a lor. De altfel, o prezentare în amănunt a lor ar necesita cunoștințe care depășesc cu mult programa din diceu.

1. Fișierul **project**.

- extensia sa este **.mak**;
- este creat automat de către mediu.
- este afișat de Windows cu icon-ul:
- executarea unui click asupra icon-ului are ca efect lansarea mediului C++ Builder și încărcarea întregului program (și a celorlalte fișiere care-l alcătuiesc);
- Pentru a-l vizualiza se utilizează meniul: **View | MakeFile**.



În linii mari acest fișier conține informații asupra celorlalte fișiere care alcătuiesc proiectul. Iată, de exemplu, cum arată câteva linii ale sale:

```
PROJECT = Project1.exe
OBJFILES = Project1.obj Unit1.obj Unit2.obj
RESFILES = Project1.res
...
```

- ✓ Nu există nici un motiv pentru care programatorul trebuie să modifice acest fișier. Toate operațiile asupra sa sunt efectuate automat de către mediu.

2. Sursa C++ a proiectului.

- Conține sevența C++ care alcătuiește programul. Evident, pentru ca programul să poată fi rulat, se utilizează alte fișiere.
- Are extensia **.cpp** și numele proiectului;
- Pentru a-l examina, apelăm la meniul: **View | Project Source**. Iată-i:

```

#include <vcl\vcl.h>
#pragma hdrstop

USESFORM("Unit1.cpp", Form1);
USERES("Project1.res");
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}

```

- ✓ Dacă pentru un program care rulează sub DOS se utilizează funcția `main()`, pentru unul sub Windows funcția utilizată este `WinMain()`.
- ✓ Observați instrucțiunea compusă care alcătuiește programul! Așa cum stii, în Builder se realizează programe care funcționează sub Windows. Aceasta înseamnă că apelul este specific acestui sistem de operare. În linii mari, se initializează aplicația, se creează formularul de bază, apoi aplicația este lansată în executare. Întrucât Builder-ul utilizează mecanismele programării orientate pe obiecte, instrucțiunile au formatul specific acestora.
- ✓ Programatorul nu lucrează cu acest fisier. El este gestionat automat de mediu.

3. Fișierele care conțin descrierea formularului.

Formularul (fereastra afisată de program) este descrisă în mai multe fișiere.

3.1 Fișierul antet.

- are extensia `.h`;
- Conține clasa care definește formularul. Mai jos este prezentat cum arată un astfel de fișier, în cazul în care formularul conține un buton:

```

#ifndef Unit1H
#define Unit1H

#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>

```

```

class TForm1 : public TForm
{
public:
    _published: // IDE-managed Components
        TButton *Button1;
        void __fastcall Button1Click(TObject *Sender);
private:
    // User declarations
public:
    __fastcall TForm1(TComponent* Owner);
};

extern TForm1 *Form1;
#endif

```

Clasa care definește formularul:

Butonul

- ✓ și acest fisier este creat automat de către mediu. Nu are sens ca programatorul să lucreze cu el.
- ✓ De fapt, obiectele se rețin în HEAP, iar programul le adresează prin intermediul pointerilor.

3.2 Fișierul care conține funcțiile utilizator:

- Are extensia `.cpp`;
- Este creat automat de către mediu, dar sarcina scrierii funcțiilor care-l alcătuiesc este a programatorului.
- Așa cum am văzut, antetul funcțiilor este scris, tot automat, dar de către mediu, iar programatorul n-a decât sarcina scrierii sevențelor pe care le dorește!
- pentru a-l examina, apelăm la meniu: `View | Units`. Iată-l, în cazul programului anterior, care conține un singur buton:

```

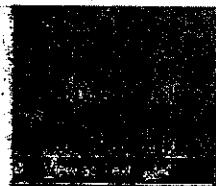
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
#pragma resource "*.dfm"
TForm1 *Form1;
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    ShowMessage("Merge!");
}

```

- ✓ Programatorul n-a scris decât `ShowMessage...`
- ✓ Vă puteți întreba: unde se rețin proprietățile formularului, unde se rețin proprietățile butonului? Pentru a le reține pe acestea există un fișier, cu extensia `.dfm` (se referă la formă).

3.3 Fișierul care reține aspectul formularului.

- extensia sa este .dfm;
- este creat automat de către mediu;
- datele sunt reținute într-un format specific, dar pot fi convertite în format text, accesibil utilizatorului.
- pentru a-l vizualiza în format text, așezăm cursorul mouse-ului deasupra formularului.



Apăsăm butonul drept al mouse-ului. Apare un astă numit meniu flotant (vezi figura alăturată). Se apelează opțiunea **View as Text**.

Nată ce apare pentru programul anterior:

```
object Form1: TForm1
  Left = 200
  Top = 109
  Width = 435
  Height = 299
  Caption = 'Form1'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 152
    Top = 72
    Width = 75
    Height = 25
    Caption = 'Apasa'
    TabOrder = 0
    OnClick = Button1Click
  end
end
```

- ✓ Există posibilitatea, după cum vom învăța, ca programul să contină mai multe formule. În acest caz există mai multe triplete de astfel de fișiere.

Regăsim toate proprietățile formularului și ale butonului inclus în formular.

Acest text a fost creat automat!

7.1.5 Salvarea programului

Spre deosebire de Borland C++, unde programul sursă ocupă de cele mai multe ori un singur fișier, sursa **Builder** este alcătuță, așa cum am văzut, din mai multe fișiere. Aceasta are o consecință directă asupra mecanismului de salvare a programului. Vom învăța să salvăm un program. Pentru aceasta putem utiliza programul anterior, cel cu un buton, sau programul implicit. Presupunem că programul care trebuie salvat este încărcat de mediul **Builder** și trebuie salvat pe **C:** în folder-ul **Programul_meu**.

Pasul 1. Creăm folder-ul **Programul_meu** pe **C:**.

- a) Minimizăm fereastra **Builder**;
- b) Apelăm, de exemplu, aplicația **My Computer**. Cu ajutorul ei facem ca folder-ul activ să fie rădăcina.
- c) Apăsăm butonul drept al mouse-ului. Apare un meniu flotant și alegem **New | Folder**, după care completăm numele ales, **Programul_meu**.

Pasul 2. Maximizăm fereastra mediului **Builder** și apelăm opțiunea **File** a meniului. Selectăm **Save Project As**. Salvăm, pe rând, fișierele care alcătuiesc programul, având grijă ca numele proiectului să fie diferit de numele comun (abstracție făcând de extensie) al fișierelor care rețin date despre formular.

- ✓ Este recomandabil ca atunci când lucrăm la un program să-l salvăm din când în când. Pentru aceasta se apelează opțiunea **File | Save All** a meniului.

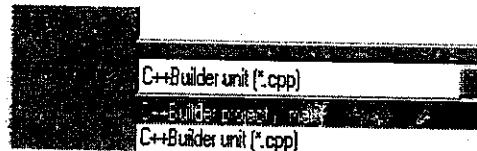
7.1.6 Încărcarea programului

Problemă. Dorim să încărcăm sursa existentă într-un anumit folder. Cum procedăm? Există mai multe metode.

Metoda 1. De exemplu, cu **MyComputer** vizualizăm folder-ul în care se găseste aplicația. Executăm dublu click pe fișierul proiect (extensia **.mak**). Astfel, se apelează **Bcb.exe** și acesta va încărca automat toate fișierele care tin de sursă.

Metoda 2. Apelăm **Bcb.exe**, de exemplu, din meniul **Start**. Apoi procedăm într-unul din modurile următoare:

- Apelăm **File | Open**. Cu ajutorul cutiei de dialog care apare se selectează fișierul proiect (extensia **.mak**).



- ✓ În cutia de dialog se va selecta afişarea fişierelor proiect.
- ✓ Lansarea cutiei de dialog pentru deschidere se poate face și prin click pe butonul următor, aflat pe bara de comenzi rapide:
- ✓ Putem apela **File/ReOpen**, pentru că **Builder-ul** ține evidența ultimelor proiecte deschise.

Probleme propuse

1. Scrieți un program care afișează o fereastră. Titlul ferestrei va fi: **Prima aplicatie**.
2. Salvați programul scris de dv. într-un folder numit **Primul_Program**.
3. Adăugați ferestrei create în programul anterior un icon furnizat o dată cu mediul **Builder**.
4. Modificați programul anterior de așa natură astfel încât fereastra afișată să aibă culoarea galbenă.
5. Modificați programul anterior astfel încât fereastra să fie pătrată.
6. Modificați programul astfel încât fereastra să fie afișată în colțul din stânga sus al ecranului.
7. Adăugați ferestrei un buton pe care scrie **"Titlul"**. La apăsarea sa se va afișa titlul ferestrei.
8. Creati un folder numit **Noul_program** și salvați în el programul astfel obținut.

7.2 Intrări / ieșiri elegante

7.2.1 Componenta de tip TLabel

Ați văzut că de înstinct se afișează diverse mesaje în Borland C++. În **Builder** afișarea textelor este deosebit de elegantă. Pentru început să afișare vom folosi componenta **TLabel**.



Simbolul alăturat are semnificația de componentă **TLabel**.



Amplasarea sa pe formular se face ca în cazul butonului. Iată cum arată componenta amplasată pe formular.

În continuare studiem câteva proprietăți ale componentelor de tip **TLabel**.

⇒ **Caption** - este de tip **AnsiString** și reține sirul afișat de componentă. Valoarea implicită este cea afișată - **Label1**.

Să presupunem că dorim să afișăm sirul **"Acesta este un text afișat elegant"**. Modificăm valoarea reținută la **Caption** de către inspectorul de obiecte, scriind sirul de mai sus:



Iată cum va arăta mesajul pe formular:

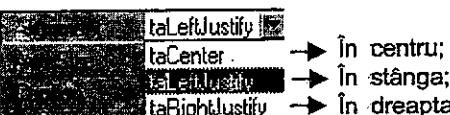
⇒ **AutoSize** - este proprietate de tip **bool**. Dacă reține **true**, componenta va ocupa pe ecran exact atâta spațiu cât are nevoie pentru a afișa întregul sir, iar dacă reține **false**, ea va ocupa spațiul stabilit de către programator.

✓ În practică este indicat ca proprietatea să rețină **false**.



Componenta alăturată reține la **AutoSize false**.

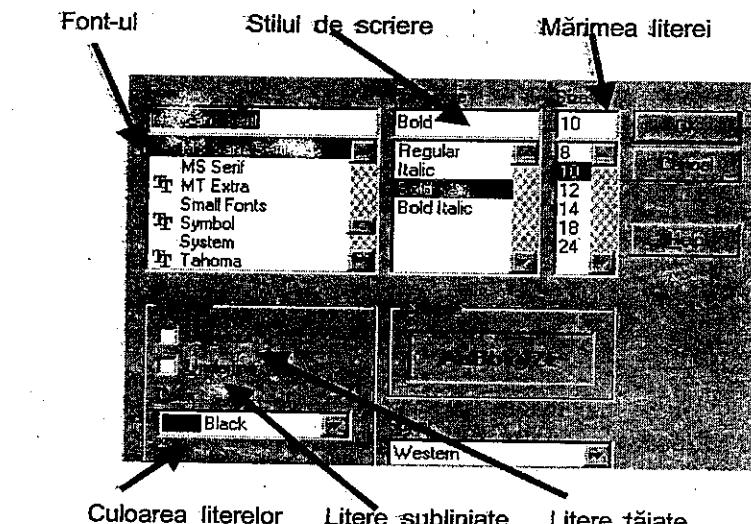
⇒ **Alignment** - alinierea sirului afișat de componentă. Poate lua valorile:



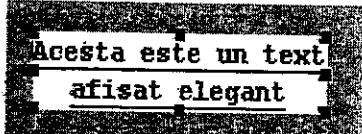
- ✓ În imaginea anterioară proprietatea **Alignment** retine `taLeftJustify`.
- ⇒ **WordWrap** – proprietate de tip `bool`. Dacă reține `true` este permis ca sirul să fie afisat pe mai multe rânduri (în cazul în care sirul nu poate fi afisat pe un rând). Exemplu:



- ⇒ **Font** – proprietate prin care se stabilesc fonturile. Executarea unui click asupra celor trei puncte afisate de inspectorul de obiecte conduce la apariția unei cutii de dialog cu ajutorul căreia se stabilește modul în care arată textul:

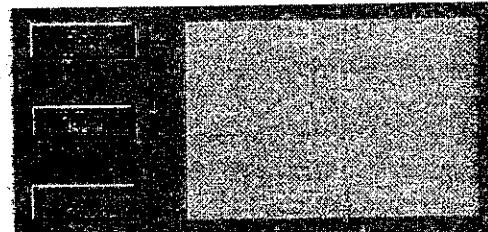


- **Color** – culoarea de fond



Până în prezent, valorile diferitelor proprietăți au fost cele implicate, sau au fost stabilite cu ajutorul inspectorului de obiecte. În realitate, acestea pot fi modificate în timpul executării la fel ca variabilele. Aplicația următoare constituie un exemplu în acest sens.

- ⇒ Programul următor modifică culoarea de afisare a unui obiect de tip `TLabel` cu ajutorul a trei butoane.



Pentru a obține acest efect am procedat astfel:

- Am amplasat pe formular 3 butoane și un obiect de tip `TLabel`.
- Proprietatea `Caption` a fiecărui buton reține un sir care stabilește culoarea;
- Culoarea inițială a componentei `Label1` (nume implicit) este roșie. Vezi proprietatea `Color`. Mai mult, ca să selectăm butonul roșu (adică să fie încadrat de chenar), arătând astfel că datorită lui obiectul afisează culoarea respectivă, proprietatea `ActiveControl` a formularului va reține `Button1` (se selecteză dintr-o listă).
- Pentru fiecare buton în parte, scriem o funcție care răspunde evenimentului `OnClick`, cu rolul de a atribui obiectului `Label1` culoarea corespunzătoare. Iată, de exemplu, funcția pentru butonul pe care scrie Galben.

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Label1->Color=clYellow; }
```

7.2.2 Tipul (clasa) `AnsiString`

Spre deosebire de Pascal, în C++ lucrul cu siruri de caractere este destul de greoi. Din acest motiv, în Builder există un tip nou, numit `AnsiString`. O variabilă de acest tip poate reține un sir de caractere. De asemenea, există o mulțime de funcții cu ajutorul căroră lucru cu siruri de caractere devine deosebit de ușor.

- ✓ Pentru cei care au studiat OOP, putem spune că **AnsiString** este, de fapt o clasă, cu o mulțime de metode, de tip constructor sau nu, funcții prietenice, care, uneori, sunt rezultate prin suprăîncărcarea operatorilor (vezi concatenarea sirurilor, unde se utilizează operatorul +).
- ⇒ O variabilă de tip **AnsiString** se declară așa cum se vede mai jos. De asemenea, se poate observa cum unei astfel de variabile i se atribuie un sir de caractere, după care poate fi afișată cu **ShowMessage()**.

```
AnsiString S;
S="Un sir";
ShowMessage(S);
```

- ⇒ O variabilă de tip **AnsiString** poate fi inițializată de la declaratie:

```
AnsiString S="Un sir";
ShowMessage(S);
```

- ⇒ O variabilă de tip **AnsiString** poate fi inițializată de la declaratie și așa:

```
AnsiString S("Un sir");
ShowMessage(S);
```

- ⇒ Două astfel de variabile pot fi concatenate cu ajutorul operatorului +. În acest fel se obține un sir de caractere format din primul sir (conținutul lui **S**) la care se adaugă al doilea sir (conținutul lui **T**).

```
AnsiString S="Un sir ",T="alt sir";
S=S+T;
ShowMessage(S);
```

- ✓ Operația de concatenare nu este comutativă!

- ⇒ Funcția **IntToStr** are rolul de a converti o valoare întreagă în sir de caractere. În secvența următoare, sirul obținut după conversie este atribuit unei variabile de tip **AnsiString**, după care este afișat:

```
AnsiString IntToStr(int);
AnsiString S;
int n=17;
S=IntToStr(n);
ShowMessage(S);
```

- ⇒ Funcția **FloatToStr** are rolul de a converti o valoare reală în sir de caractere. În secvența următoare, sirul obținut după conversie este atribuit unei variabile de tip **AnsiString**, după care este afișat:

```
AnsiString FloatToStr(float);
```

```
AnsiString S;
float x=-17.65;
S=FloatToStr(x);
ShowMessage(S);
```

- ⇒ Funcția **StrToInt** are rolul de a converti un sir de caractere către o valoare întreagă. În secvența următoare, se convertesc către valori întregi două siruri, se face suma lor, iar rezultatul este convertit către sir, pentru a fi afișat:

```
int StrToInt(AnsiString);
AnsiString S="-17",T="22";
int x=StrToInt(S)+StrToInt(T);
ShowMessage(IntToStr(x));
```

- ⇒ Funcția **StrToFloat** are rolul de a converti un sir de caractere către o valoare reală. În secvența următoare, se convertesc către valori reale două siruri, se face suma lor, iar rezultatul este convertit către sir, pentru a fi afișat:

```
float StrToFloat(AnsiString);
AnsiString S="-17.5",T="22";
float x=StrToFloat(S)+StrToFloat(T);
ShowMessage(FloatToStr(x));
```

- ✓ În cazul în care, pentru ultimele două funcții conversia nu reușește (datele sunt eronate) programul dă eroare de execuție, ca în exemplul următor, unde sirul **S** nu poate fi convertit către întreg:

```
AnsiString S="-A17",T="22";
int x=StrToInt(S)+StrToInt(T);
```

- AnsiString** este, așa cum am precizat, o clasă. Prin urmare, ea este înzestrată cu metode proprii.

- ⇒ Metoda **Length** -returnează lungimea sirului de caractere.

```
int Length();
AnsiString S("1a2na");
ShowMessage(IntToStr(S.Length())); // Se afiseaza 5
```

- ⇒ Metoda **LowerCase** -returnează sirul scris cu litere mici:

```
AnsiString LowerCase();
AnsiString S("1A2Na");
ShowMessage(S.LowerCase()); // se afiseaza 1a2na
```

- ⇒ Metoda **UpperCase** -returnează sirul scris cu litere mari:

```
AnsiStringUpperCase()
```

```
AnsiString S("1a2Na");
```

```
ShowMessage(S.UpperCase()); // se afiseaza 1A2NA
```

⇒ Metoda **SubString** are rolul de a returna subșirul care începe cu octetul de indice **index** și are lungimea **lung**:

```
AnsiString SubString(int index, int lung);
```

```
AnsiString S="12345";
```

```
ShowMessage(S.SubString(2,3)); // afiseaza 234
```

⇒ Metoda **Insert** are rolul de a insera sirul **sir** înaintea octetului de indice **index**:

```
void Insert (Ansistring& sir,int Index);
```

```
AnsiString S="12345",T="abc";
```

```
S.Insert(T,2);
```

```
ShowMessage(S); // afiseaza 1abc2345
```

⇒ Metoda **Delete** are rolul de a suprima dintr-un sir subșirul care începe cu octetul de indice **index** și are lungimea **lung**:

```
void Delete (int index,int lung);
```

```
AnsiString S="12345";
```

```
S.Delete(2,3);
```

```
ShowMessage(S); // se afiseaza 15
```

⇒ Metoda **AnsiCompare** compară sirul curent (pentru care este apelată) cu sirul transmis ca parametru. Compararea este lexicografică și întrucât a fost studiată, nu insistăm asupra ei.

```
int AnsiCompare (Ansistring& X);
```

Rezultatul este:

- 0, în caz de egalitate;
- -1 dacă sirul curent este mai mic;
- +1 dacă sirul curent este mai mare.

```
AnsiString S="A",T="B";
```

```
int n=S.AnsiCompare(T);
```

```
ShowMessage(IntToStr(n)); // se afiseaza -1
```

Uneori este necesar ca o valoare memorată în virgulă mobilă să fie convertită cu format către un sir. Pentru aceasta, este utilă metoda următoare:

⇒ **FloatToStrF**. Forma pe care o vom folosi pentru această metodă este:

```
AnsiString FloatToStr(variabila reala, AnsiString::effFixed,  
int v, int zecimale);
```

Variabila **v** trebuie să ia valorile:

- 7 dacă se converteste către sir o variabilă de tip **float**;
- 15 dacă se converteste către sir o variabilă de tip **double**;
- 18 dacă se converteste către sir o variabilă de tip **long double**;
- ✓ Dacă rezultatul nu începe se fac rotunjiri.

```
AnsiString S;  
float x=-123.5678;  
S=FloatToStrF(x,Ansistring::effFixed,7,2);  
ShowMessage(S); // se afiseaza -123.57
```

7.2.3 Exceptii

Uneori programele dă erori de executare în cazuri cum ar fi: tentativa de împărțire la 0, tentativa de conversie a unui sir către o valoare numerică, dar sirul nu poate fi convertit, tentativa de deschidere a unui fisier inexistent. În toate aceste cazuri apare o aşa numită "exceptie" și în absența unor măsuri de precauție, se generează eroare de execuție, aşa cum am arătat în capitolul unu.

Exemplu: încerc să convertesc un sir către un întreg, dar sirul nu poate fi convertit:

```
AnsiString S="-1h23";  
float x=StrToInt(S);
```

La tentativa de conversie de mai sus apare cutia de dialog alăturată.



Pentru a evita o astfel de situație neplăcută, și, de cele mai multe ori, de neînteleas de către utilizator, se procedează astfel:

A. Din meniu: Options | Environment



Se anulează opțiunea **Break on exception**

B. Se folosește instrucțiunea **try** așa cum se vede mai jos:

```
AnsiString S="-1h23";  
try  
{ float x=StrToInt(S); }  
catch(...)  
{ ShowMessage("Conversie imposibila"); }
```

- ✓ În cazul în care conversia nu se poate efectua se execută instrucțiunea compusă aflată după `catch`.
- ✓ În această carte vor fi date și alte exemple de utilizare a instrucțiunii `try`.

7.2.4 Componenta de tip TEdit

Componentele de tip `TEdit` sunt folosite pentru a afișa siruri de caractere, dar și pentru a permite utilizatorului introducerea lor. Un astfel de obiect permite să se scrie / citească texte pe un singur rând. Iată cum selectăm un obiect de tip `TEdit`:



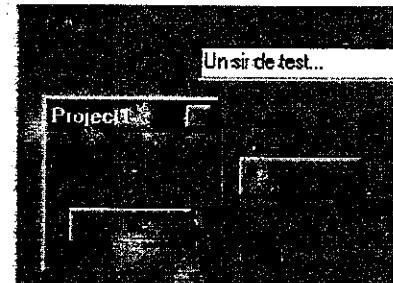
Pentru a-l utiliza, selectăm simbolul alăturat

Principalele proprietăți ale acestor obiecte sunt:

- ⇒ **Text** - de tip `AnsiString`. Reține sirul de caractere care este afișat sau care se citește. Mai precis, dacă initial această proprietate conține un sir de caractere - introdus cu ajutorul inspectorului de obiecte - atunci acest text este afișat în timpul executării. Dacă utilizatorul introduce un sir de caractere, programul va găsi aici sirul introdus.
- ⇒ **MaxLength** - stabilește numărul de caractere pe care le poate conține sirul introdus de utilizator. De exemplu, dacă conține 2, atunci se pot introduce doar siruri de două caractere. În situația în care nu dorim ca acest sir să aibă limitată lungimea, valoarea memorată trebuie să fie 0. De altfel, aceasta este valoarea implicită - adică cea pusă automat de mediu.
- ⇒ **CharCase** poate conține una dintre următoarele 3 valori, care au semnificația:
 - `ecUpperCase` - textul introdus va fi automat convertit în litere mici;
 - `ecLowerCase` - textul introdus va fi automat convertit în litere mari;
 - `ecNormal` - nu se fac conversii.
- ⇒ **ReadOnly** - este de tip `bool`. Dacă reține `true`, atunci utilizatorul nu poate introduce date.
- ⇒ **PasswordChar** - se folosește pentru introducerea parolelor. Dacă este introdus un caracter diferit de cel nul (#0), atunci, indiferent ce scrie utilizatorul, la introducerea textului, apare acel caracter. De exemplu, dacă conține * și se introduce sirul "mama", apare sirul "****". În schimb, din program se poate citi sirul real.

- ✓ Înainte de a prezenta principalele tipuri de aplicații, atrag atenția asupra faptului că în C++ toate citirile / scrierile de la tastatură se realizează sub forma sirurilor de caractere, iar conversia acestora către formatele dorite este făcută de programator.

- **Aplicatie 1.** Scrieți un program care citește un sir de caractere prin utilizarea unei componente de tip `TEdit`. Sirul citit se afișează prin:



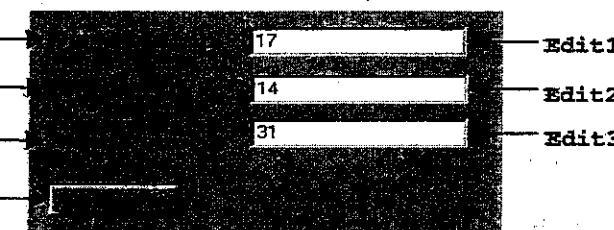
Așezăm pe formular o componentă de tip `TEdit` și un buton. Butonul afișează sirul "Ce am citit". Valoarea implicită a proprietății `Text` a lui `Edit1` este "`Edit1`". Întrucât aceasta nu convine, atribuim proprietății sirul vid. La apăsarea butonului va răspunde funcția:

`ShowMessage()`. Citirea și afișarea se face la apăsarea unui buton.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ShowMessage(Edit1->Text); }
```

- **Aplicatie 2.** Scrieți un program care citește două numere naturale și afișează suma lor. Pentru afișare se va utiliza o componentă de tip `TEdit`.

În figura următoare se prezintă modul în care arată fereastra în timpul executării. Pentru a obține aceasta se asează pe formular mai multe componente (vom prezenta numele lor implicit).

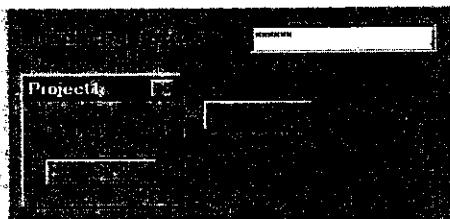


La apăsarea butonului răspunde funcția:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int Suma=StrToInt(Edit1->Text)+StrToInt(Edit2->Text);
  Edit3->Text=IntToStr(Suma);
}
```

Se citesc siruri care trebuie convertite către întregi. Pentru aceasta utilizăm funcția `StrToInt`. Valorile convertite se adună și rezultatul se depune în variabila `Suma`. Pentru a fi afisată, valoarea rezultată se convertește către un sir care este atribuit proprietății `Text` a componentei `Edit3`.

- Aplicația 3. Programul dv. va căuta o parolă. Dacă parola nu este corectă, se va afisa "Parola incorrectă", altfel se va afisa "Ok". Presupunem că parola corectă este "Builder".

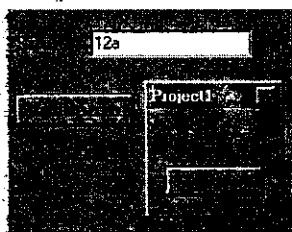


Proprietatea `PasswordChar` a componentei `Edit1` reține `'*'.`

În rest, la apăsarea butonului răspunde funcția:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    AnsiString Parola="Builder";
    if (!Parola.AnsiCompare(Edit1->Text))
        ShowMessage("Ok");
    else ShowMessage("Parola incorrectă");
}
```

- Aplicația 4. Se citește un sir de caractere. Să se decidă dacă sirul reprezintă un număr întreg. În caz afirmativ se scrie "Ok", altfel se afisează "nu este intreg".



Pentru validare folosim funcția `StrToInt` și instrucțiunea `try`, așa cum suntem obișnuiți.

Mai jos este prezentată funcția care se rulează la apăsarea butonului.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    try
    {
        int n=StrToInt(Edit1->Text);
        ShowMessage("Ok");
    }
    catch (...)
    {
        ShowMessage("nu este intreg");
    }
}
```

7.2.5 Focus-ul

Este o aplicație al cărei formular conține mai multe componente. La rulare, la un moment dat, toată anumită componentă este selectată. Pe scurt, vom spune că acea componentă care este selectată are **focus-ul**. Exemple:

- a) Butonul 2 are focus-ul. Dacă se tastează `Enter`, se apasă automat butonul 2.



- b) Prima componentă are focus-ul. Înseamnă că ea conține cursorul și dacă utilizatorul folosește tastatura, ea este cea care recepționează caracterele tastate.

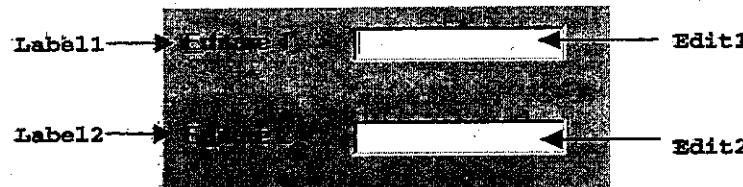


Cum se poate face ca un anume obiect să contină focus-ul?

- În primul rând, această operatie poate fi realizată de utilizator, care execută un click asupra obiectului.
- Majoritatea obiectelor au o metodă prin care li se poate atribui focus-ul, numită `SetFocus()`. Exemple:

 - a) `Button1->SetFocus();` În urma executării, butonul apare încadrat de un chenar.
 - b) `Edit1->SetFocus();` În urma executării, componenta de editare `Edit1` este selectată.

- Focus-ul poate fi dirijat prin intermediul obiectelor de editare statice. Priviți figura următoare:



Puteți observa două obiecte de editare de tip `TLabel`, `Label1` și `Label2`, precum și două obiecte de editare de tip `TEdit`, `Edit1` și `Edit2`. Componenta `Edit1` are focus-ul.

1. Fiecare obiect de editare de tip `TLabel` are o literă subliniată. Pentru obținerea acestui efect se utilizează caracterul `&`, asezat înaintea literelor respective - este vorba de sirul reținut de `Caption`. De exemplu, proprietatea `Caption` a obiectului `Edit2` reține sirul '`&Editare 2`'.

- 2. Fiecare obiect de editare de tip TLabel are o proprietate numită FocusControl, afisată de inspectorul de obiecte. Pentru Label1, proprietatea reține Edit1, pentru Label2 proprietatea reține Edit2.
- 3. În aceste condiții, dacă se tastează Alt+K, focus-ul este primit de Edit1, dacă se tastează Alt+D, focus-ul este primit de Edit2.
- ✓ Acesta este mecanismul de redirecțiere a focus-ului prin intermediu obiectelor de editare statice.

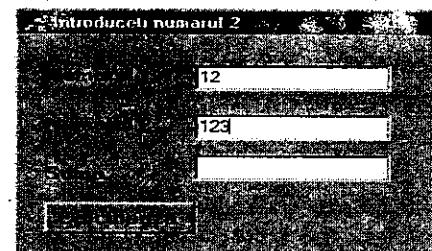
Înainte de a da un exemplu de utilizare a celor învățate, vom prezenta două evenimente care au strictă legătură cu focus-ul.

- OnEnter -asupra unui obiect apare un eveniment OnEnter atunci când acesta primește focus-ul.
- OnExit -asupra unui obiect apare un eveniment OnExit atunci când acesta pierde focus-ul.
- Aplicatie. Să se calculeze suma a două numere naturale. Utilizatorul va fi ghidat permanent asupra acțiunilor următoare, prin mesajul afișat pe formular. De asemenea, datele de intrare trebuie să fie validate.

Rezolvare:

- Se amplasează pe formular 3 componente de tip TLabel, pentru care proprietatea Caption reține respectiv: Numarul 1, Numarul 2, Suma.
- Efectul de subliniere a fost obținut prin fastarea caracterului '&'. De exemplu, proprietatea Caption a componentei Label1 conține Numarul &1.
- Proprietatea FocusControl a lui Label1 conține Edit1 și proprietatea FocusControl a lui Label2 conține Edit2.
- Se amplasează pe formular 3 componente de tip TEdit pentru introducerea celor două numere și pentru afișarea sumei.
- Pentru evenimentul OnEnter al primei componente răspunde funcția următoare, care are rolul de a afișa pe fereastră mesajul prin care utilizatorul este invitat să introducă numărul 1.

```
void __fastcall TForm1::Edit1Enter(TObject *Sender)
{ Form1->Caption="Introduceti numarul 1";
}
```



- Pentru evenimentul OnEnter al celei de-a doua componente răspunde funcția următoare, care are rolul de a afișa pe fereastră mesajul prin care utilizatorul este invitat să introducă numărul 2.

```
void __fastcall TForm1::Edit2Enter(TObject *Sender)
{ Form1->Caption="Introduceti numarul 2";
}
```

- Pentru evenimentul OnExit al primei componente răspunde funcția următoare, care are rolul de a valida valoarea introdusă, iar în caz că aceasta va fi introdusă greșit, de a redirectiona focus-ul către componentă.

```
void __fastcall TForm1::Edit1Exit(TObject *Sender)
{ try
{ StrToInt(Edit1->Text);
}
catch(...)
{ ShowMessage("Numar incorrect");
Edit1->SetFocus();
}
}
```

- Pentru evenimentul OnExit al celei de-a doua componente răspunde o funcție cu același rol ca precedența.

```
void __fastcall TForm1::Edit2Exit(TObject *Sender)
{ try
{ StrToInt(Edit2->Text);
}
catch(...)
{ ShowMessage("Numar incorrect");
Edit2->SetFocus();
}
}
```

- c) Pe formular am așezat și un buton.
- În apăsarea sa, răspunde funcția următoare, cu rolul de a efectua suma cerută și de a o afișa.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int s=StrToInt(Edit1->Text)+StrToInt(Edit2->Text);
Edit3->Text=IntToStr(s);
}
```

- Pentru evenimentul OnEnter, răspunde funcția următoare, care anunță utilizatorul că s-a calculat suma:

```
void __fastcall TForm1::Button1Enter(TObject *Sender)
{ Form1->Caption="Afisez suma";
}
```

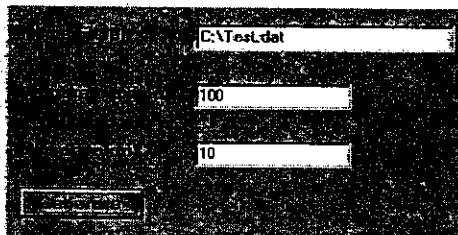
7.2.6 Din nou despre fisiere text

În Borland C++ Builder, putem lucra cu fisiere text, aşa cum am fost obişnuiţi în Borland C++ 3.0.

- ⇒ Pentru a le utiliza trebuie să includem în fișierului care conține funcțiile utilizatorului fișierul antet `#include<iostream.h>`.
- ⇒ Numele fișierului nu este de tip `AnsiString`, ci este sir de caractere, ca în Borland C++ 3.0. Dar aceasta nu ridică nici o problemă întrucât clasa `AnsiString` are o metodă care returnează un pointer (`char*`) către sirul de caractere:

```
char* c_str();
```

- ✓ Un exemplu de utilizare a acestei metode va fi dat în aplicația următoare.
- Aplicația 1. Să se creeze un fișier text cu numere aleatoare. Utilizatorul va introduce numele fișierului și valoarea maximă a numerelor care pot fi generate și numărul acestora.

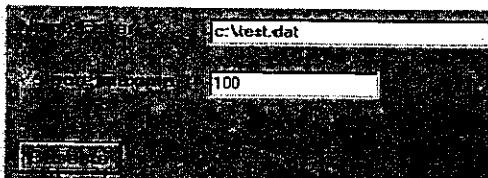


Formularul este cel elăturat. Există o singură funcție, care se apelează la apăsarea butonului.

- ✓ Datele de intrare se presupun corecte!

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int i,Max=StrToInt(Edit2->Text),valoare;
  fstream f( Edit1->Text.c_str(),ios::out);
  for (i=1;i<=StrToInt(Edit3->Text);i++)
  { valoare=1+rand() % Max;
    f<<valoare<<" ";
  }
  f.close();
}
```

- Pentru fișierul text creat în programul anterior se cere să se afișeze valoarea maximă memorată.



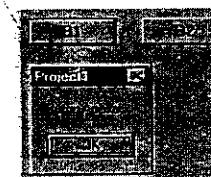
Avem o singură funcție, cea de mai jos, care răspunde la apăsarea butonului:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ fstream f(Edit1->Text.c_str(),ios::in);
  int i,Max,valoare;
  if (!f)
  { if (>Max)
    while (f>>valoare)
      if (valoare>Max) Max=valoare;
    f.close();
    Edit2->Text=IntToStr(Max);
  }
  else
  { ShowMessage("Fisierul nu exista");
    Edit1->SetFocus();
  }
}
```

7.2.7 O singură funcție, mai multe componente

În acest paragraf răspundem la întrebarea: *cum facem ca o singură funcție să răspundă mai multor componente?*

- Formularul aplicatiei conține două butoane. Dorim ca la apăsarea oricărui buton să răspundă o singură funcție care afișează prin `ShowMessage` sirul "Buton Apasat".



În fapt, la evenimentul `OnClick` trimis de unul din cele două butoane trebuie să răspundă o singură funcție.

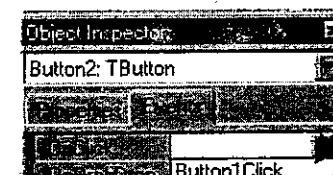
Rezolvăm problema în doi pași:

Pasul 1. Atășăm, aşa cum ştim, funcția următoare primului buton:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ShowMessage("Buton Apasat"); }
```

Pasul 2. Atășăm aceeași funcție celui de-al doilea buton.

- selectăm pe formular butonul (aceasta are ca efect vizualizarea cu inspectorul de obiecte a proprietăților și evenimentelor la care acesta poate răspunde).
- pentru evenimentul `onClick` al celui de-al doilea buton selectăm din listă aceeași funcție (există această posibilitate).



De aici se selectează lista funcțiilor și executăm click asupra numelui de funcție.

7.2.8 Parametrul **Sender** și nu numai...

Scopul acestei cărți este acela de a obișnui cititorul să scrie anumite programe în Borland C++ Builder. Pentru aceasta nu este necesară cunoașterea mai multor noțiuni privind OOP în C++. Totuși, pentru a lucra eficient, anumite noțiuni sunt necesare. Cei care nu cunosc OOP iau lucrurile ca atare și scriu propriile sevențe respectând în mod mecanic convențiile de adresare (desi acestea au o justificare serioasă pentru cine cunoaște OOP).

În acest paragraf prezentăm câteva aplicații ale OOP în C++ Builder.

- ⇒ Pentru a putea scrie cu ușurință programe care funcționează sub Windows, cu o interfață grafică de excepție, în Borland C++ Builder există un ansamblu de clase numit VCL (Visual Component Library).
- ⇒ Componentele pe care le utilizăm sunt de fapt obiecte, care rezultă ca instantieri ale anumitor clase. De exemplu, o clasă este **TButton**, care descrie butoanele astăzi de utilizate până acum.
- ⇒ Obiectele se găsesc memorate în **HEAP**. Numele obiectului, de exemplu **Button1** este de fapt un pointer către obiectul propriu-zis. Aceasta explică utilizarea operatorului "**->**" pentru a accesa datele și metodele obiectului.
- ⇒ După cum se știe, una din caracteristicile fundamentale ale OOP este moștenirea, prin care o clasă moștenește pe alta (sau pe altele). În acest fel, ansamblul tuturor claselor este organizat "ierarhic" obținându-se astfel însă numita ierarhie a claselor!
- ⇒ În orice ierarhie trebuie să existe o clasă aflată la bază. În VCL această clasă se numește **TObject**. Ea este înzestrată cu anumite metode, dintre care unele vor fi prezentate în acest paragraf.

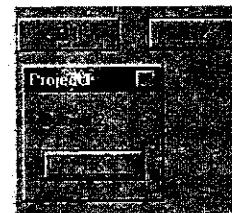
Priviți antetul funcției care răspunde pentru un buton la evenimentul **OnClick**: **void __fastcall TForm1::Button1Click(TObject *Sender)**

Observați că a fost transmis un parametru, numit **Sender**. Ce semnificație are?

- ⇒ Parametrul **Sender** este un pointer (adresă) către componenta care a reținut evenimentul.
- ⇒ Componentele sunt de multe tipuri (**TButton**, **TForm** etc). Dar toate aparțin unor clase care moștenesc clasa **TObject**. Prin urmare, parametrul **Sender** este pointer către această clasă.
- ✓ Așa cum am precizat, numele unui obiect, de exemplu **Button1**, este și el pointer către o componentă de tip **TButton**. Prin urmare, cei doi

pointeri pot fi comparați în sensul egalității. Aceasta creează posibilitatea determinării componentei care a reținut mesajul.

- Un formular are două butoane. Ambelor le atașăm o unică metodă care afisează un sir de caractere cu rolul de a transmite utilizatorului care este obiectul care a reținut evenimentul:

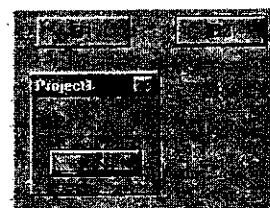


În programul anterior modificăm funcția care răspunde ambelor butoane.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    if (Sender == Button1) ShowMessage("Butonul 1");
    else ShowMessage("Butonul 2");
}
```

Problema pe care ne-o punem în continuare este de a avea acces la metodele și proprietățile obiectului transmis prin pointerul **Sender**. Am învățat că în C++ doi pointeri pot fi diferenți prin faptul că exprimă altă adresă dar și prin faptul că pot fi către tipuri diferite. Dacă s-a apăsat un buton, parametrul **Sender** este un pointer către buton, dar... atenție.. de tipul **TObject*** și nu **TButton***. De aici rezultă că putem accesa datele și metodele doar dacă utilizăm operatorul de conversie explicită, ca în exemplul următor.

- Se dau două butoane. Se cere ca la apăsarea oricărui dintre ele să se afișeze continutul proprietății **Caption** a butonului apăsat.



Pentru oricare buton apăsat răspunde o unică funcție:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ShowMessage(((TButton*)Sender)->Caption); }
```

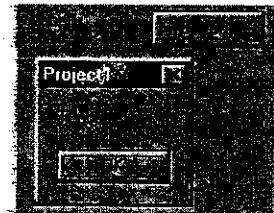
Clasa **TObject** are câteva metode care în anumite cazuri se pot dovedi utile.

- ⇒ O metodă care returnează numele clasei:

```
ShortString ClassName();
```

- ✓ **ShortString** este un tip care reține un sir de până la 255 caractere.

- Pe formular se găseste o componentă de tip **TLabel** și alta de tip **TButton**. La ambele răspunde o unică funcție care are rolul de a afișa tipul componentei asupra căreia s-a executat click.



```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ShowMessage (Sender->ClassName());}
```

⇒ O metodă care permite testarea clasei căreia îi aparține componentă:

```
bool ClassNameIs(AnsiString);
```

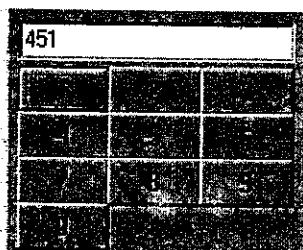
Dacă clasa coincide cu cea transmisă prin parametrul metodei returnează **true**, altfel returnează **false**.

- Rezolvați problema de mai sus prin utilizarea metodei **ClassNameIs()**.

Pentru ambele componente răspunde funcția de mai jos:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if (Sender->ClassNameIs("TButton"))
    ShowMessage("TButton");
  else
    ShowMessage(" TLabel");}
```

- Ați văzut cum funcționează aplicația **Calculator**, furnizată odată cu Windows-ul. Utilizatorul execută click pe mai multe butoane, unde fiecare buton are semnificația de cifră. În acest timp, ecranul calculatorului afișează numărul format. Cum se realizează aceasta?



- Mai întâi se așeză pe formular 10 butoane. Proprietatea **Caption** a fiecărui buton reține o cifră.
- Așezăm pe formular și o componentă de tip **TEdit** (care simulează "ecranul")

- Fiecare componentă (deci și cele de tip **TButton**) are o proprietate numită **Tag**, de tip **int**. Valoarea implicită a ei este 0, dar o putem modifica cu ajutorul inspectorului de obiecte. Astfel, pentru butonul care afișează 1, Tag va fi 1, pentru butonul care afișează 2 Tag va fi 2...s.a.m.d.

- Vom scrie o singură funcție care răspunde tuturor butoanelor și care concatenează sirul deja afișat de **Edit1** cu sirul obținut din conversia către sir a conținutului **Tag-ului** fiecărui obiect în parte. Iată-o:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ Edit1->Text=Edit1->Text+IntToStr(((TButton*)Sender)->Tag);}
```

Probleme propuse

1. Prin utilizarea unei componente de tip **TLabel** afișați mesajul "Acesta este mesajul meu". Cerinte: culoarea de fond: galben, font **Times New Roman**, mărimea 12, stil **Bold**, culoarea font-urilor rosu. Mesajul va fi aliniat în centru. Textul va fi afișat pe 2 rânduri.
2. Scrieți un program care afișează de câte ori a fost apăsat un buton.

Metoda 1 de rezolvare



Pentru a rezolva problema trebuie declarată o variabilă globală, care să rețină de câte ori a fost apăsat butonul.

O astfel de variabilă se declară, așa cum știm din C++, în modul. Din start, variabila este și inițializată cu '0'.

```
TForm1 *Form1;
int Nr_Apasari;
```

La apăsarea butonului răspunde funcția:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Nr_Apasari++;
  Label1->Caption=IntToStr(Nr_Apasari);
}
```

Metoda 2 de rezolvare.

În general, fiecare componentă are o proprietate de tip **int**, numită **Tag**. Valoarea initială implicită a ei este 0, dar programatorul poate acorda altă valoare cu ajutorul inspectorului de obiecte. În aceste condiții, rezolvarea problemei s-ar face ca mai jos, dacă utilizăm **Tag-ul** butonului.

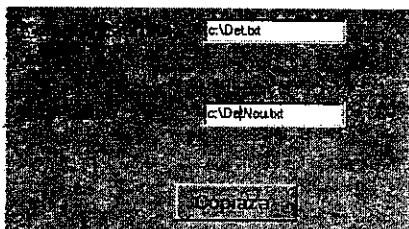
```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Label1->Caption=IntToStr(++Button1->Tag);
}

```

3. Scrieți un program care adună, scade, înmulțește sau împarte două numere reale. Cele 4 operații pot fi comandate cu ajutorul a 4 butoane. Programul va valida intrările și va gestiona focus-ul.
4. Scrieți un program care simulează un calculator de buzunar. Acesta va putea efectua cele 4 operații și va fi înzestrat cu memorie.
5. Scrieți un program care creează un fișier text. Datele se introduc de la tastatură. Pentru introducere se folosește o componentă de tip **TEdit** și un buton. La apăsarea butonului se scrie o linie din fișier. Formularul va contine și un buton de închidere.
6. Scrieți un program care copiază un fișier text existent, cu nume citit de la tastatură, iar în urma copierii noul fișier va avea, de asemenea, un nume citit de la tastatură. Se cere ca programul să verifice dacă primul fișier există. Dacă nu, se va da un mesaj de eroare. De asemenea, se cere verificarea existenței fișierului care se copiază. Dacă există, programul va afisa un mesaj de eroare.

Indicație. Propun că formularul să arate astfel:



Functia care răspunde la apăsarea butonului **Copiază** poate fi cea de mai jos. Testul existenței fișierului se face în mod clasic.

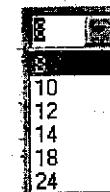
7.3 Liste

7.3.1 Liste de tip **TComboBox**

Există situații în care, chiar dacă utilizatorul poate selecta o singură opțiune, numărul total al opțiunilor este mare, motiv pentru care nu se folosesc butoane radio, ci liste derulante, care sunt obiecte de tip **TComboBox**.



Astfel de liste, numite și liste derulante, au avantajul că ocupă foarte puțin spațiu în formular, pentru că afișează numai selectia curentă.



Atrunci când utilizatorul dorește, se pot afișa toate opțiunile, așa cum se vede în figura alăturată - se execută click pe săgeată, apoi click pe opțiunea dorită.



lăță cum se selectează componenta de tip **TComboBox**.

- Opțiunile propriu-zise se încarcă cu ajutorul editorului de texte asociat proprietății **Items** (de tip **TStrings**, tip care va fi studiat).



Pentru lansarea editorului se execută click asupra celor trei puncte.

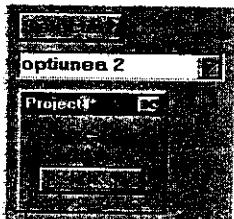


După scrierea opțiunilor, lista nu afișează nimic, dar un click asupra săgeții are ca efect afișarea lor. De acum este permisă selecția unei opțiuni și aceasta este ulterior afișată.



Opțiunile se scriu în mod obisnuit, căte una pe rând. Pentru a trece la un rând nou, se tastează **Enter**. Editorul este înzestrat cu un buton **OK**, care trebuie apăsat după introducerea opțiunilor.

- ⇒ proprietatea **Text**, de tip **AnsiString** are rolul de a retine opțiunea selectată. Dar, prin utilizarea inspectorului de obiecte, se poate atribui de la început o opțiune. Aceasta va fi afisată de la început de listă.
- ⇒ o proprietate specifică listelor este **Style**, care particularizează lista de selecție. Noi vom studia două stiluri particulare de liste derulante și anume:
 - **csDropDown** – Utilizatorul poate scrie în zona de selecție
 - **csDropDownList** – Utilizatorul nu poate scrie în zona de selecție. El poate numai selecta o opțiune și aceasta este afisată.
- Formularul afisează o listă derulantă și un buton. La apăsarea butonului se afisează selecția efectuată de către utilizator!



Opțiunile se scriu, așa cum am arătat, cu ajutorul editorului apelat din cadrul proprietății **Items**. De asemenea, opțiunea implicită s-a obținut scriind-o direct în proprietatea **Text** a listei. La apăsarea butonului răspunde funcția:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ShowMessage(ComboBox1->Text); }
```

7.3.2 Tipul **TStrings**

Variabilele cu tipul **TStrings** au rolul de a gestiona mai multe siruri de caractere de tipul **AnsiString**. O astfel de variabilă este, dacă vrei un vector de siruri de caractere. Primul sir are indicele 0 și ultimul **n-1**.

- ✓ O mulțime de componente au căte o proprietate de acest tip.
- ✓ Proprietățile și metodele acestui tip le vom studia cu ajutorul proprietății **Items** (care are tipul **TStrings**) a componentelor de tip **TComboBox**.

Iată câteva metode, mai importante:

- ⇒ Metoda **Add** are rolul de a adăuga sirul transmis ca parametru ca ultim sir în lista de siruri. Ea returnează indicele ultimului sir:

```
int Add(AnsiString S);
```

Exemplu: la apăsarea unui buton se adaugă o nouă opțiune unei liste de tip **TComboBox**:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ComboBox1->Items->Add("Optiune adaugata"); }
```

- ⇒ Metoda **Clear** are rolul de a sterge toate sirurile din lista de siruri:

```
void Clear(void);
```

Exemplu: se sterg toate sirurile din lista de siruri:

```
ComboBox1->Items->Clear();
```

- ⇒ Metoda **Delete** are rolul de a sterge sirul de indice dat din lista de siruri:

```
void Delete(int Index);
```

Exemplu: se sterg sirul de indice 0 (de fapt, prima opțiune):

```
ComboBox1->Items->Delete (0);
```

- ⇒ Metoda **Insert** are rolul de a insera un sir înaintea sirului de indice transmis ca parametru:

```
void Insert(int Index, AnsiString S);
```

Exemplu: se inserează o primă opțiune în lista:

```
ComboBox1->Items->Insert (0, "Prima Optiune");
```

- ⇒ Metoda **Move** are rolul de a insera sirul de indice **CurIndex** pe poziția **NewIndex**:

```
void Move(int CurIndex, int NewIndex);
```

Exemplu: se inversează primele două siruri în lista opțiunilor:

```
ComboBox1->Items->Move (0, 1);
```

- ⇒ Metoda **LoadFromFile** are rolul de a încărca lista de siruri dintr-un fisier text:

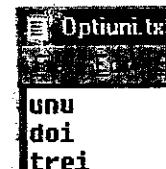
```
void LoadFromFile(AnsiString FileName);
```

- ⇒ Metoda **SaveToFile** are rolul de a salva lista de siruri într-un fisier text:

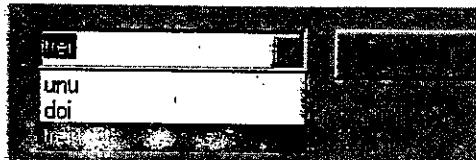
```
void SaveToFile(AnsiString FileName);
```

- ✓ Aceste două metode sunt extrem de eficiente. Fisierul text nu trebuie declarat, citirea / scrierea se face la nivelul întregului fisier.

- Formularul va contine un obiect de tip **TComboBox**. Lista opțiunilor va fi încărcată, la apăsarea unui buton, dintr-un fisier text.



Fisierul text se creează, de exemplu, cu utilitarul **NotePad**. Fisierul se salvează sub un nume ales de dumneavoastră.



La apăsarea butonului răspunde funcția de mai jos, care încarcă din fișier opțiunile:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ComboBox1->Items->LoadFromFile("C:\Optiuni.txt");}
```

Dată și câteva proprietăți ale acestui tip:

- ⇒ **Capacity** – reține numărul de siruri din listă;
- ⇒ **Strings[int Index]** – reține sirul de indice dat.

Exemplu: listez a doua opțiune:

```
ShowMessage(ComboBox1->Items->Strings[1]);
```

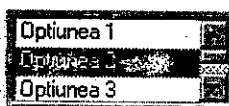
- ⇒ **Text** – reține toate sirurile (separate prin CR, LF).



Exemplu: afisez toate opțiunile:

```
ShowMessage(ComboBox1->Items->Text);
```

7.3.3 Liste de selectie de tip TListBox



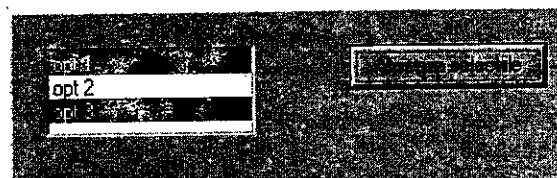
Înă că cum arată o astfel de listă, în care este selectată opțiunea 2.



Selectia obiectului de tip **TListBox** se face așa cum se vede alăturat:

- ✓ Avantajul utilizării acestor liste este dat de faptul că ele permit selecția multiplă, adică este posibil ca utilizatorul să selecțeze mai multe opțiuni.
- ✓ Pentru a putea selecta mai multe opțiuni se operează cu mouse-ul, dar se ține tasta **CTRL** apăsată.
- ⇒ Principala proprietate a acestui obiect este **Items**, de tipul **tstrings**. Ea reține lista de opțiuni, care este afișată. Fiecare opțiune este, de fapt, un sir din listă. Aceasta înseamnă că toate proprietățile și metodele acestui tip pot fi folosite la fel ca până acum. Să de această dată, se dispune de un editor de texte prin care programatorul poate scrie, de la început, lista opțiunilor. Încercăți!

- ⇒ Proprietatea **Sorted**, de tip **bool**, permite ca lista de opțiuni să fie sortată alfabetic, înainte de rularea programului. Pentru aceasta este necesar să rețină valoarea **true**.
- ⇒ Proprietatea **MultiSelect**, de tip **bool**, permite ca, în cadrul unei astfel de liste, utilizatorul să poată selecta mai multe opțiuni în același timp. Pentru aceasta trebuie ca ea să rețină valoarea **true**, contrar, o singură opțiune poate fi selectată.
- ⇒ Proprietatea **ItemIndex** de tip **int**; reține indexul opțiunii selectate – adică numărul de ordine al șirului care conține opțiunea respectivă. Se folosește în cazul în care **MultiSelect** reține **false**. În cazul în care nu a fost efectuată selecția, reține -1. Reamintim că prima opțiune din listă are indicele 0.
- ⇒ Proprietatea **SelCount** de tip **int**; reține numărul de opțiuni selectate la un moment dat, în cazul selecției multiple.
- ⇒ Proprietatea **Selected[int Index]** de tip **bool**; reține sub forma unui vector, cu componente de tip **bool**, ce opțiuni au fost selectate. De exemplu, dacă **Selected[3]==true** atunci a patra opțiune a fost selectată.
- ✓ În cazul în care numărul de opțiuni este prea mare pentru ca acestea să poată fi simultan afișate, obiectul este înzestrat automat cu bară de defilare.
- Formularul afișează o listă pentru care este permisă selecția multiplă (vezi proprietatea **MultiSelect**). Se cere ca, la apăsarea unui buton, să se afișeze toate opțiunile selectate!



```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ for (int i=0;i<ListBox1->Items->Count;i++)
    if (ListBox1->Selected[i]==true)
        ShowMessage(ListBox1->Items->Strings[i]); }
```

7.3.4 Liste de selectie de tip TTreeView

Până acum am studiat numai liste pentru care putem selecta o opțiune sau mai multe dintre cele **n** posibile. Cu toate că acesta este cazul cel mai frecvent, există și situații când astfel de liste se dovedesc a fi ineficiente. Mă refer aici la cazul în care opțiunile sunt arborescente, așa cum va

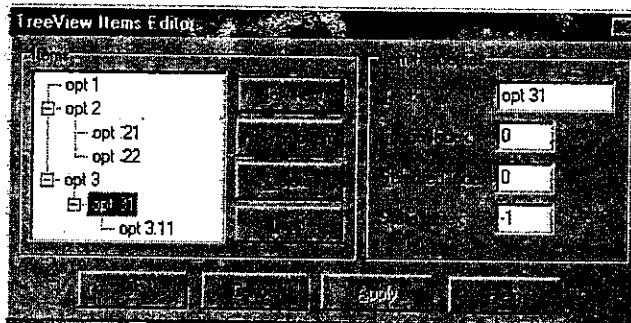
rezultă din exemplele care vor fi date. Acest din urmă caz este tratat de componenta de tip **TTreeView**.



Pentru a lucra cu un obiect de acest tip se atașează formularului simbolul alăturat.

După amplasare, pe formular apare un dreptunghi alb. Dacă-l selectăm, inspectorul de obiecte afișează proprietățile obiectelor de acest tip.

- Proprietatea cea mai importantă este **Items**, de tip **TTreeNodes**. Pentru început, observăm că un obiect de tip **TTreeNodes** poate reține un arbore oarecare. Structura de date propriu-zisă este reținută de această proprietate. Selectia ei (dublu click) asupra celor trei puncte determină lansarea unui editor specializat să accepte ca date de intrare aceste structuri:



Cu ajutorul editorului se poate scrie cu ușurință structura, prin utilizarea butoanelor **New Item**, **New SubItem**, **Delete** (dacă am gresit).

Deja putem testa modul de funcționare a programului. Iată cum apare structura în timpul executării:

```
opt 1
  opt 2
    opt 21
    opt 22
  opt 3
```

✓ Semnul "+" informează utilizatorul că opțiunea legată de el poate fi expandată, adică să pot fi afișate pentru ea opțiunile subordonate! Un click asupra sa determină expandarea.

✓ Semnul "-" informează că opțiunea a fost expandată. Un click asupra sa determină ca opțiunile subordonate să nu mai fie afișate.

Așa cum am arătat, întregul arbore are structura **TTreeNodes**. În schimb, un nod al arborelui are structura **TTreeNode**. Ambele tipuri sunt prezentate în carte. Deocamdată mai reținem o proprietate a tipului **TTreeView**, numită **Selected**, și alta a tipului **TTreeNode**, numită **Text**.

- ⇒ Proprietatea **Selected** a tipului **TTreeView** reține modul selectat de utilizator (un pointer către el). Fiind vorba de un nod, variabila are tipul **TTreeNode**.
- ⇒ Proprietatea **Text** a tipului **TTreeNode** este de tipul **AnsiString**, și reține textul afișat al nodului respectiv (de exemplu **opt 1**).
- Formularul contine o listă de tip **TreeView** și un buton la apăsarea căruia se afișează selecția efectuată!



Funcția care se află în spatele butonului este:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ ShowMessage(TreeView1->Selected->Text); }
```

Să recapitulăm:

- Lista, în ansamblul ei este de tipul **TTreeView**;
- ea reține arborele prin proprietatea **Items**, de tip **TTreeNodes**;
- un nod al arborelui are tipul **TTreeNode**.
- ✓ Vârful arborelui este implicit, nu este reținut!

Tipul **TTreeNode**

Iată câteva proprietăți ale acestui tip (tipul unui nod):

- ⇒ **Count** - reprezintă numărul de noduri subordonate de nodul dat;
- ⇒ **Item** - este un vector, de indicii întregi care reține nodurile subordonate. Primul nod subordonat are indicele 0, al doilea are indicele 1, ultimul are indicele **count-1**;
- ⇒ **Text** - textul asociat unui nod;

Exemplu: o funcție care, pentru un nod selectat, listează textele nodurilor subordonate. Funcția se rulează la apăsarea unui buton:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ for (int i=0;i<TreeView1->Selected->Count-1;i++)
  ShowMessage(TreeView1->Selected->Item[i]->Text);
}
```

⇒ **HasChildren** - proprietate de tip `bool`, reține `true` dacă nodul are noduri subordonate. Fie secvența:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{if (TreeView1->Selected->HasChildren)
 ShowMessage ("are copii");
 else ShowMessage ("n-are copii");
}
```

Iată și câteva metode ale acestui tip:

⇒ **AlphaSort()**: funcție de tip `bool`. Are rolul de a sorta în ordinea alfabetică a proprietății `Text`, toate nodurile subordonate nodului căruia î se aplică. Returnează `true` în cazul în care operația a reușit.

Exemplu: sortarea alfabetică a nodurilor subordonate nodului selectat:

```
TreeView1->Selected->AlphaSort();
```

⇒ **Delete()** - sterge nodul și toate nodurile subordonate.

Exemplu: `TreeView1->Selected->Delete();`

⇒ **DeleteChildren()** - sterge toate nodurile subordonate unui nod.

Exemplu: `TreeView1->Selected->DeleteChildren();`

⇒ **Expand(valoare de tip bool)** expandează nodul (afisează nodurile subordonate). Dacă este apelată cu `true`, expandarea se face total, adică afisează și nodurile subordonate fililor etc. Contrar, expandarea se face numai pe un nivel.

Exemplu: expandează nodul selectat;

```
TreeView1->Selected->Expand(true);
```

⇒ **Collapse(valoare de tip bool)**. Nu mai afisează fiil nodului. Dacă se apelează cu `true`, la următoarea expandare, indiferent de valoarea de adevăr cu care a fost apelată, se afisează toți fiil. Contrar, dacă se face expandarea cu `false`, se afisează numai primul nivel.

Exemplu: nu mai afisează fiil nodului selectat

```
TreeView1->Selected->Collapse(true);
```

Tipul `TTreeNode`

Un obiect de acest tip reține și tratează arborele. Un nod al arborelui are tipul `TTreeNode`.

⇒ Proprietatea `Count` - reține numărul de noduri ale arborelui.

Exemplu: afisez această valoare:

```
ShowMessage (IntToStr(TreeView1->Items->Count));
```

Iată și câteva metode:

⇒ **TTreeNode Add(TTreeNode Nod, AnsiString S)**; Adaugă un nod arborelui. Aici `Nod` este un nod aflat pe același nivel cu nodul care urmează să fie adăugat, iar `S` este sirul reținut de proprietatea `Text` a nodului care urmează să fie adăugat.

Exemplu: Se adaugă un nod pe același nivel cu nodul selectat:

```
TreeView1->Items->Add(TreeView1->Selected, "Nod nou");
```

⇒ **TTreeNode AddChild(TTreeNode Nod, AnsiString S)**; Adaugă nodului `Nod` un fiu, care reține `S`.

Exemplu: Se adaugă un nod fiu pentru nodul selectat:

```
TreeView1->Items->AddChild(TreeView1->Selected, "Nod nou");
```

⇒ **Clear()**; Sterge toate nodurile.

Exemplu: Sterg întreg arborele:

```
TreeView1->Items->Clear();
```

⇒ **Delete(TTreeNode Nod)**; Sterge un nod. Evident, se sterg și toti descendenții acestuia.

Exemplu: sterg nodul selectat:

```
TreeView1->Items->Delete(TreeView1->Selected);
```

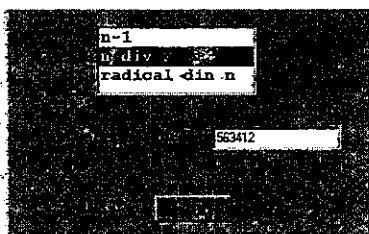
Câteva evenimente la care răspund obiectele de tip `TTreeView`

- **On Collapsing** - evenimentul se produce după comanda utilizatorului de a nu mai afisa fiil unui nod, dar înainte ca acestia să nu mai fie afișați.
- **On Collapsed** - evenimentul se produce după comanda utilizatorului de a nu mai afisa fiil unui nod, dar după ce acestia nu mai sunt afișați.
- **On Expanding** - evenimentul se produce după comanda utilizatorului de a afisa fiil unui nod, dar înainte ca acestia să fie afișați.
- **On Expanded** - evenimentul se produce după comanda utilizatorului de a afisa fiil unui nod, dar după ce acestia sunt afișați.

Probleme propuse

- Prin utilizarea unui obiect de tip **TListBox** permiteți utilizatorului să selecteze algoritmul prin care se testează dacă un număr natural citit (n) este sau nu număr prim. Utilizatorul va putea selecta între:
 - Algoritmul prin care se testează toți divizorii până la $n-1$;
 - Algoritmul prin care se testează toți divizorii până la \sqrt{n} ;
 - Algoritmul prin care se testează toți divizorii până la radical din n .

Indicație. Propun formularul:

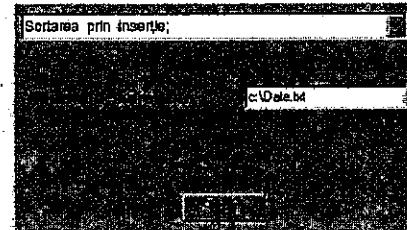


În spatele butonului **Start**, atașați o funcție care apelează una sau alta dintre cele 3 funcții corespunzătoare algoritmilor utilizati.

- Prin utilizarea unei liste de tip **TComboBox** permiteți utilizatorului să selecteze algoritmul prin care se sortează n valori întregi, citite dintr-un fișier text (pe prima linie este n , pe a doua cele n valori întregi). Programul va include următorii algoritmi de sortare:

- Sortarea prin interschimbare;
- Sortarea prin calculul maximului (minimului);
- Sortarea prin inserție;
- Sortarea prin interclasare;
- Sortarea rapidă.

Indicație. Puteti crea un formular așa cum arată cel alăturat:



- Scriți o funcție care vizualizează un arbore binar, creat. Arborele va fi creat în **Heap**.

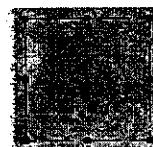
7.4 Alte componente des folosite

7.4.1 Componenta de tip **TImage**

De multe ori este necesar ca ferestre să afișeze anumite imagini. Cum obținem acest efect? Prin utilizarea componentelor de tip **TImage**.



Iconul unei astfel de componente se găsește în pagina **Additional** a paletei de componente.



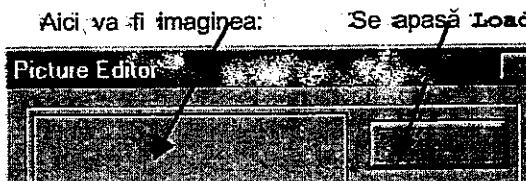
Dată cum arată componenta după așezarea ei pe formular:

Vom studia câteva proprietăți ale ei:

⇒ **Picture** – o astfel de componentă poate retine, la un moment dat o singură imagine de tip **bitmap (.bmp)**.

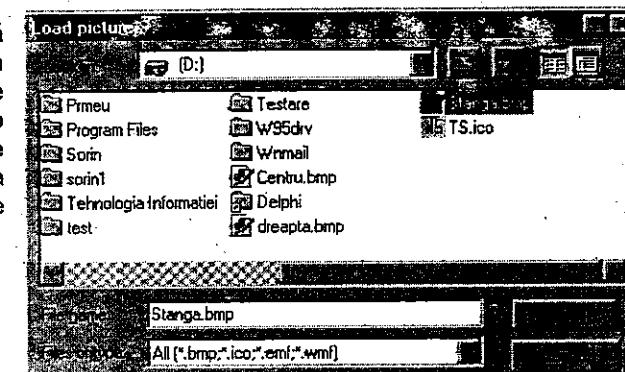


Pentru încărcarea imaginii din fișier se execută click asupra celor trei puncte:



Aici va fi imaginea:

Se apasă Load



Automat se lansează în execuțare un "încărător de imagini". Apare o cutie de dialog care permite selectarea fișierului ce conține imaginea:

⇒ **Width**, **Height** sunt proprietăți care rețin lățimea și înălțimea dreptunghiului în care este afisată imaginea.

Cum putem afla dimensiunile imaginii (în pixeli)?

⇒ **AutoSize** este o proprietate de tip **bool** a componentelor de tip **TImage**. Valoarea implicită este **false**, dar dacă o stabilim la **true**, componenta va avea exact dimensiunea necesară că să încapă întreaga imagine. Acum putem vedea care sunt valorile reținute de **Width** și **Height**.

✓ Versiunile mai noi de **C++ Builder** conțin cutii de dialog care afisează atât imaginea selectată cât și dimensiunile ei!

Nu întotdeauna dimensiunile dreptunghiului care afisează imaginea coincid cu dimensiunile imaginii. Ce este de făcut?

a) Dacă este convenabil (adică dacă începe pe formular) scriem în dreptul proprietății **AutoSize** valoarea **true**.

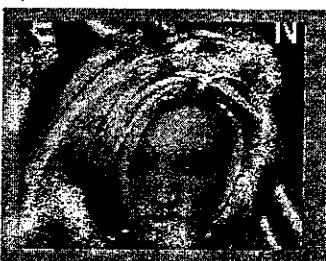


În acest caz imaginea începe în totalitate!

Dar nu întotdeauna aceasta se poate face. Există situații în care imaginea nu începe pe formular, sau, prin dimensiunile ei, este inestetică!

În astfel de cazuri se apelează la alte proprietăți ale obiectelor de tip **TImage**!

⇒ **Stretch** (deformare). Este o proprietate de tip **bool**. Dacă reține **true**, imaginea va fi deformată de așa natură încât să încapă exact în suprafața de afisare! Contrar, deformarea nu are loc.



Exemplu:

Stretch reține **true**.

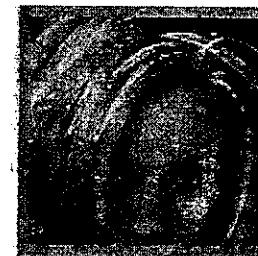
Dreptunghiul de afisare are dimensiunile (150,120).

Imaginea are dimensiunile: (189,187).

Dacă **Stretch** reține **false**, atunci utilizăm altă proprietate:

⇒ **Center** - dacă reține **true**, imaginea este afisată în centrul dreptunghiului de afisare, iar dacă reține **false** ea este afisată în colțul din stânga sus.

Exemplu: imaginea de mai sus se afisează într-un dreptunghi de (120,120).



Center:**true**



Center:**false**

7.4.2 Componenta de tip **TImageList**

Cum se procedează atunci când se dorește ca aceeași componentă să afișeze, pe rând, mai multe imagini?

Listă de imagini este un obiect de tip **TImageList**. În esență, o astfel de listă conține mai multe imagini (**n**) de aceeași dimensiune, numerotate între 0 și **n-1**. Aceste imagini pot fi folosite pentru a fi afisate prin intermediu altor obiecte.

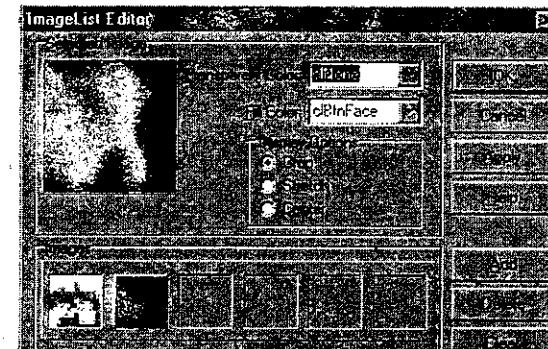


Îată simbolul grafic care se amplasează pe formular:

Odată amplasat, prin dublu click asupra sa se lansează în executare o cutie de dialog care permite selecția imaginilor de pe suport extern:

Principalele operații care pot fi efectuate sunt:

- **Add** - Adaugă o nouă imagine în listă. Apare o altă cutie de dialog, care vizualizează conținuturile folder-elor, de unde utilizatorul poate selecta fisierele dorite.



- **Delete** – Sterge o imagine din listă și anume cea selectată.
- **Clear** – Sterge toate imaginile din listă.

Cutia de dialog care permite încărcarea imaginilor are trei opțiuni:

- Crop** – dacă dimensiunile dreptunghiului care reține imaginea sunt mai mari decât cele ocupate de imagine, imaginea este afișată în colțul din stânga sus al acestuia, iar dacă nu, se afișează atât cât începe din imagine și anume partea aflată în colțul din stânga sus al acesteia.
- Stretch** – imaginea este transformată de așa natură încât să încapă exact în dreptunghiul de afișare.
- Center** – imaginea este pusă în centrul dreptunghiului de afișare. Dacă nu începe, este decupată.

După încărcarea imaginilor se execută click asupra butonului **OK** și lista este construită. Principalele proprietăți ale obiectelor de tip **TImageList** sunt:

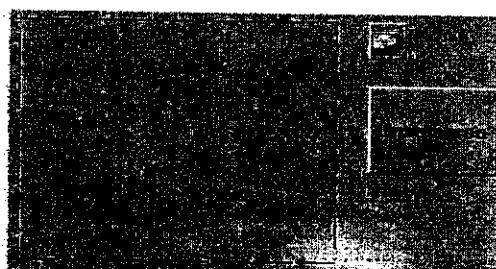
- ⇒ **Width** (lățime), **Height** (înălțime) – dimensiunile dreptunghiurilor care vor reține imaginile. Fiecare dimensiune se măsoară în pixeli. Aceste proprietăți trebuie initializate înainte de încărcarea imaginilor. Implicit, se reține 16 și 16.
- ⇒ **Count** – reține numărul de imagini din listă.

Builder-ul este livrat cu câteva fișiere **.bmp**, toate cu dimensiunile 240x180 aflate în:

Program Files/Borland/CBuilder/Images/Splash/16color.

- Să se scrie un program care afișează toate aceste imagini. Pentru afișarea următoarei imagini se va utiliza un buton.

Pasul 1. Formularul trebuie să conțină 3 componente: una de tip **TImage**, care afișează imaginea, o componentă de tip **TButton** și una de tip **TImageList**.



Componenta **Image1** va reține pentru **Width 240** și pentru **Height 180**.

Aceleasi valori vor fi reținute de aceleasi proprietăți ale componentei **ImageList1**.

Pasul 2. Se încarcă lista de imagini cu fișierele enumerate anterior.

Pasul 3. Declaram o variabilă globală **i**, de tip **int**.

```
TForm1 *Form1;
int i;
```

Pasul 4. La evenimentul **onClick**, butonul va răspunde cu funcția:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Image1->Picture=0;
    ImageList1->Draw(Image1->Canvas,0,0,i);
    if (i<ImageList1->Count-1) i++;
    else i=0;
}
```

- ⇒ Pentru a desena o imagine se folosește o metodă a obiectelor de tip **TImageList**, numită **Draw**.
- ⇒ **Draw** (Destinație, Coordonatele colțului din stânga sus, indice imagine).
- ⇒ **Canvas** este proprietatea care are semnificația de suprafață pe care se poate desena.
- ✓ Si formularul are proprietatea **Canvas**. De exemplu, se poate desena pe suprafața lui tot cu metoda **Draw** a listei de imagini.

```
ImageList1->Draw(Canvas,0,0,i);
```

- Formular cu imagine. Uneori dorim ca fereastra să afișeze o anumită imagine. Dar imaginea trebuie să se păstreze și atunci când maximizăm fereastra și atunci când o redimensionăm cu mouse-ul.



Pe formular amplasăm un obiect de tip **TImage**, aliniat **alClient**. De asemenea, proprietățile **AutoSize** și **Stretch** ale acestuia rețin **true**. Apoi încarcăm imaginea. Toate celelalte componente se pot aseza deasupra acestui desen.

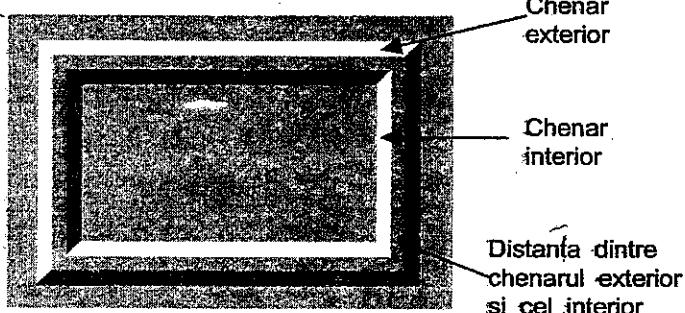
7.4.3 Componenta de tip TPanel

Este una dintre cele mai utilizate componente, putând lua diverse forme. În dimii mari, are forma unui dreptunghi. Pe suprafața sa se pot amplasa alte componente, de exemplu butoane, componente de tip **TEdit**, **TImage** etc.



Pentru utilizarea ei se folosește iconul alăturat, care se amplasează pe formular.

Mai jos este prezentată o componentă de tip **TPanel**. Se observă aspectul său tridimensional. Iată elementele care-l caracterizează:



Iată câteva proprietăți ale acestui tip de obiect:

⇒ **BevelOuter** -Stilul chenarului exterior. Poate lua una din valorile:

- bvNone** - chenarul exterior lipsește;
- bvLowered** -chenarul exterior este coborât;
- bvRaised** - chenarul exterior este ridicat.

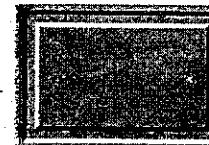
⇒ **BevelInner** -Stilul chenarului interior. Poate lua una din valorile:

- bvNone** - lipseste;
- bvLowered** -este coborât;
- bvRaised** - chenarul este ridicat.

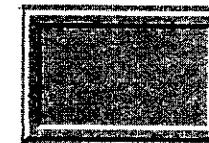
⇒ **BevelWidth** -grosimea comună, în pixeli atât a chenarului interior cât și exterior.

⇒ **BorderWidth** -distanța în pixeli între cele două chenare.

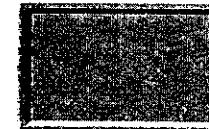
Cu ajutorul acestor proprietăți se poate particulariza aspectul grafic al obiectelor de tip **TPanel**, așa cum rezultă din exemplele următoare:



BevelInner:	bvRaised
BevelOuter:	bvLowered
BevelWidth:	6
BorderWidth	9

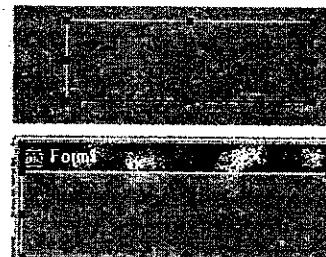


BevelInner:	bvLowered
BevelOuter:	bvRaised
BevelWidth:	6
BorderWidth	9

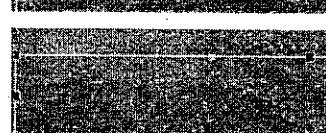


BevelInner:	bvLowered
BevelOuter:	bvNone

⇒ **Align** - precizează locul unde este amplasată componenta. Ea poate retine una dintre următoarele constante:



AlNone - acolo unde o asează programatorul;



AlTop -sus. În cazul în care formularul este înzestrat cu meniu, componenta va fi așezată după acesta. Este utilizată la crearea barei de comenzi rapide.



AlBottom -jos. Se utilizează pentru crearea "barei de stare".



AlLeft, Al Right -stânga sau dreapta. Se utilizează pentru crearea unor bare de comenzi.



Al Client -pe întreaga suprafață a formularului. Această formă de aliniere este rar utilizată.

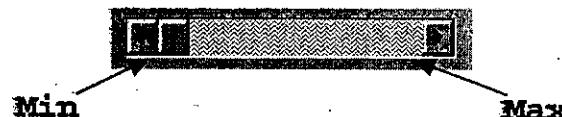
7.4.4 Componenta de tip TScrollBar

O componentă de tip **TScrollBar** se mai numește și bară de scroll.



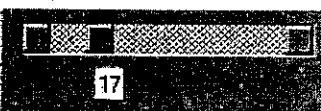
Iată care este obiectul care se amplasează pe formular.

Bara de scroll arată astfel:



Prin bara de scroll se pot selecta numere întregi cuprinse între o valoare minimă și una maximă (**Min** și **Max** sunt proprietăți ale obiectelor de acest tip). De asemenea, bara de scroll retine poziția selectată prin proprietatea **Position**. Posizionarea barei se face cu ajutorul proprietății **Kind**, afișată de inspectorul de obiecte (**SbHorizontal** sau **SbVertical**). În general, programul poate regăsi numărul selectat printr-o funcție care răspunde evenimentului **OnScroll**.

De exemplu, dorim ca după operația de scroll, atunci când mouse-ul se stăzonează asupra barei să fie afișat numărul selectat, așa cum rezultă din figura următoare:

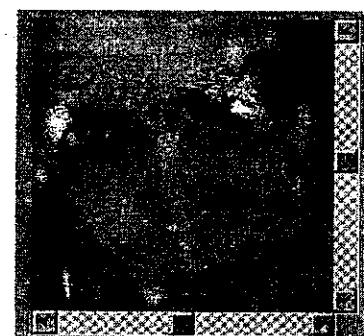


Iată funcția care realizează aceasta și răspunde evenimentului **OnScroll**:

```
void __fastcall TForm1::ScrollBar1Scroll(TObject *Sender,
    TScrollCode ScrollCode, int &ScrollPos)
{ ScrollBar1->Hint=IntToStr(ScrollBar1->Position); }
```

- Se cere ca programul să afișeze o imagine cu ajutorul barelor de scroll. Evident, se presupune că imaginea are dimensiuni mai mari decât suprafața de afișare!

Dimensiunile imaginii sunt **526x620**, iar suprafața de afișare este de **200x200**.



Pasul 1. Așezăm pe formular o componentă de tip **TPanel**. Proprietățile **Width** și **Height** ale acesteia vor reține **200** și **200**.

Pasul 2. Pe componentă de tip **TPanel** se asează o imagine de tip **TImage**. trebuie ca aceasta să ocupe întreaga suprafață a componentei de tip **TPanel**. Pentru aceasta, ea va fi aliniată inițial **alClient**. O astfel de alinieră înseamnă că dimensiunile ei vor fi aceleasi cu ale componentei de tip **TPanel**. Întrucât dimensiunile ei trebuie să poată expune întreaga imagine, o aliniem, din nou **alNone**. În continuare stabilim dimensiunile componentei la **526x620** și încărcăm imaginea (fisierul **bmp**).

Pasul 3. Atăzăm cele două bare de scroll, una orizontală, alta verticală (proprietatea **Kind**). Lipirea acestora de componentă de tip **TPanel** și dimensionarea lor se face cu ajutorul mouse-ului. Proprietatea **Min** a fiecărei bare va reține valoarea **0** (valoarea implicită), iar proprietatea **Max** va reține pentru bara orizontală **526-200**, iar pentru cea verticală **520-200** (nu uitați că imaginea este inițial afișată în colțul din stânga sus al componentei de tip **TImage**).

Pasul 4. Pentru fiecare bară scriem căte o funcție care răspunde evenimentului **OnScroll**. Aceste funcții nu fac altceva decât să fixeze coordonatele colțului din stânga sus ale componentei de tip **TImage** față de cele ale componentei de tip **TPanel** în funcție de valoarea selectată.

```
void __fastcall TForm1::ScrollBar1Scroll(TObject *Sender,
    TScrollCode ScrollCode, int &ScrollPos)
{ Image1->Left=-ScrollBar1->Position; }

void __fastcall TForm1::ScrollBar2Scroll(TObject *Sender,
    TScrollCode ScrollCode, int &ScrollPos)
{ Image1->Top=-ScrollBar2->Position; }
```

7.4.5 Componenta de tip TUpDown

Componentele de tip **TUpDown** folosesc mouse-ul pentru incrementarea, decrementarea unei anumite valori.



Pentru a folosi un astfel de obiect, se amplasează pe formular simbolul alăturat.

O astfel de componentă are proprietățile **Min**, **Max**, **Position**. Proprietatea **Position** reține o valoare inițială (implicit **0**). De asemenea, există și proprietatea **Increment** care are valoarea implicită **1**. În cazul în care utilizatorul dorește, aceasta poate fi modificată. Pentru incrementare / decrementare se execută click în portinea corespunzătoare a obiectului.



De exemplu, dorim ca un obiect de tip **TLabel** să afișeze o valoare care poate fi incrementată / decrementată cu ajutorul mouse-ului.

Initial, obiectul de tip **TLabel** va afișa valoarea reținută de proprietatea **Position** a obiectului de tip **TUpDown**. Apoi, evenimentului **OnClick** pentru obiectul de tip **TUpDown** îl se răspunde prin funcția:

```
void __fastcall TForm1::UpDown1Click(TObject *Sender, TUDBtnType
Button)
{ Label1->Caption=IntToStr(UpDown1->Position);}
```

- Lupa. O componentă de tip **TImage** se află amplasată pe o alta de tip **TPanel** și afisează o imagine. Alături, se găsește o componentă de tip **TUpDown**. La executarea unui click pe această ultimă componentă, componenta de tip **TImage** își dublează (triplează etc) laturile, sau, invers, laturile ajung la jumătate etc. Pentru a putea vizualiza întreaga imagine folosim bare de scroll. Mai jos puteți vedea două imagini afișate de program. Componenta de tip **TPanel** are dimensiunile 200x200.



Pentru realizarea acestei aplicații vom respecta următorii pași:

Pasul 1. Se asează pe formular componenta de tip **TPanel**. Proprietățile **Width** și **Height** vor reține fiecare 200.

Pasul 2. Urmează amplasarea pe **Panel** a unei componente de tip **TImage**. Pentru ca aceasta să ocupe întreaga suprafață a **Panel**-ului, o vom alinia mai întâi **alClient**, apoi, pentru a-i modifica dimensiunile o aliniem **alNone**. Proprietatea **Stretch** a componentei de tip **TImage** va reține **true**, pentru ca imaginea afișată să poată fi mărită sau micșorată, după dimensiunile ei.

Pasul 3. **Panel**-ul este încadrat de două bare de scroll. Acestea permit explorarea întregii imagini. Ele răspund evenimentului **OnScroll** prin funcțiile:

```
void __fastcall TForm1::ScrollBar1Scroll(TObject *Sender,
TScrollCode ScrollCode, int &ScrollPos)
{
  Image1->Left=-ScrollBar1->Position;
}
```

```
void __fastcall TForm1::ScrollBar2Scroll(TObject *Sender,
TScrollCode ScrollCode, int &ScrollPos)
{ Image1->Top=-ScrollBar2->Position;}
```

Pasul 5. Asează pe formular componenta de tip **TUpDown**. Proprietatea **Min** va reține 1, iar **Max** va reține 4 (deci putem mări imaginea de patru ori succesiv) sau o putem micșora. Pentru această componentă evenimentul **OnClick** îl răspunde funcția:

```
void __fastcall TForm1::UpDown1Click(TObject *Sender, TUDBtnType
Button)
{ Image1->Height=200*UpDown1->Position;
  Image1->Width=200*UpDown1->Position;
  ScrollBar1->Max=Image1->Width-200;
  ScrollBar2->Max=Image1->Height-200;
}
```

Funcția are rolul de a mări sau micșora suprafața de afișare a componentei de tip **TImage** și de a mări sau micșora plaja de valori afișată de fiecare bară de scroll. Normal, mărirea imaginii, impune pentru bara de defilare altă plajă de valori.

- ✓ În fapt se mărește (micșorează) suprafața de afișare a componentei de tip **TImage**, iar imaginea este deformată (proprietatea **Stretch**). Cu toate acestea **Panel**-ul expune numai o parte din suprafața de afișare.

7.4.6 Componete de tip **TStringGrid**

Componetele de tip **TStringGrid** permit vizualizarea unei matrice (așa cum o numesc matematicienii) sau tabel (după cum îl numesc economistii). Fiecare componentă (elementul de coordonate **i** și **j**) se numește celulă. În cazul obiectelor de tip **TStringGrid** o celulă poate reține un sir de caractere. Iată componenta care se amplasează pe formular:



Prezentăm principalele proprietăți ale obiectelor de acest tip:

- ⇒ **ColCount** -numărul de coloane;
- ⇒ **RowCount** -numărul de linii.

- ✓ Valoarea implicită reținută de cele două proprietăți este 5.
- ✓ Obiectele de acest tip sunt înzestrate cu bare de defilare pentru cazul în care suprafața de afișare este mai mică decât cea necesară.

Dimensiunea celulelor, în pixeli, este stabilită cu ajutorul altor două proprietăți:

- ⇒ **DefaultColWidth** – lățimea;
- ⇒ **DefaultRowHeight** – înălțimea.

În majoritatea cazurilor, prima linie (sau primele linii) și prima coloană (sau primele coloane) reținaza numitul "cap de tabel". Din acest motiv ele trebuie să nu participe la defilare, pentru ca utilizatorul să stie întotdeauna ce vede. Aici sunt utile următoarele proprietăți:

- ⇒ **FixedCols** - reține numărul de coloane care nu participă la defilare;
- ⇒ **FixedRows** - stabilește numărul de linii care nu participă la defilare.

Următoarele proprietăți stabilesc aspectul grafic al obiectului:

- ⇒ **Color** - stabilește culoarea celulelor care participă la defilare;
- ⇒ **FixedColor** - stabilește culoarea celulelor care nu participă la defilare;
- ⇒ **GridLineWidth** - stabilește lățimea linioarelor care marchează celulele (în pixeli). În cazul în care acestea sunt absente proprietatea trebuie să rețină 0.
- ⇒ **Cells[Coloana, Linie: Integer]: AnsiString**; reține sirul afișat de o celulă.

- ✓ Mai întâi se trece coloana apoi linia.
- ✓ Linile și coloanele se numerotează începând cu 0. Astfel regăsim vechea regulă: colțul din stânga sus are coordonatele (0,0).
- La apăsarea unui buton se completează obiectul următor de tip **TStringGrid** cu numere aleatoare mai mici ca 10. De asemenea, se numerotează linile și coloanele:

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{ int i,j;
for (i=1;i<=4;i++)
{ StringGrid1->Cells[0][i] = "Linia "+IntToStr(i);
  StringGrid1->Cells[i][0] = "Coloana "+IntToStr(i);
}
for (i=1;i<=4;j++)
  for (j=1;j<=4;j++)
    StringGrid1->Cells[i][j]=IntToStr(rand()%11);
}
```

	1	2	3	4
1	0	2	3	0
2	0	6	4	8
3	8	3	0	0

- ⇒ **Options** - este o proprietate compusă, din altele de tip **bool**:

GoFixedVertLine - dacă reține **false** nu sunt desenate linile verticale care separă celulele care aparțin linioarelor care nu participă la defilare.

GoFixedHorzLine - dacă reține **false** nu sunt desenate linile orizontale care separă celulele care aparțin linioarelor care nu participă la defilare.

GoVertLine - dacă reține **false** nu sunt desenate linile verticale care separă celulele care participă la defilare.

GoHorzLine - dacă reține **false** nu sunt desenate linile orizontale care separă celulele care participă la defilare.

GoEditing - dacă reține **true** utilizatorul va putea scrie în celule.

goDrawFocusSelected - dacă reține **true**, culoarea de fond a celulei selectate este diferită. Astfel, celula selectată este în evidență.

goRowSizing - dacă reține **true** linile care separă celulele pot fi mutate cu mouse-ul (se acționează asupra lor în dreptul celulelor care nu participă la defilare).

goColSizing - la fel pentru coloane.

goAlwaysShowEditor - dacă reține **true** și **goEditing** reține **true** la selecția unei celule apare imediat cursorul de editare.

- Programul următor afisează o matrice, citită dintr-un fisier text. Prima linie a matricei reține numărul de linii (**m**) și numărul de coloane (**n**). Următoarele **m** linii ale fisierului contin, fiecare, câte o linie a matricei.

4	6				
1	2	3	4	5	6
7	8	9	10	11	12
12	18	45	76	98	9
190	67	978	56	97	31

Fisierul text este creat cu ajutorul editorului **Notepad** și salvat sub numele dorit.

Îată cum arată formularul după apăsarea butonului care comandă citirea fisierului și afisarea lui:

Linie	Coloana	Coloana	Coloana	Coloana	Coloana
1	2	3	4		
2	7	8	9	10	
3	12	18	45	76	

```

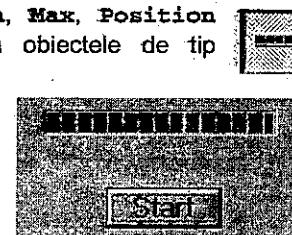
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
ifstream f("c:\\mat.in",ios::in);
int m,n,i,j,v;
f>>m>>n;
StringGrid1->RowCount=m+1;
StringGrid1->ColCount=n+1;
for (i=1;i<=m;i++)
    for (j=1; j<=n;j++)
    {
        f>>v;
        StringGrid1->Cells[j][i]=IntToStr(v);
    }
for (i=1;i<=m;i++)
    StringGrid1->Cells[0][i] = "Linia "+IntToStr(i);
for (j=1;j<=n;j++)
    StringGrid1->Cells[j][0] = "Col "+IntToStr(j);
}

```

- ✓ În acest fel avem posibilitatea să afișăm matrice mari, iar sarcina utilizatorului este de a vizualiza elementul dorit cu ajutorul barelor de defilare.
- ✓ Componentelor de acest tip li se asociază automat bare de defilare.

7.4.7 Componenta de tip TProgressBar

Componentele de acest tip au proprietățile **Min**, **Max**, **Position** (de tip **int**) având aceeași semnificație ca la obiectele de tip **TScrollBar**.



Să testăm modul de funcționare a barelor de progres. Vom ataşa formularului o bară de progres și un buton, aşa cum se vede în figura alăturată.

Pentru buton, la evenimentul **OnClick** "răspunde" următoarea funcție:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int i,j;
    for (i=0; i<=30000;i++)
    {
        Progressbar1->Position=i;
        for (j=0;j<=30000;j++);
    }
}

```

În fapt, sunt două cicluri imbricate. Ele n-au alt rost decât acela de a pun programul să efectueze o operatie care consumă timp. Evolutia acestui calcul este vizualizată cu ajutorul barei de progres. Pentru aceasta trebuie ca **Min** să fie 0, și **Max** să fie 30000.

- ✓ Rolul acestor bare este acela de a evidenția vizual timpul scurs și cel rămas unui proces în executare.

7.4.8 Meniuri

Meniurile sunt de două feluri: de tip **TMainMenu** și meniuri de tip **TPopupMenu**.

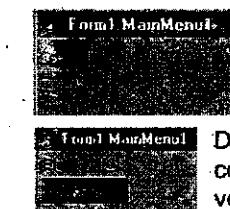


În stânga aveți un meniu de tip **TMainMenu**, cu trei niveluri. El se caracterizează prin faptul că opțiunile primului nivel se află în partea superioară a formularului (în top), așezate orizontal.

Iată care este componenta care se pune pe formular, indiferent de poziția pe care o ocupă în cadrul acestuia:

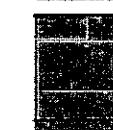


- ✓ O astfel de componentă este nonvizuală. Ea nu se vede pe formular în timpul executării, nu are importanță poziția ei, dar cu ajutorul ei se poate genera meniu!
- Un dublu click asupra componentei sau un click asupra proprietății **Items** (afisată de inspectorul de obiecte) are ca efect lansarea editorului de meniuri, așa cum se vede mai jos:



Opțiunea propriu-zisă se scrie în dreptul proprietății **Caption**, afisată de inspectorul de obiecte. În exemplu, scriem **Grup1**.

După scrierea opțiunii, avem posibilitatea să continuăm în același mod, fie pe orizontală, fie pe verticală.



Acest efect se obține prin introducerea unei opțiuni (în exemplu după **opt2**) pentru care câmpul **Caption** reține caracterul '-'.

- Uneori, este necesar ca anumite opțiuni să fie separate printr-o linie orizontală -numită și break orizontal, așa cum se vede mai jos:
- Orice opțiune din meniu este obiect, de tipul **TMenuItem**. La rândul lui, acest tip are proprietăți și metode.
- ⇒ Proprietatea **Break**. Poate reține una din următoarele trei valori, care pot fi selecționate dintr-o listă:
 - **MbNone** – opțiunea respectivă este scrisă pe aceeași coloană, în continuare.

- **MbBreak** – opțiunea respectivă este scrisă pe o nouă coloană a meniului.



În exemplul alăturat, proprietatea **Break** a opțiunii **opt2** reține **MbBreak**.

- **MbBarBreak** – opțiunea respectivă este scrisă pe o nouă coloană a meniului, dar noua coloană este separată printr-o bară verticală.

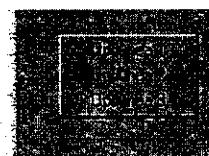


Am preluat exemplul anterior, dar proprietatea **Break** pentru **opt3** reține **mbBarBreak**.

- ⇒ Proprietatea **RadioItem** este de tip **bool**. Dacă reține **true**, poate fi combinată cu proprietatea **Checked** pentru a determina ca opțiunile afiate pe un anumit nivel să aibă comportamentul unor butoane radio.
- ⇒ Proprietatea **Checked**, este de tip **bool**. Dacă reține **true**, atunci pe lângă numele opțiunii este afisat un caracter 'v', dacă opțiunea nu este de tip radio, altfel este afisat un punct, similar cu marcarea unui buton de tip radio.
- ⇒ Proprietatea **Default** este de tip **bool**. Dacă reține **true**, opțiunea va fi scrisă îngroșat. Motivul este dat de faptul că acea opțiune este considerată implicită. Numai o singură opțiune, dintre cele afiate pe un anumit nivel, poate fi implicită.
- ⇒ Proprietatea **Enabled**, de tip **bool**, reține dacă opțiunea este activă sau nu. În cazul în care nu este activă, are o formă grafică specială de afişare, forma pe care o putem întâlni la mai toate obiectele și pe care am prezentat-o pentru butoane.
- ⇒ Proprietatea **ShortCut** permite ca opțiunea respectivă să poată fi apelată rapid, cu ajutorul tastaturii, printr-o combinație de taste. Exemplu: **Alt+x**. Programatorul va introduce combinația dorită, cu ajutorul inspectorului de obiecte, în căsuța rezervată proprietății.

Orice comandă dată printr-o opțiune a unui meniu se face prin scrierea unei funcții care răspunde evenimentului **OnClick**.

Meniuri de tip **TPopupMenu**



Astfel de meniuri nu apar decât dacă utilizatorul dorește acest lucru. Pentru aceasta utilizatorul trebuie să apese butonul drept al mouse-ului. În stânga, vedeti cum arată un astfel de meniu. Ele se mai numesc și meniuri flotante!



Componența selectată se pune pe formular. Locul în care este așezată nu prezintă importanță.

Pentru ca acest meniu să apară la solicitarea programatorului, este necesar ca proprietatea **PopupMenu** a formularului să rețină numele componentei (de exemplu **PopupMenu1**, adică numele implicit). În rest, cu astfel de meniuri se lucrează la fel cum se lucrează cu cele de tip **TMainMenu**.

Probleme propuse

1. Memorati în **Clipboard** imaginea care înfățișează **Desktop-ul** de pe calculatorul dv. (se apasă **Print Screen**). Prin utilizarea **Paint-ului** creați un fisier cu această imagine (se utilizează opțiunea **Paste**). Atașați formularului o componentă **TImage** care reține imaginea anterior memorată ca fisier.
2. Vizualizați întreaga imagine cu ajutorul barelor de scroll.
3. Creați un album cu diverse imagini. Selectia imaginii afisate se va face prin intermediul unei liste. Se cere ca în interiorul listei opțiunile să fie afisate prin nume atașate imaginilor.
4. Cu ajutorul unei bare de scroll pentru care **Min=0** și **Max=2000**, selectați un număr. Creați un fisier text care conține 400 de numere naturale, pe o singură linie și separate printr-un spațiu, între 1 și valoarea selectată.
5. Presupunem că fisierul creat la problema anterioară reprezintă o matrice patrată cu 40 de linii și 10 coloane memorată pe coloane (mai întâi prima coloana, apoi a doua etc). Vizualizați pe formular matricea memorată din fisier.
6. Prin utilizarea formularului de la problema precedentă, la care puteți adăuga componentele necesare, scrieți o funcție care inversează două linii ale matricelui. Indicii linilor se citesc de la tastatură.
7. Salvați matricea astfel obținută într-un fisier text, care va conține matricea memorată pe linii.

7.5 Cutii de dialog

7.5.1 Cum obținem o cutie de dialog

Formularul principal poate fi înzestrat cu un meniu, sau butoane, care să permită afișarea altor formular. Acestea din urmă pot fi sub forma unor cutii de dialog. Cine nu le-a folosit? Prin intermediul lor se pot introduce anumite informații strict necesare. Sunt utile prin faptul că obiectele prin care se poartă dialogul cu utilizatorul nu încap întotdeauna pe suprafața formularului principal. Etapele construirii unei cutii de dialog sunt:

- ⇒ Se atașează aplicației un nou formular. Acesta are numele implicit: `formX`, unde `X` poate fi `2,3...` în funcție de numărul cutiei de dialog. Pentru aceasta, din cadrul meniului principal al mediului `Builder` se apelează opțiunea `File | New Form`; Din acest moment aplicația afișează noul formular. El arată exact ca și formularul principal.
- ⇒ Proprietatea `Border Style` la noului formular trebuie să rețină valoarea `bsDialog`. Aceasta determină ca, în timpul executării, formularul să aibă câteva proprietăți speciale cum ar fi: imposibilitatea modificării dimensiunilor cu mouse-ul, prezența doar a butonului de închidere (acest `x`). De reținut: nu este obligatoriu ca `Border Style` să rețină acea valoare, dar așa este ușual. Utilizatorul nu se așteaptă să poată modifica dimensiunile cutiei de dialog (nici n-ar avea rost).
- ⇒ Formularului i se stabilesc noile dimensiuni (cu mouse-ul), apoi i se atașează obiectele prin care utilizatorul comunică cu aplicația. De asemenea se stabilește sirul afișat -proprietatea `Caption`, culoarea de fond -proprietatea `Color`, fonturile folosite și culoarea lor -`Font`.
- ⇒ Urmează "legarea" cutiei de aplicație, operatie specifică modului în care cutia va fi afișată.

7.5.2 Cutii de dialog afișate modal

- Afișarea modală se caracterizează prin:
 - Odată afișată, până la închiderea lor, nu mai este permis să efectuăm nici o operatie cu obiectele formularului principal. Singurele operații permise sunt doar asupra obiectelor afișate de cutii.
 - Afișarea unei astfel de cutii se face prin utilizarea metodei de tip funcție `ShowModal()`, metodă a obiectelor de tip `TForm` (tipul obiect al formularelor). Funcția întoarce o anumită valoare care comunică modul în care utilizatorul a închis cutia de dialog. În funcție de această valoare se poate lua o decizie în ce privește acțiunile următoare. Forma generală este:

```
int ShowModal();
```

- Atât timp cât utilizatorul n-a închis cutia de dialog, nu se execută instrucțiunea următoare de după `ShowModal()`.
- ⇒ Să se scrie o aplicație a cărei formular principal conține un buton. La apăsarea acestuia va apărea o cutie de dialog afișată modal. Cutia conține două butoane: `Ok` și `Cancel`. Oricare dintre ele închide cutia. După închiderea cutiei, se va afișa modul în care aceasta a fost închisă (cu `Ok` sau `Cancel`).

Pasul 1.



Se atașează formularului principal un buton pe care vom scrie `Afiseaza`.

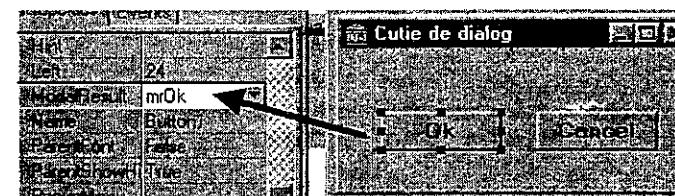
Pasul 2.



Se construiește cutia de dialog. Se apelează la `File | New Form`. Dupa apariția formularului se completează `Caption` cu titlul alăturat și se atașează cele două butoane.

Se scrie în `Unit1.h`: `#include "Unit2.cpp"`

Pasul 3. Pentru fiecare buton se completează proprietatea `ModalResult` (rezultatul afișării modale). Pentru `Ok` proprietatea va reține `mrOk`, iar pentru `Cancel` `mrCancel`. Aceasta are dublu efect:



- La apăsarea oricărui buton se închide cutia de dialog;
- Se poate testa modul în care utilizatorul a închis cutia.

De asemenea, pentru cutia de dialog, proprietatea `BorderStyle` va reține `bsDialog`.

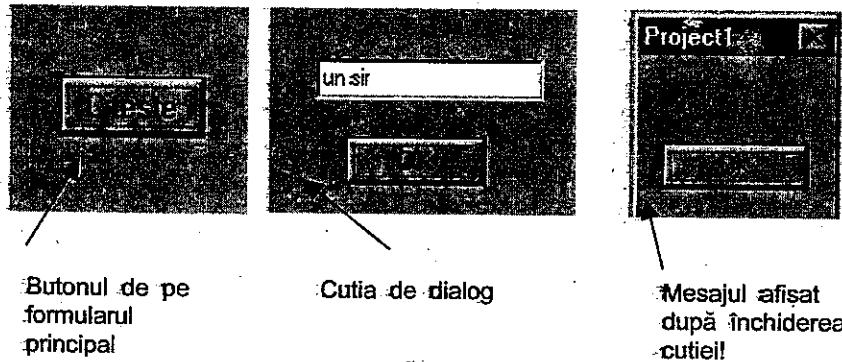
Pasul 4. La apăsarea butonului de formularul principal se lansează funcția următoare:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{ Form2->ShowModal();
  if (Form2->ModalResult==mrOk)
    ShowMessage ("Ok");
  else ShowMessage ("Cancel");
}

```

- ✓ Cutia de dialog poate conține orice componentă dintre cele învățate!
- ✓ Cutia de dialog este, așa cum am văzut un formular.
- ✓ Vizualizarea uneia sau alteia dintre unitățile de program se face cu **View Units**.
- ✓ Vizualizarea unuia sau altuia dintre formulare se face cu **View Forms**.
- Cutia de dialog, lansată cu ajutorul programului principal, conține o componentă de tip **TEdit** în care utilizatorul va introduce un sir. După închiderea cutiei se va afișa sirul introdus.



Butonul aflat pe formularul principal are în spate funcția:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{ Form2->ShowModal();
  if (Form2->ModalResult==mrOk)
    ShowMessage (Form2->Edit1->Text);
}

```

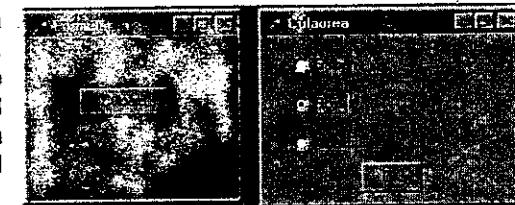
- ✓ Adresarea unei componente aflate pe alt formular se face prin a pune înaintea numelui ei, numele formularului pe care se află: Exemplu: **Form2->Edit1->Text**

7.5.3 Afisarea nemodală

- Afisarea nemodală se caracterizează prin:
- în timpul afișării, se poate acționa și asupra butoanelor, meniurilor etc. ale formularului principal;

- după afișare, nu se așteaptă închiderea cutiei, ci se rulează instrucțiunile următoare.
- Activarea unei astfel de cutii se realizează prin funcția numită **Show()**, iar închiderea prin **Close()**. Depistarea modului în care a fost închisă fereastra se poate face ca la afișarea modală (cu ajutorul proprietății **ModalResult**) deosebitre fiind dată de faptul că în spatele butonului trebuie să fie o funcție care răspunde evenimentului **onClick** și care conține **Close()**;
- închiderea formularului principal are ca efect și închiderea cutiei de dialog, dar închiderea cutiei de dialog nu afectează formularul principal;
- pentru ca să putem adresa **Form2** în **Unit1.cpp** se include **Unit2.h** (cu **#include "Unit2.h"**);
- pentru ca să putem adresa **Form1** în **Unit2.cpp** se include **Unit1.h** (cu **#include "Unit1.h"**);
- ✓ Cutia de dialog afișată nemodal are rol de cutie de asistență. Prin componentele sale se poate acționa asupra formularului principal.

- Programul următor, la apăsarea unui buton, afisează nemodal o cutie de dialog. Prin intermediul ei se schimbă culoarea formularului principal, initial galbenă.



La apăsarea butonului **Culoare** răspunde procedura:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{ Form2->Show(); }

```

Cutia de dialog prezintă 3 butoane radio, care au rolul de a stabili culoarea formularului principal. Culoarea initială a formularului principal este galben, iar butonul radio corespunzător este marcat (proprietatea **Checked** este **true**).

Pentru fiecare buton radio răspunde căte o funcție la evenimentul **OnClick**, care are rolul de schimbare a culorii formularului principal:

```

void __fastcall TForm2::RadioButton1Click(TObject *Sender)
{ if (RadioButton1->Checked) Form1->Color=c1Yellow; }

```

Butonul aflat pe cutie are rolul de a o închide:

```

void __fastcall TForm2::Button1Click(TObject *Sender)
{ Form2->Close(); }

```

7.5.4 Cutii de dialog predefinite

Pentru atașarea acestor cutii aplicaților, se utilizează pagina **Dialogs** a paletelor de componente.



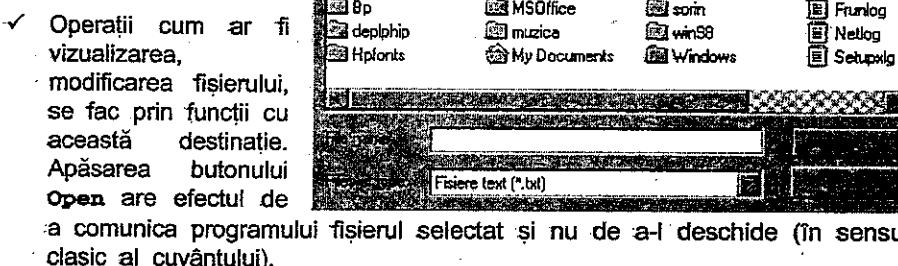
Majoritatea cutiilor de dialog prezente aici sunt de mare complexitate! Ele au metode și proprietăți proprii.

- **Important!** Pentru utilizarea cutiilor de dialog se pun pe formular iconurile corespunzătoare. Simpla așezare a acestora n-are nici un efect, dacă nu se scriu funcțiile corespunzătoare care le vizualizează (adică în timpul executării sunt învizibile). Nici locul în care se asează (de exemplu, peste altă componentă) n-are nici o importanță! Din acest motiv, despre astfel de componente se zice că sunt nonvizuale!

Cutia de tip **TOpenDialog**

- Este folosită pentru selectarea unui fișier în vederea deschiderii.

Cu ajutorul acestei cutii de dialog se selectează doar numele fișierului, nu se actionează asupra sa.

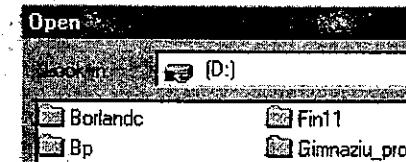


Pentru a utiliza așezăm pe formular icon-ul alăturat.



- Lansarea în executare a acestei cutii se face prin metoda de tip funcție **bool** numită **Execute()**. Ea returnează **true**, dacă a fost selectat fișierul. Principalele proprietăți ale unei astfel de cutii sunt:

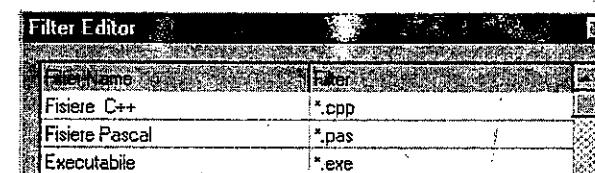
- ⇒ **FileName** – conține numele fișierului selectat de utilizator.
- ⇒ **InitialDir** – lansarea în executare a acestei cutii se face prin vizualizarea componentelor unui anumit folder. Proprietatea reține numele (și calea) folder-ului inițial afișat. De exemplu, cutia de mai sus, vizualizează, inițial, componentele folder-ului rădăcină pe D:. Pentru aceasta, am atribuit proprietății respective, cu ajutorul inspectorului de obiecte, valoarea initială D:\;



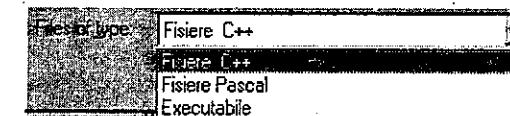
Cutia afisează rădăcina de pe D:\

- ✓ În cazul în care proprietatea respectivă conține sirul vid, cutia vizualizează continutul folder-ului **My Documents**.

- ⇒ **Filter** – nu întotdeauna utilizatorul dorește vizualizarea tuturor fișierelor, ci numai a celor cu o anumită extensie (extensiil). Extensiile fișierelor care vor fi vizualizate sunt reținute de această proprietate. Pentru a le introduce se execută click asupra celor trei puncte afișate de inspectorul de obiecte în dreptul proprietății respective. Astfel se lansează editorul de filtre. Se pot introduce mai multe filtre. Utilizatorul selectează din listă **Files of type** filtrul dorit. Editorul de filtre prezintă două coloane. În prima, se trec anumite informații auxiliare (numele în clar al fișierelor și extensia lor) iar în a doua coloană se trec extensiile efective. Pentru fiecare extensie, lista **Files of type** afisează doar continutul primei coloane. În cazul în care pentru un anumit filtru (o opțiune a listei) se dorește vizualizarea mai multor tipuri de fișiere, pentru ambele coloane opțiunile sunt separate prin punct și virgulă ','. Inițial cutia afisează doar fișierele de tipul cerut pe prima linie a tabelului (vezi cutia prezentată anterior).



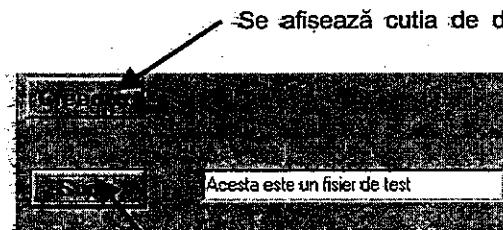
Editorul de filtre.



Așa selectează utilizatorul filtrul.

⇒ **FilterIndex** – La un moment dat, cutia de dialog afisează doar un anumit filtru (o linie a editorului de filtre), selectat de utilizator din cadrul listei, implicit, lista afisează initial primul filtru (prima linie a editorului de filtre). Dacă se dorește ca aceasta să afiseze initial un alt filtru se utilizează această proprietate. Pentru prima linie a editorului **FilterIndex** trebuie să conțină 0 sau 1, pentru a doua linie 2, și.m.d.

□ Să se creeze un fisier text cu o singură linie. Numele fișierului va fi introdus cu ajutorul unei cutii de dialog de tip **TOpenDialog** (în acest fel se poate selecta calea automat). Sirul conținut de fisier va fi introdus cu ajutorul unei componente de tip **TEdit**.



Se afisează cutia de dialog la apăsare!

Se scrie fișierul la apăsare!

Pentru a realiza programul, procedăm astfel:

Pasul 1. Pentru a putea lucra cu fisiere includem **fstream.h**:

```
#include "Unit1.h"
#include <fstream.h>
```

Pasul 2. Declarăm o variabilă globală de tipul **AnsiString**, în scopul memorării numelui fișierului:

```
TForm1 *Form1;
AnsiString Nume;
```

Pasul 3. Se așează pe formular componentele necesare.

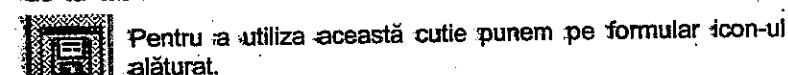
Pasul 4. La apăsarea butonului **Creeaza** se execută funcția următoare, care permite deschiderea cutiei de dialog și memorarea numelui fișierului (inclusiv calea) în variabila globală **Nume**:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ if(OpenDialog1->Execute())
  Nume=OpenDialog1->FileName;
}
```

Pasul 5. La apăsarea butonului **Scrie** se creează fișierul cu textul introdus de utilizator cu ajutorul componentei de tip **TEdit**:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ fstream f(Nume.c_str(),ios::out);
  f<<Edit1->Text;
  f.close();
}
```

Cutia de tip TSaveDialog. Cutia este utilă pentru "a naviga" printre folder-e pentru a găsi locul unde să salvăm un fișier sub un nume introdus de la tastatură.



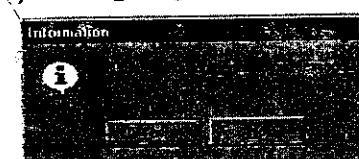
Pentru a utiliza această cutie punem pe formular icon-ul alăturat.

Intrucât programarea acestei cutii este identică cu programarea cutiei precedente, vom renunța la prezentarea ei.

Imaginiile (fișierele de tip bitmap) pot fi și ele încărcate / salvate.

Cutii de dialog în serie. Astfel de cutii se folosesc uzuale. Unele dintre ele pot conține o mică imagine (bmp). Se apelează ușor, prin utilizarea unor funcții. Până în prezent am folosit numai **ShowMessage()**. Datorită faptului că o secvență a programului poate apela mai multe astfel de funcții, una după alta, se mai numesc și cutii de dialog în serie.

1) MessageDlg()



Afisează în centrul ecranului o cutie de dialog, ca cea alăturată, în care utilizatorul răspunde prin apăsarea unor butoane

Functia are forma următoare:

```
word MessageDlg(AnsiString Msg, TMsgDlgType AType,
  TMsgDlgButtons AButtons,int HelpCtx);
```

- **Msg** – sirul de caractere care urmează a fi tipărit;
- **AType** – imaginea care este conținută de cutie. Ea poate lua una din valorile afisate de figura următoare -în care se vede și imaginea care apare sau **mtCustom**, caz în care cutia nu va conține nici o imagine.



mtConfirmation



mtInformation



mtWarning



mtError

- **AButtons** - butoanele care sunt continute de cutie. Tipul precizat este, de fapt, un tip multime de butoane, unde fiecare buton este reprezentat printr-o constantă. Constantele sunt prezentate mai jos:

```
mbYes mbNo, mbOK, mbCancel, mbHelp, mbAbort, mbRetry,
mbIgnore, mbAll, mbYesNoCancel, mbOkCancel,
mbAbortRetryIgnore
```

Acet parametru se trece ca mai jos:

```
TMsgDlgButtons() << mbYes << mbNo << mbCancel
```

- ✓ Este un tip multime (Set), tip care va fi prezentat în această carte.
- Pe ultimul parametru nu îl prezint. Vom trece pentru el întotdeauna valoarea 0.

Functia returnează una din valorile următoare, în funcție de care programul execută o secvență sau altă: mrNone, mrAbort, mrYes, mrOk, mrRetry, mrNo, mrCancel, mrIgnore, mrAll.

În funcție de numele constantei returnate, de tip word, se deduce cu ușurință care buton a fost apăsat, ca în exemplul următor:

```
if (MessageDlg("O cutie cu MessageDlg", mtConfirmation,
    TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes) ...
```

- 2) **MessageDlgPos()** - afisează o cutie la fel cu cea anterior prezentată. Diferența este dată de faptul că afisarea nu se face în centrul ecranului ci într-un loc dorit de programator. Ultimii doi parametri **x** și **y** reprezintă coordonatele colțului din stânga sus al cutiei. **x** are semnificația de coloană și **y** de linie. Valorile maxime ale lui **x** și **y** depind de rezoluția ecranului.

```
word MessageDlgPos(AnsiString Msg, TMsgDlgType AType,
    TMsgDlgButtons AButtons, int HelpCtx, int X, int Y);
```

3) InputBox()



O cutie de acest gen se poate vedea alăturat. Ea are rolul de a permite introducerea unui sir de caractere.

Functia este: **AnsiString InputBox(AnsiString ACaption, AnsiString APrompt, AnsiString ADefault)** unde:

- **ACaption** - este sirul care va constitui titlul ferestrei;

- **APrompt** - un sir care este afisat de cutie și are rolul de a întări utilizatorul asupra semnificației sirului care trebuie introdus.
- **ADefault** - conține sirul care este afisat inițial în obiectul de editare.

Functia returnează sirul introdus de utilizator. După apăsarea butonului **OK** va afisa sirul introdus de utilizator. În cazul în care se apasă **Cancel**, functia returnează sirul afisat inițial: "un text". Exemplu:

```
AnsiString x="Un text";
x=InputBox("Titlul cutiei", "Introduceti x",x);
ShowMessage(x);
```

- 4) **InputQuery** - arată la fel ca cea anterior prezentată și are aceiasi parametri. Deosebirile sunt date de faptul că returnează o valoare bool (true ,dacă a fost "apăsat", OK- sau false, dacă a fost "apăsat" Cancel), iar parametrul **ADefault** este transmis prin referință. Prin urmare, sirul introdus va fi reținut de această variabilă. Exemplu:

```
AnsiString x="Un text";
InputQuery("Titlul cutiei", "Introduceti x",x);
ShowMessage(x);
```

Probleme propuse

1. Scripti un program în care:

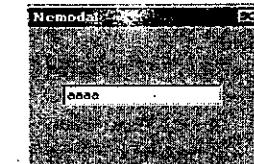
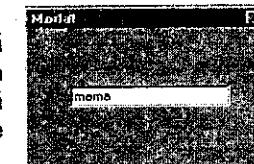
- formularul principal conține mai multe butoane radio cu opțiunile: **Adunare, Scadere, Înmulțire, Impartire**;
- după efectuarea selecției, de fiecare dată apare o cutie de dialog, afisată modal, cu ajutorul căreia se introduc cele două valori reale. La apăsarea butonului **Ok** (prezent și el pe cutie) se efectuează operația selectată.

Acet exercițiu urmărește ca dv. să sesizati diferența esențială între afisarea modală și cea nemodală a unei cutii de dialog.

- a) Formularul principal va conține un buton cu rolul de a afisa modal o cutie de dialog și un altul care afisează nemodal o altă cutie de dialog.



- b) Fiecare dintre cele două cutii de dialog afisate va conține câte o componentă de tip **TLabel**, așa cum se vede:



- c) Funcțiile care răspund apăsării celor două butoane vor afisa valorile reținute de componente de tip **TLabel**. Ce observați?

7.6 Introducerea textelor

7.6.1 Tipul Set

Tipul **Set** este deosebit de util pentru ca poate lucra ușor și elegant cu multimi. Introducerea acestui tip a fost inspirată din limbajul **Pascal**.

- ✓ O variabilă de tip **Set** este, de fapt, o variabilă de tip multime (retine o multime cu până la 255 elemente).
- ✓ Evident, acest tip a fost realizat cu ajutorul **OOP**. Operatiile permise asupra variabilelor de acest tip (reuniune, intersecție, diferență) sunt realizate prin supraîncărcarea operatorilor.
- > Forma generală prin care se declară acest tip este:

Set<type, minim, maxim> unde:

- **type** - reprezintă tipul elementelor care alcătuiesc multimea. El poate fi: **int, enum, char**;
- **minim** - reprezintă cel mai mic element din multime.
- **maxim** - reprezintă cel mai mare element din multime.
- ✓ **minim** trebuie să fie mare sau egal cu 0, **maxim** mai mic sau egal cu 255. Pentru a putea realiza mai multe declarații de variabile de acest tip, în locuri diferite în cadrul textului sursă, se utilizează **typedef**.

Exemple:

- 1) **typedef Set<int, 0,255> Multime;**
Multime **A,B**; **A** și **B** sunt variabile care pot reține multimi de numere naturale mai mari sau egale cu 0, mai mici sau egale cu 255.
- 2) **typedef Set <char, 'a','z'> Litere_Mici;**
Litere_Mici Z,T; **Z** și **T** sunt variabile care pot reține multimi de caractere de la 'a' la 'z', adică litere mici.
- ✓ Evident, literele au codurile **ASCII** mai mari ca 0 și mai mici ca 255.

În continuare prezentăm câteva metode ale acestui tip:

- ⇒ **Clear()** - atribuie variabilei multimea vidă.
- Exemplu: **A.Clear();** Variabila **A** este initializată cu multimea vidă.
- ⇒ **Contains(elm)** este metodă de tip **bool** care testează apartenența elementului **elm** la multime.
- Exemplu: **if (B.Contains(i))** Se testează dacă conținutul variabilei de tip **int** numită **i** aparține sau nu multimii **B**.

⇒ Adăugarea unui element multimii se face printr-o metodă realizată prin supraîncărcarea operatorului **<<**.

Exemplu: **B<<10<<17;** Adăugăm multimi reținută de variabila **B** elementele 10 și 17.

⇒ Extragerea unui element dintr-o multime se face printr-o metodă realizată prin supraîncărcarea operatorului **>>**.

Exemplu: **B>>10>>17;** Se calculează **B-{10}**, apoi **B-{17}**, iar rezultatul este atribuit, de fiecare dată, variabilei **B**.

⇒ Reuniunea a două multimi se face cu ajutorul a două metode obținute prin supraîncărcarea operatorilor **+** și **+=**. Ultima metodă are ca efect și atribuirea rezultatului operandului aflat în stânga operatorului. Exemple:

a) **A=A+B** -se calculează reuniunea multimilor reținute de **A** și **B**, iar rezultatul este atribuit variabilei **A**.

b) **A+=B** -la fel ca mai sus.

⇒ Intersecția a două multimi se face cu ajutorul a două metode obținute prin supraîncărcarea operatorilor ***** și ***=**. Ultima metodă are ca efect și atribuirea rezultatului operandului aflat în stânga operatorului. Exemple:

c) **A=A*B** -se calculează intersecția multimilor reținute de **A** și **B**, iar rezultatul este atribuit variabilei **A**.

d) **A*=B** -la fel ca mai sus.

⇒ Diferența a două multimi se face cu ajutorul a două metode obținute prin supraîncărcarea operatorilor **-** și **-=**. Ultima metodă are ca efect și atribuirea rezultatului operandului aflat în stânga operatorului. Exemple:

e) **A=A-B** -se calculează intersecția multimilor reținute de **A** și **B**, iar rezultatul este atribuit variabilei **A**.

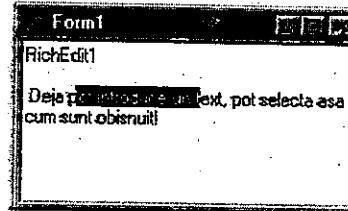
f) **A+=B** -la fel ca mai sus.

7.6.2 Notiuni introductive



Pentru introducerea textelor vom folosi o componentă de tip **TRichEdit**, aflată în pagina **Windows** a paletelor de componente.

Imediat ce am așezat pe formular componenta respectivă, apare o suprafață în care avem posibilitatea să introducem texte. Se recomandă ca respectiva componentă să fie aliniată - proprietatea **Align** va reține **AlClient** -(adică pe întreaga suprafață a formularului).



De acum putem scrie ce dorim, putem sterge sau putem selecta textele dorite, toate aşa cum suntem obişnuiţi. În schimb, nu putem salva ce am scris, nu putem utiliza mai multe fonturi, mai multe mărimi ale literelor etc.

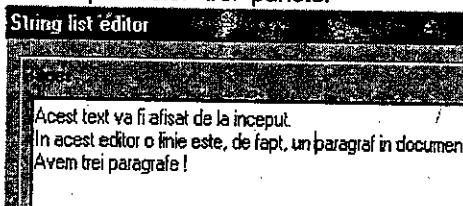
- Toate cele de mai sus devin posibile dacă studiem proprietăile și metodele componentei de tip **TRichEdit**.

7.6.3 Cum este reținut textul introdus

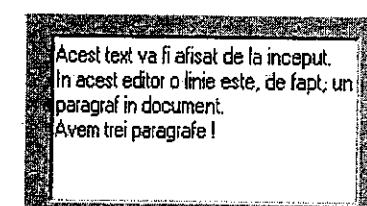
Documentul (întregul text introdus) este alcătuit din unul sau mai multe paragrafe. După cum știm, paragrafele sunt separate între ele prin **Enter**.

- Pentru a memora un paragraf este suficient să avem o variabilă de tipul **AnsiString**. Am învățat că în Borland C++ Builder tipul **AnsiString** poate reține un sir de orice lungime.
- Pentru a reține întreg documentul se foloseste un tip numit **TStrings**. Acest tip descrie, în liniile mari, un vector de siruri de caractere de tip **AnsiString**. Fiecare sir reține un paragraf, vectorul de siruri reține întreg documentul.
- Proprietatea **Lines** a componentei de tip **TRichEdit** este de tipul **TStrings** și reține întregul text introdus de utilizator.

Inspectorul de obiecte a atașat acestei proprietăți un editor de texte care permite să se afișeze de la bun început un text inițial sau să se steargă textul implicit afișat (de exemplu **RichEdit1**). Pentru a-l utiliza se execută click asupra celor trei puncte:



Așa scriu cu ajutorul editorului asociat!



Așa arată textul afișat de componentă.

Tipul **TStrings** este, la rândul lui, un tip obiect. Prin urmare, are proprietăți și metode proprii.

- Să se creeze un editor de texte care printr-un meniu să ofere posibilitatea de salvare sau încărcare a fișierelor text cu extensia **.txt**.

Așezăm pe formular doar o componentă de tip **TRichEdit**, una de tip **TMainMenu** (pentru meniu), una de tip **TOpenDialog** (pentru o cutie de selectie a fișierelor) și una de tip **TSaveDialog** (pentru fixarea folder-ului în care se va face salvarea). Componenta de tip **TRichEdit** va fi aliniată **alClient** (proprietatea **Align**).



Meniul va avea opțiunile **Save As** și **Open**. Nu contează locul în care se asează componentele nonvizuale!

Executarea unui click asupra opțiunii **Open** conduce la rularea funcției:

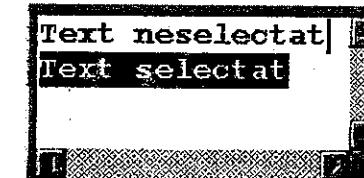
```
void __fastcall TForm1::Open1Click(TObject *Sender)
{ if (OpenDialog1->Execute())
    RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);
}
```

La executarea unui click asupra opțiunii **Save As** se execută funcția:

```
void __fastcall TForm1::SaveAs1Click(TObject *Sender)
{ if (SaveDialog1->Execute())
    RichEdit1->Lines->SaveToFile(SaveDialog1->FileName);
}
```

- Cu ajutorul programului astfel obținut introduceți un text, salvați-l cu un nume dorit (tastați și extensia) și apoi reîncărcați-l.

7.6.4 Despre selectări



Textele refiinute de obiectele de tip **TRichEdit** pot fi selectate, aşa cum se vede alăturat.

Componentele de tip **TRichEdit** au câteva proprietăți și metode care permit lucrul cu textul selectat:

- SelectLength** - reține numărul de caractere pe care le are textul selectat.
- SelectStart** - reține poziția în sir a cursorului de editare.
- SelectText** - reține textul selectat.

Prezentăm în continuare câteva metode, mai des folosite:

- ⇒ `void Clear(void);` - șterge textul reținut de obiect (toate șururile).
- ⇒ `void ClearSelection(void);` - șterge textul selectat
- ⇒ `void CopyToClipboard(void);` - copiază textul selectat în Clipboard (memorie folosită în Windows pentru transferul datelor între aplicații). Textul selectat rămâne în cadrul textului de bază.
- ⇒ `void CutToClipboard(void);` - la fel ca precedentă, cu deosebirea că textul selectat este sters din cadrul textului reținut de obiect.
- ⇒ `void PasteFromClipboard(void);` - inserează textul din Clipboard în cadrul obiectului, începând cu poziția curentă a cursorului.
- ⇒ `void SelectAll(void);` - selectează întreg textul din document.

□ Adăugați la aplicația precedentă submeniuul **Edit** cu opțiunile:

- **Select All** - selectează întreg textul introdus;
- **Del** - șterge textul selectat;
- **Copy** - copiază în Clipboard textul selectat, dar acesta rămâne în document;
- **Cut** - copiază în Clipboard textul selectat, iar acesta este sters din document;
- **Paste** - introduce din Clipboard textul reținut. Introducerea se face începând cu poziția curentă a editorului.



Se adaugă meniului opțiunile alăturate.

Pentru fiecare opțiune se scrie metoda care răspunde evenimentului **OnClick**, după cum urmează:

```
void __fastcall TForm1::SelectAllClick(TObject *Sender)
{ RichEdit1->SelectAll(); // Select All }

void __fastcall TForm1::DelClick(TObject *Sender)
{ RichEdit1->ClearSelection(); // Del }
```

7.6.5 Despre font-uri

⇒ Componențele de tip **TRichEdit** (și nu numai ele) au o proprietate, numită **Font**, de tipul **TFont** prin care se pot stabili font-urile care apar pe formular. Aceasta este un tip obiect, deci la rândul său are alte proprietăți care, de altfel caracterizează font-urile care apar în document:

■ **Name** - numele fontului, proprietatea de tip **AnsiString**. Exemple: **Courier New**, **Times New Roman** etc.

■ **Size** - mărimea, proprietate de tip **int**. Exemple: 8, 10, 11;

■ **Color** - culoare. Culoarea este identificabilă prin intermediul unor constante cum sunt: **clBlue**, **clWhite** etc.

■ **Style** - stilul. Vă să cum sănse definite stilurile în **Borland C++ Builder**:

- enum **TFontStyle** { **fsBold**, **fsItalic**, **fsUnderline**, **fsStrikeOut** };
- **TFontStyles** este multime cu elemente de tip **TFontStyle** (vezi tipul **Set**).
- property **Style: TFontStyles**; Unde:

fsBold - înseamnă bold (îngroșat);

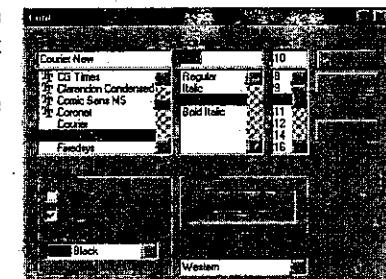
fsItalic - înseamnă italic (înclinat);

fsUnderline - subliniat;

fsStrikeOut - făiat de o linie.

După cum observați, stilurile sunt definite prin tipul enumerare. Proprietatea **Style** este de tip multime. O literă poate apărea, de exemplu, atât îngrosată -*Bold*-, cât și înclinată -*italic*. Aceasta înseamnă că are în același timp două stiluri. Din acest motiv stilul este de tip multime - ca să poată reține două valori sau mai multe valori în același timp.

Valorile initiale ale proprietății **Font** a componentei de tip **TRichEdit** se pot stabili cu ajutorul unei cutii de dialog, apelabilă din cadrul inspectorului de obiecte - click pe cele trei puncte:



Un text scris altfel

Exemplu: Scriu bold italic, cu font de 12.

Am văzut faptul că font-urile pot fi stabilite de la bun început cu ajutorul cutiei de dialog afisată de inspectorul de obiecte dacă se execută click asupra celor trei puncte. Font-urile pot fi modificate și dinamic, adică de către utilizator, în timpul executării programului.

⇒ Ideea de bază este de a modifica font-urile textului selectat.

- ⇒ În ipoteza în care nu avem text selectat, modificarea are efect începând cu poziția curentă a cursorului, pentru toate caracterele introduse în continuare.
- ⇒ Componentele de tip **TRichEdit** au o proprietate, care nu este afișată de inspectorul de obiecte și care permite cele prezentate mai sus: **SelAttributes**: **TTextAttributes**; -retine atributele textului selectat.

Ce semnificație are tipul **TTextAttributes**? Este un tip obiect -care are, la rândul lui câteva proprietăți, dintre care mai importante sunt:

- **Color** -retine culoarea fonturilor (**clBlack**, **clMaroon**, **clGreen**, ...)
- **Name** -numele fonturilor (Arial, Courier_new...);
- **Size** -mărimea fontului;
- **Style** -stilul de scriere, multime cu elemente în (**fsBold**, **fsItalic**, **fsUnderline**, **fsStrikeout**);

Această proprietate se va folosi pentru modificarea dinamică a font-urilor.

A. Modificarea cu ajutorul cutiei de dialog de tip **TFontDialog**.



Icon-ul acestei cutii se găsește în pagina **Dialogs** a paletelor de componente.

Cutia propriu-zisă este aceeași cu cea folosită la fixarea inițială a font-urilor. În rest, se folosește ca orice cutie de dialog. Așa că vom trece direct la aplicație!

- La aplicația precedentă dorim să se adauge submeniu **Format** cu opțiunea **Font**. Această opțiune va activa o cutie de dialog de tip **TFontDialog**, cu rolul de a modifica dinamic font-urile.



La executarea unui click asupra acestei opțiuni se rulează funcția următoare:

```
void __fastcall TForm1::Font1Click(TObject *Sender)
{ FontDialog1->Font->Assign(RichEdit1->SelAttributes);
  if (FontDialog1->Execute())
    RichEdit1->SelAttributes->Assign(FontDialog1->Font);
}
```

- ⇒ Se atribuie font-urilor afișate de cutie font-urile existente în document în poziția curentă a cursorului!
- ⇒ La sfârșit se atribuie textului selectat font-urile stabilite cu ajutorul cutiei!

- ✓ Pentru atribuire am utilizat metoda **Assign()**, existentă atât pentru **Font** cât și pentru **SelAttributes**. Metoda în sine nu va fi prezentată în această carte, înțelegerea ei necesită cunoștințe superioare. Scriu cu un stil **acum cu alt stil si acum cu altul** de OOP.

De acum textele pot arăta și asta:

B. Modificarea cu ajutorul barei de instrumente.

Bara de instrumente se întâlnește în multe aplicații. De această dată vom atașa programului nostru.

- ⇒ Pentru a obține bara de instrumente vom folosi o componentă de tip **TPanel** aliniată **alTop**. Pe aceasta vom așeza diverse butoane și o listă de selecție.
- Atașați aplicației anterioare o bară de instrumente prin care să se modifice font-urile, stilul lor (**Bold**, **Italic** ...) și mărimea.
- Selectăm o componentă de tip **TPanel** și o aliniem **alTop**. Iată cum arată:



Pe bară putem atașa orice componentă dintre cele învățate. În cazul nostru vom atașa butoane de tip **TSpeedButton** și o listă de selecție de tip **TComboBox**.

- Adăugăm următoarele 3 butoane pentru modificarea stilului font-urilor:



Pentru că cele trei butoane sunt independente, fiecare buton va face parte dintr-un grup separat (**GroupIndex**).

- Pentru că la o apăsare butonul trebuie să rămână apăsat, iar la următoarea să se ridice, pentru fiecare buton proprietatea **AllowAllUp** trebuie să rețină **true**.
- Proprietatea **Caption** a fiecărui buton reține litera specifică acestuia (B, I, U). Stilul de scriere a fost stabilit cu ajutorul cutiei de dialog care apare atunci când se execută click asupra celor trei puncte pe proprietatea **Font** a butonului.
- Pentru fiecare buton răspunde căte o funcție care schimbă stilul de scriere în componenta de tip **RichEdit1**. Ea ține cont dacă după click butonul este apăsat sau nu! Nu uitați că proprietatea care reține stilul este de tip **multime**! Exemplu:

```
void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
{ if (SpeedButton1->Down)
```

```

RichEdit1->SelAttributes->Style=;
RichEdit1->SelAttributes->Style<<fsBold;
else RichEdit1->SelAttributes->Style=;
RichEdit1->SelAttributes->Style>>fsBold;
}

```

În continuare vom atașa barei de comenzi două liste de tip **TComboBox** prin care putem modifica font-urile folosite și mărimea lor.

- Evenimentul **OnCreate** (la creare) pentru formular, îl vom atașa la funcție care încarcă listele cu font-urile existente în sistem și mărimele posibile ale lor, între 6 și 40 (am ales eu).

```

void __fastcall TForm1::FormCreate(TObject *Sender)
{
int i;
for (i=1;i<=Screen->Fonts->Count-1;i++)
  ComboBox1->Items->Add(Screen->Fonts->Strings[i]);
  ComboBox1->Text=RichEdit1->Font->Name;
for (i=6;i<=40;i++)
  ComboBox2->Items->Add(IntToStr(i));
  ComboBox2->Text=IntToStr(RichEdit1->Font->Size);
}

```

- ✓ Variabila **Screen** este globală și este de tipul **TScreen**. Ea conține numărul font-urilor existente (în Windows) și font-urile propriu-zise. Am utilizat-o fără o prezentare detaliată.
- ✓ În initial cele două liste vor afisa valorile curente reținute **RichEdit1** (stabilite cu proprietatea **Font** a acestei componente).
- Cele două liste răspund evenimentului **OnClick** prin funcțiile de mai jos, care atribuie componentei de tip **TRichEdit** font-ul și mărimea selectată, după care focus-ul este redat componentei de tip **TRichEdit**.

```

void __fastcall TForm1::ComboBox1Click(TObject *Sender)
{
RichEdit1->SelAttributes->Name=ComboBox1->Text;
RichEdit1->SetFocus();
}

void __fastcall TForm1::ComboBox2Click(TObject *Sender)
{
RichEdit1->SelAttributes->Size=StrToInt(ComboBox2->Text);
RichEdit1->SetFocus();
}

```

7.6.6 Despre paragrafe

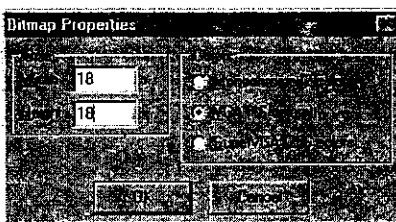
Am văzut că un document este format din mai multe paragrafe. Cunoaștem faptul că paragrafele pot fi și ele formate.



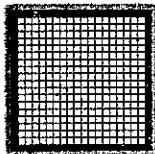
- ⇒ Obiectele de tip **TRichEdit** au proprietatea **Paragraph**, de tip **TParaAttributes**.
- ⇒ Proprietatea **Paragraph** reține anumite attribute ale unui paragraf și textului reținut. După cum știm din partea de utilizare, pentru ca textul introdus să contină un nou paragraf, se tastează **Enter**. Iată cum este definită în **Borland C++ Builder** această proprietate: **Paragraph: TParaAttributes**:

La rândul său, tipul **TParaAttributes** are următoarele proprietăți:

- **Alignment** - alinierarea paragrafului, poate lua valorile:
 - **taLeftJustify** -aliniat la stânga;
 - **taCenter** - aliniat în centru;
 - **taRightJustify** aliniat la dreapta.
- **FirstIdent** -specifică numărul de pixeli ce desparte prima linie a paragrafului de marginea stângă a documentului.
- **LeftIdent** -specifică în biți numărul de pixeli ce desparte liniile paragrafului de marginea stângă a documentului.
- **RightIdent** - specifică în biți numărul de pixeli ce desparte liniile paragrafului de marginea dreaptă a documentului.
- **Numbering** -poate lua valorile:
 - **nsBullet** -paragraful apare identat cu acele discuri (bulete);
 - **nsNone** -absența buletelor.
- ✓ Cu ajutorul acestei proprietăți paragrafele pot fi formatare de la început. Există și posibilitatea de a fi formatare dinamic, de către utilizator.
- Să se înzestreze bara de comenzi a aplicatiei anterioare cu trei butoane care să permită modificarea dinamică a alinierii paragrafelor.
- ⇒ Vom folosi trei butoane de tip **TSpeedButton**, asezate pe componenta de tip **TPanel**.
- Întrucât butoanele nu funcționează independent, apăsarea uneia determină ridicarea altuia, toate butoanele vor face parte dintr-un grup. Pentru fiecare dintre ele proprietatea **GroupIndex** va reține 4 (mai avem alte trei butoane).
- Pentru a construi imaginile asociate vom apela **Image Editor** -editor de imagini care aparține mediului **Borland C++ Builder** și este apelat din submeniul **Tools**.
- Cu ajutorul său vom crea o imagine bitmap de 19x19.



Cu ajutorul acestei cutii de dialog se stabilesc dimensiunile imaginii.



Imaginea este mărită până când putem să trasăm pe ea linile care o alcătuiesc. Mărirea se face din meniul editorului **View | Zoom In**, iar micșorarea cu **Zoom Out**.

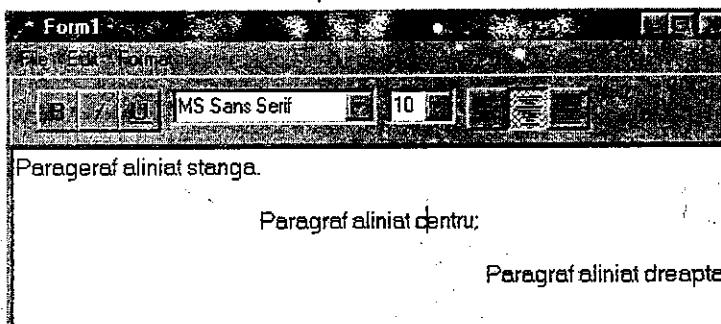
- ✓ Stabilirea culorilor de fond și de desen, utilizarea instrumentelor se face în **Image Editor** exact ca în **Paint**.
- Se încarcă în cele trei butoane imaginile create și pentru fiecare dintre ele va răspunde una din funcțiile de mai jos:

```
void __fastcall TForm1::SpeedButton4Click(TObject *Sender)
{ RichEdit1->Paragraph->Alignment=taLeftJustify;
}

void __fastcall TForm1::SpeedButton5Click(TObject *Sender)
{ RichEdit1->Paragraph->Alignment=taCenterJustify;
}

void __fastcall TForm1::SpeedButton6Click(TObject *Sender)
{RichEdit1->Paragraph->Alignment=taRightJustify;
}
```

Îată cum arată editorul de texte pe care tocmai l-am creat:



Bibliografie

4. Croitoru C. Tehnici de bază în optimizarea combinatorie - Editura Universității Al. I. Cuza Iași 1992.
5. Mateescu (Cerchez) E, Maxim I -Arbore, Editura Tara Fagilor, Suceava, 1997.
6. Knuth D. E. "Tratat de programarea calculatoarelor Algoritmi fundamentali" Editura Tehnică, 1974
7. Lica D, Onea E "Informatică" manual pentru clasa a IX-a Editura L&S Infomat 1999
8. Lica D, Onea E "Informatică" manual pentru clasa a X-a Editura ProGnosis 2000.
9. Livovschi L., Georgescu H., Sinteză și analiza algoritmilor, București 1986
10. Patrău B, Mișescu M. "Informatică" Editura Teora 1999
11. Pintea R, Oprescu D, Grigoriu D, Balanescu C, Voicu A, Cerchez E, Seban M "Teste de sinteză în programare", bacalaureat. Editura L&S Infomat 2000;
12. Pintea R, Voicu A, Voicu A, Informatică manual clasa a X-a Editura L&S Infomat 2000.
13. Pintea R, Atanasiu A, Culegere de probleme Pascal, Editura Petron 1997.
14. Rancea D și colectiv -"Informatică" Editura Computer Libris Agora, 1999
15. Tudor S, Informatică, manual clasa a IX-a Varianta C++, Editura L&S Infomat 2000;
16. Tudor S, Informatică, manual clasa a X-a, Varianta C++, Editura L&S Infomat 2000;
17. Tudor S, Initiere în Programarea Vizuală, Varianta Borland C++ Builder, Editura L&S Infomat 2000.
18. Leblanc Gérard - Borland C++ Builder 3 - EYROLLES, Paris, 1998;

CUPRINS

CAPITOLUL 1 ALOCAREA DINAMICĂ A MEMORIEI	5
1.1. Variabile de tip pointer.....	5
1.2. Alocarea dinamică a memoriei.....	8
1.3. Alocarea dinamică a masivelor	10
Probleme propuse	14
CAPITOLUL 2 LISTE LINIARE	20
2.1. Definiția listelor	20
2.2. Liste alocate simplu înlățuit	20
2.2.1. Prezentare generală	20
2.2.2. Crearea și afisarea listelor	21
2.2.3. Operatii asupra unei liste liniare	24
2.2.4. Aplicatii ale listelor liniare	29
2.2.4.1. Sortare prin inserție	29
2.2.4.2. Sortare topologică	32
2.3. Liste alocate dublu înlățuit	37
2.4. Siva alocată ca listă liniară simplu înlățuită	42
2.5. Coada alocată ca listă liniară simplu înlățuită	43
Probleme propuse	45
CAPITOLUL 3 ELEMENTE DE TEORIA GRAFURILOR	52
3.1. Grafuri orientate.....	52
3.1.1. Noțiuni introductive	52
3.1.2. Metode de reprezentare în memorie a unui graf orientat.....	54
3.1.3. Parcurserea grafurilor orientate	57
3.1.3.1. Parcurserea în fâșime	57
3.1.3.2. Parcurserea în adâncime	61
3.1.3.3. Estimarea timpului necesar parcurgerii grafurilor	62
3.1.4. Matricea drumurilor	62
3.1.5. Componențe tare conexe	64
3.1.6. Drumuri în grafuri	68
3.1.6.1. Matricea ponderilor	68
3.1.6.2. Algoritmul Roy-Floyd	70
3.1.6.3. Algoritmul lui Dijkstra	76
3.2. Grafuri neorientate.....	81
3.2.1. Noțiuni generale	81
3.2.2. Grafuri hamiltoniene	85
3.2.3. Grafuri euleriene	87
3.3. Retele de transport	93
3.3.1. Ce este o rețea de transport	93
3.3.2. Flux într-o rețea de transport	94

3.3.3. Determinarea fluxului de valoare maximă	95
Probleme propuse	102
CAPITOLUL 4 ARBORI SI ARBORESCENTE	113
4.1. Arbori	113
4.1.1. Noțiunea de arbore	113
4.1.2. Metode de memorare a arborilor	115
4.1.3. Arbore parțial de cost minim	121
4.2. Arborescente	124
4.2.1. Noțiunea de arborescentă	124
4.2.2. Arbori binari	128
4.2.2.1. Notiunea de arbore binar	128
4.2.2.2. Modalități de reprezentare a arborilor binari	129
4.2.2.3. Modalități de parcursere a arborilor binari	130
4.2.2.4. HeapSort	133
4.2.2.5. Arbori de căutare	138
4.2.2.6. Forma poloneză - aplicații	144
Probleme propuse	150
CAPITOLUL 5 ÎNITIERE ÎN PROGRAMAREA ORIENTATĂ PE OBIECTE	157
5.1. Introducere.....	157
5.2. Încapsularea	158
5.2.1. Definirea claselor	158
5.2.2. Constructori, destrutori	162
5.2.3. Supraîncărcarea operatorilor	166
5.2.4. Funcții prieten	173
5.2.5. Aplicații ale încapsulării	174
5.2.5.1. Multimi	174
5.2.5.2. Matrice rare	177
5.3. Moștenirea	181
5.3.1. Noțiuni de bază care privesc moștenirea	181
5.3.2. Accesul la membrii unei clase	185
5.3.3. Din nou despre constructori și destrutori	186
5.3.4. O aplicație a moștenirii	188
5.4. Polimorfism	190
5.4.1. O aplicație a polimorfismului	192
Probleme propuse	195
CAPITOLUL 6 PROIECTAREA APlicațIILOR	197
6.1. O problemă din practică	197
6.2. Analiza problemei	197
6.3. Concepția	198
Probleme propuse	199
ANEXA Modulul "Grafuri"	201

CAPITOLUL 7 ÎNITIERE ÎN PROGRAMAREA VIZUALĂ, PRINCIPALA APLICAȚIE A PROGRAMĂRII ORIENTATE PE OBIECTE	197
7.1. Primii pași în C++ Builder.....	204
7.2. Întriări / ieșiri elegante.....	217
7.3. Liste.....	237
7.4. Alte componente des folosite.....	247
7.5. Cutii de dialog.....	264
7.6. Introducerea textelor.....	274

BIBLIOGRAFIE	285
--------------	-----

Tiparul executat la
S.C. LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, București
Tel./Fax 211.32.60; Tel. 212.29.27

