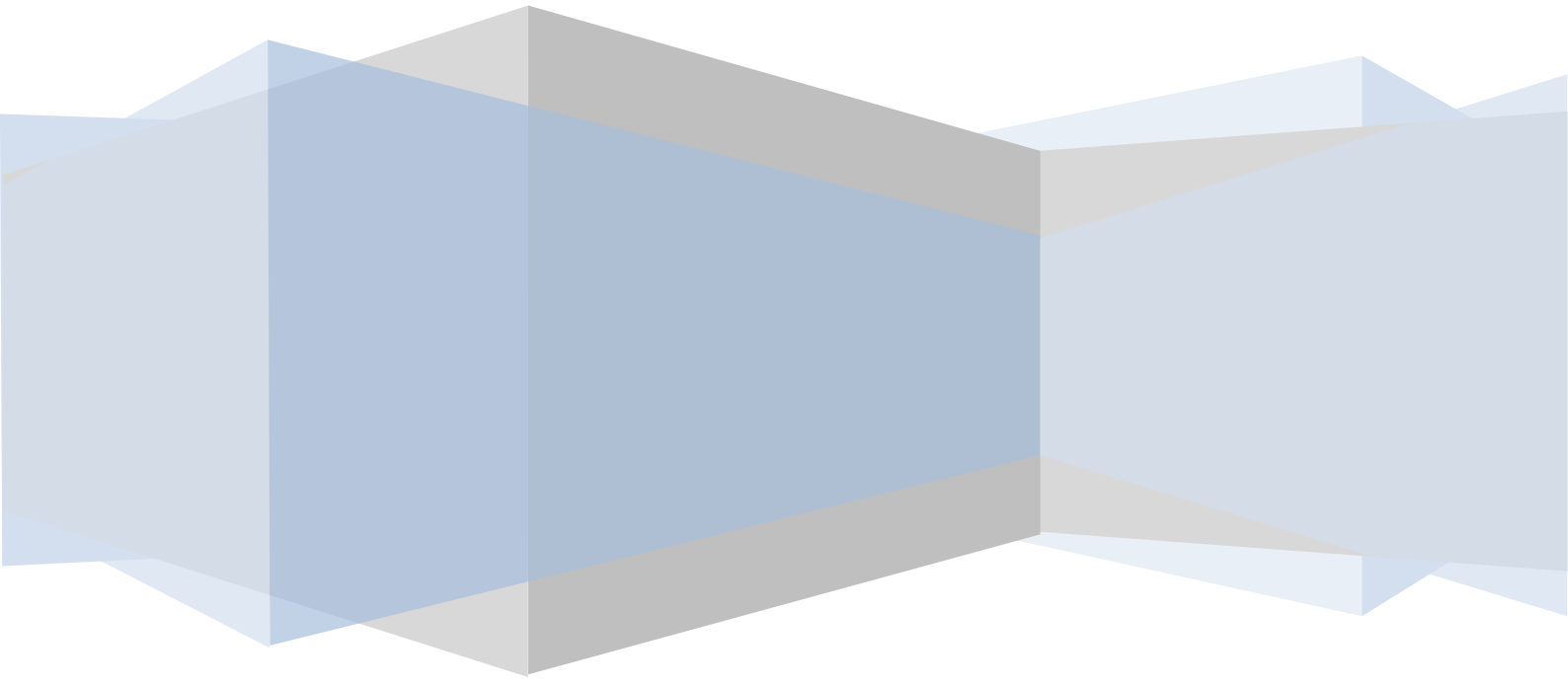
Universitatea Transilvania din Braşov
Facultatea de Inginerie Electrică şi Ştiinţa Calculatoarelor
Departamentul Automatică şi Tehnologia Informaţiei

Programarea Calculatoarelor şi Limbaje de Programare III

Îndrumar de laborator

prof. dr. ing. Sorin-Aurel MORARU
şef lucr. dr. ing. Dominic Mircea KRISTÁLY



BRAŞOV, 2012

Cuprins

Introducere în programarea calculatoarelor, algoritmi şi tehnici de programare

Majoritatea problemelor practice pe care le întâlnim în fiecare zi se pot rezolva efectuând un număr finit de operații într-o anumită ordine, tot timpul aceleaşi pentru aceiaşi problemă. Unele probleme (înrudite) acceptă aceleaşi tipuri de soluții, adică pot fi rezolvate urmând aceiaşi paşi. Prin pas se înțelege o operație sau un raționament.

Suita de paşi ce au ca scop rezolvarea unor probleme poartă numele de **algoritm**.

Noțiunea de algoritm este des întrebuințată atunci când vine vorba de scrierea programelor pentru calculator. Sistemele de calcul actuale sunt capabile doar să efectueze, foarte rapid, calcule matematice simple, fără a înțelege scopul sau sensul lor (sunt lipsite de inteligență); prin urmare nu sunt capabile să rezolve probleme prin raționamente proprii.

Pentru a putea rezolva o problemă, un program trebuie să implementeze, într-o formă înțeleasă de calculator, unul sau mai mulți algoritmi.

Scrierea de programe pentru calculator se realizează cu ajutorul *limbajelor de programare*. Acestea oferă programatorului o interfață de comunicație cu sistemul de calcul. Printre limbajele de programare utilizate pe scară largă la ora actuală pot fi menționate: C/C++, Java, C#, Visual Basic.

Scopul unui program de calculator este de a genera anumite rezultate pe baza unor date de intrare. Schematic, un program este asemenea unei cutii negre căreia i se dau niște date de intrare și generează la ieșire niște rezultate.

Fig. 1 – Schema de principiu a unui program

Pentru a uşura lucrul cu datele necesare unui program, toate limbajele de programare oferă mecanismul de *variabil*. O variabilă este, într-o definiție foarte simplă, o etichetă asociată unei adrese de memorie, deci, indirect, – unei locații de memorie.

Cea mai importantă caracteristică a unei variabile îl constituie *tipul de date* asociat.

Tipul de date specifică natura valorilor, modul de reprezentare a acestora şi plaja de valori ce se poate memora în zona de memorie asociată variabilei. Majoritatea limbajelor de programare oferă

un set redus de tipuri de date, numite de bază. Pe lângă acest set, limbajele de programare moderne oferă şi mecanisme pentru crearea de noi tipuri de date (tipuri de date *utilizator*), utilizând tipurile deja existente. Printre aceste mecanisme se remarcă enumerările, structurile și clasele.

Există şi limbaje de programare (cum ar fi *Microsoft Visual Basic*) care nu au decât un singur tip de date, numit generic tipul *variant*. Practic, tipul variabilelor sunt extrapolate din contextul în care apar sau sunt convertite explicit în momentul utilizării. Astfel, o variabilă poate fi tratată atât ca număr, cât şi ca şir de caractere.

Majoritatea algoritmilor necesită structuri de date specifice pentru a-și putea îndeplini scopul. Aceste structuri se obțin făcând apel la mecanismele de creare a tipurilor de date utilizator. Acest îndrumar va trata structurile de date în mod progresiv, pe măsură ce algoritmii prezentați o vor necesita.

Unii algoritmi fac apel la metode speciale de calcul/prelucrare, denumite **tehnici de programare**, majoritatea fiind împrumutate din matematică. Printre tehnicile ce vor fi utilizate în acest îndrumar se numără: *greedy*, *recursivitate*, *backtracking* și *divide et impera*.

Etimologia cuvântului *algoritm* provine de la denumirea cărții *Algoritmi de numero Indorum* (sec. XII), care este o traducere în latină după *Muhammad ibn Musa Khwarizmi* (محمد بن موسى الخوارزمي – *Muhammad bin Mūsā Abū Ǧa'far al-Ḫawārazmī*), matematician, astronom și geograf persan din secolul VII. Cuvântul *algoritmi* este traducerea latină a numelui *Al-Khwārizmī* (*cel din Khwārizm*).

Al-Khwārizmī este recunoscut de mulți matematicieni ca fiind părintele algebrei. El a prezentat, pentru prima dată, un mod sistematic de rezolvare a ecuațiilor liniare și pătratice. În *Arithmetica* a explicat utilizarea cifrelor arabe și a fost printre primii matematicieni care au folosit cifra zero.

Muhammad ibn Musa Khwarizmi

1. Lucrul cu fluxuri de tip fişier

Datorită faptului că informațiile prelucrate de majoritatea programelor se stochează pe suporturi externe, este necesară existența unor metode de lucru cu aceste suporturi.

Un **fişier** reprezintă un grup de octeți înrudiți. Un sistem de fişiere este un produs software folosit pentru organizarea şi păstrarea fişierelor pe un dispozitiv secundar de stocare.

Fiecare fişier are proprietăți care îl descriu. Sistemul de fişiere determină tipul proprietăților care descriu fişierul. În mod obişnuit, acesta înseamnă numele fişierului, permisiunile de acces, data şi ora ultimei modificări a fişierului. În general, cele trei permisiuni de acces folosite sunt: citire / scriere (read / write), numai citire (read-only) şi execuție.

Fişierele sunt organizate în directoare şi subdirectoare. Directorul aflat pe poziția cea mai înaltă a ierarhiei se numeşte **director rădăcina**. De aici începe întreaga ierarhie de directoare şi subdirectoare. Colectiv, directoarele şi subdirectoarele formează **structura de directoare**.

1.1. Utilizarea fluxurilor de tip fişier în limbajul C/C++

Fişierele pot fi clasificate după conținutul lor în:

- fişiere text – conțin o secvență de caractere ASCII, structurate pe linii;
- fişiere binare – conțin o secvență de octeți, fără o structură predefinită.

1.1.1. Testarea existenței unui fişier

```
#include <stdio.h>
void main()
{
  FILE *fp;
  if (!(fp=fopen("d:\\fisier.txt","r+b")))
  {
    puts("Nu pot deschide fisierul");
    exit(1); //terminare fortata a executiei cu returnarea codului 1
  }
}
```

Fişierul "d:\fisier.txt" este deschis pentru citire şi scriere ("r+"), în mod binar ("b"). Pentru modul "r", dacă fişierul nu există pe disc sau operația de deschidere nu reuşeşte (de exemplu, dacă unitatea de disc nu este pregătită), `fopen()` întoarce valoarea `NULL`. În caz contrar deschide fişierul, creează o structură `FILE` şi întoarce adresa sa.

1.1.2. Copierea unui fişier

```
#include <stdio.h>
void main()
{
  FILE *fps, *fpd;
  char c;
  if (!(fps=fopen("d:\\fin.dat","rb"))==NULL)
  {
    puts("Nu pot deschide fisierul sursa");
    return;
  }

  if((fpd=fopen("d:\\fout.dat","wb"))==NULL)
  {
    puts("Nu pot crea fisierul destinatie");
    return;
  }

  while (!feof(fps))
  {
    c=getc(fps);
    if (!feof(fps))
    {
      putc(c, fpd);
    }
  }

  fclose(fps);
  fcolse(fpd);
}
```

Deoarece fişierele conțin date oarecare, ciclul de copiere foloseşte funcția `feof( )` pentru a detecta sfârşitul de fișier. Această precauție nu era necesară în cazul unui fişier text (alcătuit din coduri ASCII), unde ar fi suficientă testarea valorii întoarse de getc( ) pentru a detecta caracterul EOF. Trebuie remarcat că funcția `feof()` semnalează sfârşitul fişierului abia după ce acesta a fost întâlnit la operația de intrare precedentă. Din acest motiv, dacă instrucțiunea `if` ar lipsi, în fişierul "d:\fout.dat" s-ar copia în plus valoarea `EOF`.

1.2. Utilizarea fluxurilor în limbajul Java

Un flux, în contextul limbajului Java, este un **canal de comunicație unidirecțional** între doi actori; unul este producător și reprezintă sursa de date, iar cel de-al doilea reprezintă consumatorul. Prin prisma acestui aspect, fluxurile se clasifică în:

- fluxuri de intrare (prin care se primesc date în program);
- fluxuri de ieșire (prin care se transmit date din program).

De asemenea, datorită faptului că limbajul Java utilizează sistemul Unicode pentru tipul caracter, ceea ce înseamnă că un caracter ocupă 16 biți, API-ul Java introduce două tipuri de fluxuri:

- fluxuri de caracter (pentru date de tip text);
- fluxuri de octeți (pentru date binare).

Fluxurile sunt implementate în clase aflate în pachetul `java.io`.

Clasele pentru lucrul cu fluxuri de caracter de intrare moștenesc clasa abstractă `Reader`, iar clasele pentru fluxuri de octeți de intrare moștenesc clasa abstractă `InputStream`.

Clasele pentru lucrul cu fluxuri de caracter de ieșire moștenesc clasa abstractă `Writer`, iar clasele pentru fluxuri de octeți de ieșire moștenesc clasa abstractă `OutputStream`.

Clasele de lucru cu fluxurile de intrare oferă metoda `read()` pentru a prelua date din flux, care returnează o valoare `int`. În cazul fluxurilor de caracter, această valoare reprezintă codul Unicode al caracterului citit, prin urmare doar cei mai puțin semnificativi 2 octeți sunt utilizați în mod real. Ceilalți 2 octeți au tot biții 0. Pentru fluxurile de octeți, doar un octet este utilizat (cel mai puțin semnificativ). Dacă fluxul de intrare nu mai are date, atunci metoda `read()` returnează valoarea -1.

Clasele de lucru cu fluxurile de ieșire oferă metoda write().

De exemplu, lucrul cu fișiere poate fi realizat cu ajutorul claselor:

- `FileReader` – flux de caractere de intrare;
- `FileWriter` – flux de caractere de ieșire;
- `FileInputStream` – flux de octeți de intrare;
- `FileOutputStrem` – flux de octeți de ieșire.

Aceste fluxuri sunt fluxuri primitive, deoarece sunt conectate direct de sursa sau destinația datelor. Există și fluxuri de procesare care se construiesc pe baza fluxurilor primitive și oferă facilități suplimentare. Un tip de flux de procesare este fluxul cu zona de memorie tampon ("buffered"); astfel se pot utiliza următoarele clase:

- `BufferedReader` – flux de caractere de intrare;
- `BufferedWriter` – flux de caractere de ieșire;
- `BufferedInputStream` – flux de octeți de intrare;
- `BufferedOutputStream` – flux de octeți de ieșire.

Un tip util de fluxuri e procesare sunt cele de conversie a fluxurilor:

- `InputStreamReader` – flux de caractere de intrare care convertește un flux de octeți în flux de caractere;

- `OutputStreamWriter` – flux de octeți de ieșire care convertește un flux de caractere în flux de octeți.

Pașii de urmat în lucrul cu fluxuri sunt:

1. Deschidere flux (prin instanțierea unei clase de lucru cu fluxuri)
2. Procesare date (folosind metodele specifice clasei alese pentru lucrul cu fluxuri)
3. Închidere flux (prin apelarea metodei `close()` a instanței clasei de lucru cu fluxuri)

Orice eroare în lucrul cu fișierele va rezulta în aruncarea unei excepții, care provine din clasa `IOException`. Din acest motiv este obligatoriu ca programele care utilizează fluxuri să trateze toate excepțiile de acest tip.

CitireaDinFisier.java

```
1  importjava.util.*;
2  import java.io.*;
3
4  public class CitireaDinFisier
5  {
6  public static void main(String[] args)
7  {
8      String linie;
9      try {
10 BufferedReader in = new BufferedReader(new FileReader(
11           "\\fis.txt"));
12       while ((linie = in.readLine()) != null)
13       {
14 System.out.println(linie);
15       }
16     }
17     catch (IOException e)
18     {
19 System.out.println(e);
20     }
21   }
22 }
```

1.2.1. Obținerea versiunii în tonalități de gri (grayscale) a unei imagini BMP

Formatul BMP (bitmap) este utilizat pentru stocarea informațiilor de tip grafic. Formatul defineşte o parte de header, care conține informații despre documentul de tip imagine şi ocupă, în mod tipic, 54 de octeți, şi o parte de date, care conține efectiv imaginea.

În mod tipic, pentru fiecare punct din imagine (pixel) sunt necesari 24 de biți (care defineşte şi adâncimea de culoare), care stochează nivelele de intensitate pentru cele 3 componente ale culorii (R – roşu, G – verde şi B – albastru), formatul BMP utilizând spațiul de culoare RGB. Fiecare

componentă este memorată pe un octet, astfel că intensitatea luminoasă poate varia între 0 (lipsa componentei de culoare) şi 255 (intensitate maximă a culorii).

Pentru a obține versiunea în tonalități de gri a unei imagini este suficient a se calcula o medie între intensitățile culorilor componente ale fiecărui pixel. Deoarece ochiul uman este mai sensibil la culoarea verde, se preferă utilizarea unei medii ponderate, în care culoarea verde este predominantă. Această medie este folosită în calcularea luminanței (componenta Y a spațiului de culoare YUV). Formula de calcul poate fi regăsită pe linia 27 a programului.

BmpClass.java

```
 1  package bmp;
 2
 3  import java.io.BufferedInputStream;
 4  import java.io.BufferedOutputStream;
 5  import java.io.FileInputStream;
 6  import java.io.FileOutputStream;
 7
 8  public class BmpClass
 9  {
10    public static void main(String[] args)
11    {
12      try
13      {
14        FileInputStream fis=new FileInputStream("C:\\qwerty.bmp");
15        BufferedInputStream dis=new BufferedInputStream(fis);
16        FileOutputStream sif=new FileOutputStream("C:\\qwerty1.bmp");
17        BufferedOutputStream bos = new BufferedOutputStream(sif);
18        byte[] sti=new byte[54];
19        dis.read(sti,0,54);
20        bos.write(sti);
21        while(dis.available()>0)
22        {
23          int b,g,r;
24          b=dis.read();
25          g=dis.read();
26          r=dis.read();
27          int gri=(int)(0.114*b+0.587*g+0.299*r);
28          bos.write(gri);
29          bos.write(gri);
30          bos.write(gri);
31        }
32        dis.close();
33        bos.close();
34      }
35      catch (Exception e)
36      {
37      }
38    }
39  }
```

Headerul fişierului de intrare se copiază fără modificări, deoarece doar culorile imaginii se schimbă, nu proprietățile acesteia.

Fişierul este parcurs apoi octet cu octet şi grupuri de câte trei octeți sunt utilizate pentru a calcula luminanța. În fişierul de ieşire este înscrisă de trei ori luminanța, ceea ce înseamnă că se înlocuieşte un pixel colorat cu unul de culoare gri la intensitatea dată de luminanță.

Să se scrie un program care să genereze, pe baza unui fişier bitmap de intrare, trei fişiere bitmap, care prezintă separat cele trei componente de culoare.

2. Liste

2.1. Tipuri de date

2.1.1. Tipuri de date definite de utilizator

Acestea se obțin din tipurile de bază sau din combinații ale acestora sau chiar ale altor tipuri utilizator.

Limbajele C/C++ pun la dispoziție o serie de mecanisme şi structuri ce permit definirea unor tipuri de date noi. Printre acestea putem menționa *structurile*, *uniunile*, *enumerările* şi *clasele*.

Noilor tipuri de date li se pot asocia nume (identificatori) fie direct, fie prin utilizarea declarației `typedef`.

Declararea unor variabile de tip utilizator se face la fel cu cea a tipurilor predefinite. Sintaxa generală este:

Sintaxa utilizată:	`id_tip id_var1[, id_var2 [, ...]];`

Spre deosebire de declararea variabilelor de tip predefinit, declararea variabilelor de tip utilizator se poate face şi în locul definirii tipului respectiv (în cazul în care nu se utilizează declarația `typedef`).

2.1.1.1. Enumerări

Se folosesc la definirea unor mulțimi de constante. Cuvântul cheie care semnalează declarația unei enumerări este *enum*. Sintaxa este:

Format general:

```
enum [id_tip_nou] {nume_const_1 [=<valoare>], ...} [lista_variabile];
```

unde:

- `id_tip_nou` – identificatorul tipului (poate lipsi);
- `nume_const_x` – identificatorul constantei *x* din enumerare;
- `valoare` – este valoarea atribuită constantei respective; poate lipsi, caz în care valoarea ei va fi egală cu valoarea constantei definite anterior incrementată cu 1;
- `lista_variabile` – lista variabilelor ce se doresc a fi utilizate; această listă poate lipsi, fiind posibilă o declarare ulterioară (în cazul în care a fost denumit noul tip).

Se pot utiliza şi declarațiile de tip astfel:

```
typedef enum {<lista_constante>} id_tip_nou;
```

Exemplu:

```
enum mVid {LASTMODE=-1, BW40=0, C40, BW80, C80, MONO=7};
```

Aceeaşi declarație se poate scrie şi astfel:

```
typedef enum {LASTMODE=-1, BW40=0, C40, BW80, C80, MONO=7} mVid;
```

Exemplu de declarație de variabile:

```
mVid v1, v2;
```

2.1.1.2. Structuri

Structurile reprezintă unul dintre cele mai puternice mijloace de definire a noi tipuri de date, permițând înglobarea mai multor variabile într-un ansamblu unitar. Definirea unei structuri se realizează cu ajutorul cuvântului cheie `struct`.

Format general:

```
struct [id_structura]
{
  [id_tip id_var[, id_var, ...]] ;
  ...
} [lista_variabile];
```

unde:

- `id_structura` – identificatorul (numele) structurii; el poate lipsi, caz în care structura va fi *anonimă*, nemaiputându-se declara alte variabile de acest tip în afara celor declarate imediat după definirea ei;
- `lista_variabile` – listă declarații de variabile (poate lipsi).

Operatorul ‘;’ nu trebuie să lipsească după definirea unei structuri.

Trebuie menționat faptul că dacă lipsesc atât numele cât şi lista de variabile, structura va fi inutilizabilă în program (nu va putea fi referită).

Declararea variabilelor de tip structură se poate realiza în două moduri:

- în locul definirii tipului structură (dacă nu a fost definită cu ajutorul declarației `typedef`);
- oriunde în program, utilizând numele dat structurii ca tip al variabilei.

În cazul utilizării declarației `typedef`, definirea unei structuri trebuie să respecte sintaxa:

Format general:

```
typedef struct
{
  [id_tip id_var[, id_var, ...]] ;
  ...
} [id_structura];
```

După cum a fost menționat anterior, o structură grupează mai multe variabile de diferite tipuri într-o aşa-numită *înregistrare* (*record*); aceste variabile poartă denumirea de membri ai structurii şi ei pot fi accesați atât direct, cât şi referiți prin adresele lor. Practic, o variabilă de tip structură va încorpora mai multe variabile de diferite tipuri sub un singur identificator.

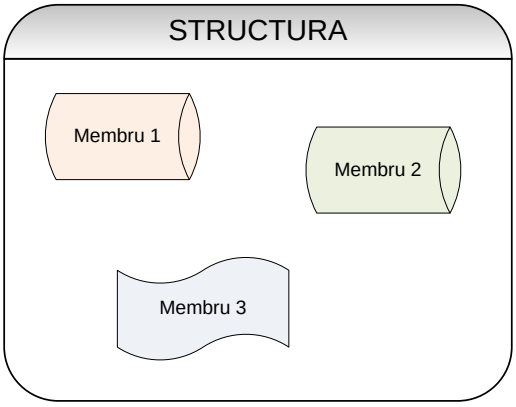


Fig. 2 – Structura grupează mai multe variabile sub un singur nume (identificator)

Accesul la oricare dintre membri unei variabile de tip structură se face cu ajutorul operatorului de selecție directă '.'.

Sintaxa utilizată pentru accesarea unui membru al unei structuri este:

Sintaxa utilizată:	`nume_variabilă_structură.identificator_membru`

Operația de inițializare a unei structuri respectă sintaxa:

Sintaxa utilizată:	`id_structura id_var={<valoare_membru_1>, ...};`

Atribuirea valorilor membrilor structurii se face *exact* în ordinea în care au fost aşezați în definirea tipului structura (prima valoare va fi atribuită primului membru din definiția structurii, a doua – celui de-al doilea membru, ş.a.m.d.).

Pentru a ilustra modul de lucru cu structurile, vom defini o structură ce va conține numele unui student şi media notelor acestuia; o vom inițializa, apoi, cu valorile „Popescu Andrei" pentru nume şi 9,78 pentru medie; vom afişa numele şi media pe ecran.

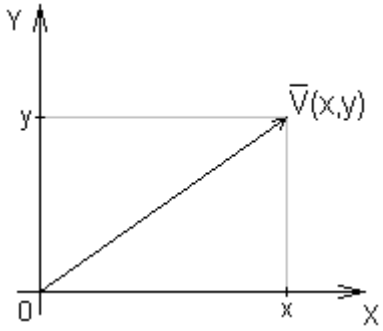
Student.cpp

```
#include <stdio.h>

// definim structura

struct Student
{
  char strNume[30];
  float fMedia;
};

// programul principal
void main()
{
  // declaram o variabila v1 de tip Student si o initializam
  Student v1={"Popescu Andrei", 9.79};

  // afisam continutul structurii v1
  printf("Studentul %s are media %g.\n",v1.strNume,v2.fMedia);
}
```

Să se scrie un program care să determine vectorul sumă obținut prin adunarea mai multor vectori bidimensionali a căror componente se vor citi de la tastatură, şi să se reprezinte grafic obținerea acesteia (fiecare vector va fi reprezentat prin altă culoare).

Indicații:

Un vector bidimensional este definit de două componente (luând un sistem de referință ortogonal) aşa cum este prezentat în figura de mai jos:



Problema se rezolvă uşor dacă se utilizează metoda triunghiului de însumare a vectorilor.

2.2. Pointeri

Pointerul este un tip de date ce permite declararea unor variabile capabile să memoreze adresa unei alte variabile sau a unei funcții.

Variabilele de tip pointer nu includ nici o informație despre tipul datelor stocate la acea adresă, de aceea, în momentul declarării unei variabile de tip pointer, trebuie să se specifice către ce tip de date va face referire.

Pointerul indică adresa primului byte din variabila la care face referire. Prin urmare, un pointer va ocupa acelaşi spațiu de memorie, indiferent de tipul de date către care face referire. Spațiul de memorie ocupat de o variabilă pointer poate varia între 16 biți şi 64 de biți.

În limbajele C/C++, declararea unui pointer către o variabilă de tip `int` se face astfel:

```
int *pVal;
```

Caracterul `*` ne indică faptul că variabila `pVal` este un pointer şi nu o variabilă de tip `int`.

Preluarea adresei de memorie a unei variabile se face cu ajutorul operatorului `&` (adresă).

Accesul la conținutul variabilei referite de un pointer (conținutul zonei de memorie referite) se realizează cu ajutorul operatorului `*` plasat înaintea numelui variabilei de tip pointer. Tipul variabilei pointer indică numărul de octeți ce vor fi citiți din memorie şi modul cum vor fi interpretați.

Pentru a exemplifica modul de lucru cu pointerii, vom comenta următorul exemplu:

```
#include <stdio.h>

void main()
{
1  int val=2,*pVal;
2  pVal=&val;
3  *pVal+=5;
4  printf("Noua valoare a variabilei val este: %d.\n",val);
5  printf("Variabila val este memorata la adresa %p, ocupa %d octeti si
    are valoarea %d.\n",pVal,sizeof(val),*pVal);
}
```

În linia 1 a funcției `main()` se declară două variabile: un `int` (`val`) şi un pointer către `int` (`pVal`). Variabila `val` este inițializată cu valoarea 2.

Linia 2 determină inițializarea variabilei de tip pointer `pVal` cu adresa varabilei `val`.

O variabilă de tip pointer trebuie întotdeauna inițializată înainte de a o întrebuința. Pentru a specifica o non-valoare (adică variabila pointer nu conține nici o adresă) se utilizează constanta `NULL`.

Utilizând pointerul `pVal` se modifică valoarea variabilei `val` (conținutul de tip `int` aflat la adresa specificată de pointerul `pVal`) în linia 3.

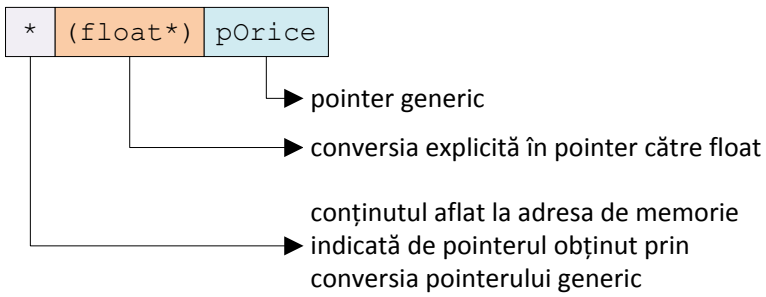
În linia 4 se afişează valoarea variabilei `val`, care acum va fi 7.

Linia 5 afişează adresa variabilei `val` (în format hexazecimal), dimensiunea în octeți şi valoarea de tip `int` aflată la adresa dată de valoarea pointerului.

Se pot declara şi pointeri generici (la care nu se specifică un tip bine definit), dar în momentul operării asupra lor trebuie convertiți în mod explicit, în concordanță cu tipul datelor referite. De exemplu , se poate scrie un program astfel:

```
   #include <stdio.h>

   void main()
   {
1    float PI=3.14;
2    void *pOrice;
3    pOrice=&PI;
4    printf("Variabila PI se afla la adresa %p si are
              valoarea %g.\n",pOrice,*(float*)pOrice);
   }
```

În momentul afişării a trebuit convertit pointerul generic pOrice într-un pointer către tipul float (tipul variabilei referite) şi apoi afişat conținutul acestei locații de memorie.



2.2.1. Pointerii şi tablourile

Pointerii şi tablourile sunt corelate strâns în limbajele C/C++, o variabilă tablou nefiind altceva decât un pointer către primul element al tabloului.

Având o declarație `int Tab[10]`, egalitatea `Tab=&Tab[0]` va fi adevărată.

2.2.2. Pointerii şi structurile

Ca şi pentru orice alt tip de date, şi pentru structuri se pot declara pointeri. Accesul la membrii structurii se va face, în acest caz, cu ajutorul operatorului de selecție indirectă "`->`" (aşa cum arată exemplul „PStruct.cpp").

PStruct.cpp

```
#include <stdio.h>

struct Coord
{
int x,y;
};

void main()
{
  Coord varf, *pVarf;
  varf.x = 0;
  varf.y = 1;
pVarf=&varf;
printf("Varful are coordonatele x=%d, y=%d", pVarf->x, pVarf->y);
}
```

2.2.3. Gestiunea dinamică a memoriei cu operatorii `new` şi `delete`

Rolul esențial al pointerilor este reprezentat de gestiunea dinamică a memoriei. Cu ajutorul lor se poate eficientiza un program din punctul de vedere al utilizării memoriei.

În cazul în care nu se poate şti dinainte de câtă memorie va avea nevoie un program, sau cantitatea necesară depăşeşte capacitatea alocării statice de memorie, se utilizează alocarea dinamică de memorie. Această metodă, aplicată corect, asigură utilizarea eficientă a memoriei.

Metoda presupune alocarea unor zone de memorie doar în momentul în care programul are nevoie de acestea şi le eliberează imediat ce devin inutile.

Specific acestui mod de lucru este faptul că alocarea şi eliberarea memoriei cade în grija programatorului. În acest sens, limbajul C++ pune la dispoziție doi operatori dedicați gestiunii memoriei: `new` şi `delete`.

2.2.3.1. Operatorul `new`

Cu ajutorul operatorului `new` programatorul poate aloca memorie în mod dinamic (în timpul rulării programului).

Sintaxa utilizată:	`var_pointer = new tip_alocat;`

În urma unei atribuiri de acest fel se alocă o zonă de memorie (în memoria *heap* – memoria de acumulare, adică nu aparține stivei programului) corespunzătoare tipului declarat, care va putea fi referită prin intermediul variabilei pointer `var_pointer`. În cazul imposibilității alocării de memorie, operatorul `new` va returna valoarea NULL.

Operatorul `new` permite alocarea de memorie pentru orice tip de date.

Tipul variabilei pointer trebuie să coincidă cu tipul furnizat operatorului `new`.

2.2.3.2. Operatorul `delete`

Orice zonă de memorie alocată în mod dinamic trebuie eliberată, sarcina revenind programatorului.

Limbajul C++ defineşte pentru această operație operatorul `delete` (complementar operatorului `new`).

Deşi limbajele C/C++ pun la dispoziție mai multe funcții destinate lucrului dinamic cu memoria, este indicată utilizarea "în pereche" a acestora; astfel, dacă alocarea s-a făcut utilizând operatorul `new`, eliberarea memoriei se va face cu operatorul `delete`.Aceeaşi regulă este valabilă şi pentru funcțiile `malloc()` şi `free()`.

Sintaxa utilizată:	`delete var_pointer;`

Acest apel va produce eliberarea zonei de memorie, alocată în prealabil cu operatorul `new` şi referită de pointerul `var_pointer`.

Întotdeauna trebuie păstrată o referință către o zonă de memorie alocată dinamic. În caz contrar, ea nu va mai putea fi eliberată, lucru ce poate conduce la consumarea memoriei din sistem şi, implicit, funcționări defectuoase ale programului ce pot bloca sistemul pe care rulează.

2.3. Liste

Definiție: Listele ordonate liniar sunt structuri de date alcătuite dintr-o mulțime $A=\{A_1, A_2, \ldots, A_n\}$de elemente (de obicei identice), între care există o relație determinată de poziția lor relativă. Astfel, fiecare element A_k are un predecesor A_{k-1} şi un succesor A_{k+1} (mai puțin elementele prim şi ultim).

Reprezentarea în memorie se poate realiza:

- secvențial;
- prin înlănțuirea elementelor.

2.3.1. Reprezentarea secvențială

În acest tip de reprezentare, elementele sunt dispuse succesiv, într-o zonă contiguă de memorie. Reprezentarea este similară cu cea a unui tablou (nu trebuie făcută identificarea listei cu obiectul

tablou în care sunt memorate elementele ei). (Un tablou este un caz particular de listă, acela în care elementul i al listei se află memorat în tab [i]).

2.3.2. Liste înlănțuite

Definiție: O listă înlănțuită este alcătuită din noduri cu structura date şi legături în care:

- câmpul DATE reprezintă informația propriu zisă (un element al listei);
- câmpul LEGĂTURĂ reprezintă informația de secvență, legătura spre elementele adiacente din listă.

Observație: Informația de secvență se adaugă explicit pentru fiecare element, sub forma adreselor elementelor adiacente, astfel încât ordinea listei devine independentă de ordinea plasării elementelor în memorie.

Clasificare:

- listă simplă înlănțuită;
- listă dublu înlănțuită.

Listă simplu înlănțuită - conține noduri în care este specificată legătura (adresa) spre elementul următor.

Exemplu de structură nod pentru lista simplu înlănțuită:

```
struct nlsi
{
  double data;  /* informația din nodul listei */
  struct nlsi *urm;     /* adresa nodului următor */
};
```

Definiție: O listă înlănțuită în care legătura ultimului element are valoarea NULL, care marchează sfârşitul listei, se numeşte lanț.

Definiție: Dacă legătura elementului final specifică primul element se obține o listă **circulară**.

Observație: În anumite aplicații algoritmii sunt simplificați de utilizarea unui nod distinct, numit nod **capăt**, la care este ataşată lista propriu-zisă.

Pentru o listă simplu înlănțuită, ataşarea unui element nou şi extragerea unui element din listă sunt operații banale şi rapide, cu condiția să fie cunoscută adresa elementului precedent.

Exemplu de utilizare a listelor simplu înlănțuite:

```
1  /*
2   cap = [inf|urm] -> [inf|urm] -> ... -> [inf|NULL]
3  */
4  #include <iostream.h>
5  #include <conio.h>
6  #include <stdio.h>
```

```
 7  // structura de date
 8  struct Nod
 9  {
10    int inf; // informatia utila, pe care dorim sa o memoram
11    Nod* urm; // adresa urmatorului Nod (un pointer catre o variabila Nod)
12  };
13
14  // Functia returneaza numarul de elemente din lista
15  int nrElemente(Nod* lista)
16  {
17    int nr=0;
18    Nod *aux = lista;
19    while (aux!=NULL)
20    {
21      nr++;
22      aux = aux->urm;
23    }
24
25    return nr;
26  }
27
28  /*
29  Adaugare in pozitia N a unui element:
30  - daca N este 1, atunci adaugarea se face la inceputul listei;
31  - daca N este mai mare decat numarul de elemente, atunci se face adaugare
32    la sfarsit de lista.
33  Lista vida va fi simbolizata de valoarea NULL pentru capul acesteia.
34  */
35
36  void adauga(Nod *&lista, int val, int N)
37  {
38    if (N==1 || lista == NULL) // adaugare la inceputul listei => capul
    listei
39                               // isi va schimba valoarea
40    {
41      Nod *aux = new Nod; // se aloca memoria necesara memorarii
42                          // unei variabile de tip Nod
43      aux->inf = val; // se completeaza partea de informatie utila
44      aux->urm = lista; // urmatorul element va fi chiar vechiul cap al
    listei
45      lista = aux; // noul cap al listei este noul element creat
46    }
47    else
48    {
49      int nr = nrElemente(lista); // preiau numarul de elemente din lista
50      if (N>nr+1) // daca pozitia primita ca parametru este mai mare decat
51                  // numarul de elemente + 1, atunci ii modific valoarea in
52                  // numar de elemente in lista + 1
53        N=nr+1;
54
55      Nod *tmp = new Nod;  // creare noului nod - alocare memorie
56      tmp->inf = val; // completarea informatiei utile
57
58      // se parcurge lista pana la al N-1 -lea element
59      Nod *aux = lista;
60      for (int i=1;i<N-1;i++)
61        aux = aux->urm;
62
63      // realizarea inlantuirii
64      tmp->urm = aux->urm;
65      aux->urm = tmp;
66    }
67  }
```

```
 68 // functie de afisare a elementelor din lista
 69 void afisare(Nod *lista)
 70 {
 71   Nod *aux = lista; // afisarea incepe de la capul listei
 72 
 73   cout<<"Lista este: ";
 74 
 75   while(aux != NULL)
 76   {
 77     cout<<aux->inf<<" ";
 78     aux = aux->urm;
 79   }
 80 
 81   if (lista == NULL)
 82     cout<<"vida."<<endl;
 83   else
 84     cout<<endl;
 85 }
 86 
 87 // functie pentru eliberarea memoriei
 88 void eliberaremem(Nod *&lista)
 89 {
 90   Nod* aux = lista;
 91 
 92   while(aux != NULL)
 93   {
 94     lista = lista->urm; // mutam capul listei pe urmatorul element
 95     delete aux; // stergem elementul curent
 96     aux = lista; // primul element va deveni elementul curent
 97   }
 98 }
 99 
100 /*
101 Functia elimin elibereaza memoria alocata elementului de pe pozitia poz
102 - daca poz este 1, atunci  se va elimina de la capul listei
103 - daca poz este egal sau mai mare decat nr. de elem., atunci se elimina
104   de la sfarsitul listei
105 */
106 void elimin(Nod *&lista, int poz)
107 {
108   Nod* aux = lista;
109   if (lista == NULL)
110     return;
111 
112   if (poz>nrElemente(lista))
113     poz = nrElemente(lista);  // daca poz este mai mare decat numarul de
114                               // elemente din lista, atunci lui poz i se
115                               // va da valoarea pozitiei ultimului elem.
116   if (poz == 1)
117   {
118     lista=lista->urm; // se muta capul listei pe elementul urmator
119     delete aux; //se sterge capul listei ce e tinut acum de variabila aux
120   }
121   else
122   {
123     // pozitionare pe elementul dinaintea pozitiei poz
124     for (int i=1;i<poz-1;i++)
125       aux = aux->urm;
126 
127     Nod *tmp = aux->urm;
128     aux->urm = tmp->urm;
129     delete tmp;
130   }
```

```
131 }
132
133 void main()
134 {
135   clrscr(); // stergere ecran
136   Nod *L = NULL; // initial lista L este vida
137   int opt=0;
138
139   do
140   {
141     // se afiseaza "meniul" programului
142     cout<<"Operatii"<<endl<<"------"<<endl<<"1 - Adaugare element"<<endl;
143     cout<<"2 - Eliminare element"<<endl<<"3 - Afisare lista"<<endl;
144     cout<<"0 - Iesire"<<endl<<"------"<<endl;
145
146     // se citeste codul operatiei pe care utilizatorul vrea sa o execute
147     cout<<"?> ";
148     cin>>opt;
149
150     switch (opt)
151     {
152       case 1: // adaugare
153       {
154         int v,p;
155         cout<<endl<<"Introduceti valoarea: ";
156         cin>>v;
157         cout<<"Introduceti pozitia (lista are "<<nrElemente(L)<<" el): ";
158         cin>>p;
159         adauga(L,v,p);
160         afisare(L);
161         break;
162       }
163       case 2: // eliminare
164       {
165         int p;
166         cout<<endl<<"Pozitia (lista are "<<nrElemente(L)<<" el): ";
167         cin>>p;
168         elimin(L,p);
169         afisare(L);
170         break;
171       }
172       case 3: // afisare
173       {
174         cout<<endl;
175         afisare(L);
176         break;
177       }
178     }
179
180     fflush(stdin); // pentru eliberarea bufferului tastaturii
181     getch();
182   }
183   while (opt!=0);
184
185   eliberaremem(L); // se elibereaza memoria alocate la crearea listei
186 }
```

Să se modifice programul de mai sus prin implementarea opțiunii de eliminare a elementelor negative din listă (opțiunea 4).

3. Arbori binari

Organizarea liniară de tip listă nu este întotdeauna cea mai adecvată pentru unele aplicații. Astfel, dacă trebuie să descriem structura unui produs, de cele mai multe ori nu prezentăm o listă a tuturor componentelor, ci utilizăm o descriere ierarhică. De exemplu, din punct de vedere constructiv, un calculator este compus din unitate centrală, terminal, claviatură, alte periferice. Unitatea centrală are o carcasă în care se află conectori şi plăci, pe fiecare placă fiind montate diverse componente - circuite integrate, condensatori, etc.

De altfel, în vederea studierii unor proprietăți, aproape orice obiect poate fi descompus în alte obiecte, mai simple. Procesul de descompunere poate fi continuat pe mai multe niveluri, terminându-se însă după un număr finit de etape, dependent de natura aplicației. Fiecare obiect în parte este definit printr-un set de atribute şi prin mulțimea obiectelor componente, care la rândul lor, sunt descrise în acelaşi mod . Se observă că această definiție este recursivă (un obiect este compus din mai multe obiecte) şi pune în evidență o ierarhie a obiectelor.

Organizarea ierarhică este întâlnită în cele mai diverse domenii, de la organizarea administrativă a unei țări, la planificarea meciurilor în cadrul unui turneu sportiv, de la structurarea unei cărți, până la stabilirea ordinii de execuție a operațiilor efectuate pentru determinarea valorii unei expresii aritmetice.

Tot o structură ierarhică este şi cea a cataloagelor în care sunt grupate fişierele de pe discurile fixe sau flexibile. Această organizare este impusă, in principal, de rațiuni de gestionare cât mai comodă a fişierelor de diverse tipuri, aparținând diverşilor utilizatori ai aceluiaşi sistem de calcul.

Dacă toate nodurile dintr-un arbore au cel mult doi descendenți direcți (fii), atunci arborele este denumit arbore *binar*, iar cei doi potențiali subarbori ai unui arbore nevid sunt denumiți *subarbore stâng* şi *subarbore drept*.

3.1. Parcurgerea arborilor

Prelucrarea informațiilor memorate într-o structură de arbore implică parcurgerea arborelui, adică inspectarea (vizitarea) fiecărui nod şi prelucrarea informației specifice. Problema care se pune este cea a ordinii în care se prelucrează nodurile arborelui (rădăcina şi, respectiv, nodurile din subarbori). De cele mai multe ori, aceasta este impusă de specificul aplicației.

De exemplu, dacă memorăm într-o structură de arbore multicăi informațiile despre organizarea unei societăți comerciale, acest arbore poate fi parcurs în mai multe moduri, în funcție de

prelucrarea dorită. În cazul în care este solicitată lista personalului cu funcții de conducere, aceasta poate fi tipărită in două variante:

1. grupând persoanele pe nivelurile ierarhice;
2. astfel încât să reflecte relațiile de subordonare.

În prima variantă, parcurgerea arborelui se efectuează în *lățime*, iar în cea de-a doua în *adâncime*.

Tot o parcurgere implică şi calculul numărului de persoane angajate în fiecare compartiment (reprezentat printr-un subarbore). Cele două parcurgeri în adâncime menționate se deosebesc prin ordinea relativă de prelucrare a nodului rădăcină şi, respectiv, a subarborilor. În primul caz, informația specifică nodului rădăcină este prelucrată (tipărită) înaintea informațiilor din celelalte noduri ale (sub)arborelui. Acest tip de parcurgere este denumit parcurgere în *preordine*. Deoarece numărul de persoane angajate într-un compartiment poate fi calculat (şi eventual tipărit) numai atunci când se cunoaşte numărul de angajați din toate compartimentele subordonate, rezultă că prelucrarea de la nivelul nodului rădăcina se face după prelucrarea restului arborelui respectiv, deci parcurgerea se realizează în *postordine*.

Implementarea de mai jos utilizează *recursivitatea* ca tehnică de programare.

Parcurgerea arborilor binari:

```
 1  #include <iostream.h>
 2  #include <conio.h>
 3
 4  struct ArboreBinar
 5  {
 6    int inf;
 7    ArboreBinar *st;
 8    ArboreBinar *dr;
 9  };
10
11  ArboreBinar* createTree()
12  {
13    int are;
14    ArboreBinar *nod = new ArboreBinar;
15    cout<<"\nInformatia pentru nodul nou: ";
16    cin>>nod->inf;
17    cout<<"Nodul "<<nod->inf<<" are subarbore stang [0=NU;1=DA]? ";
18    cin>>are;
19    if (are!=0)
20      nod->st = createTree();
21    else
22      nod->st = NULL;
23    cout<<"Nodul "<<nod->inf<<" are subarbore drept [0=NU;1=DA]? ";
24    cin>>are;
25    if (are!=0)
26      nod->dr = createTree();
27    else
28      nod->dr = NULL;
29    cout<<endl;
30    return nod;
31  }
```

```
32
33  void inordine(ArboreBinar *root)
34  {
35    if (root->st!=NULL)
36      inordine(root->st);
37    cout<<root->inf<<" ";
38    if (root->dr!=NULL)
39      inordine(root->dr);
40  }
41
42  void preordine(ArboreBinar *root)
43  {
44    cout<<root->inf<<" ";
45    if (root->st!=NULL)
46      preordine(root->st);
47    if (root->dr!=NULL)
48      preordine(root->dr);
49  }
50
51  void postordine(ArboreBinar *root)
52  {
53    cout<<root->inf<<"(";
54    if (root->st!=NULL)
55      postordine(root->st);
56    cout<<",";
57    if (root->dr!=NULL)
58      postordine(root->dr);
59    cout<<")";
60  }
61
62  void freeTree(ArboreBinar *root)
63  {
64    if (root->st!=NULL)
65      freeTree(root->st);
66    if (root->dr!=NULL)
67      freeTree(root->dr);
68    delete root;
69  }
70
71  void main()
72  {
73    // Pointer catre radacina arborelui
74    ArboreBinar *root=NULL;
75
76    // Crearea recursiva a arborelui
77    root = createTree();
78
79    cout<<"Apasati orice tasta pentru a continua!";
80    getch();
81    clrscr();
82
83    cout<<"Arborele este: ";
84    postordine(root);
85    cout<<endl<<endl;
86
87    cout<<"Parcurgere in inordine: ";
88    inordine(root);
89    cout<<endl<<endl;
90
91    cout<<"Parcurgere in preordine: ";
92    preordine(root);
93    cout<<endl;
94
```

```
95    // Eliberarea memoriei
96    freeTree(root);
97    root = NULL;
98    getch();
99  }
```

Să se scrie un program care parcurge un arbore binar în lățime.

4. Backtracking

Tehnica *Backtracking* (a căutării cu revenire) se poate aplica doar pentru probleme ce admit conceptul de „candidat parțial de soluție” și oferă un test relativ rapid asupra posibilității ca un astfel de candidat să fie completat către o soluție validă. Când se poate aplica, însă, backtrackingul este adesea mult mai rapid decât căutarea prin metoda forței brute prin toți candidații, întrucât este capabilă să elimine dintr-un singur test un mare număr de candidați.

Metoda *Backtracking*, în general ineficientă, având complexitate exponențială, poate fi utilizată la optimizarea procesului de căutare, evitând căile care nu duc la o soluție.

În multe aplicații, găsirea soluțiilor este rezultatului unui proces de căutare sistematică, cu încercări repetate şi reveniri în caz de nereuşită. De exemplu, un întreprinzător care dispune de un capital C, pe care doreşte să-l investească în vederea obținerii unui profit, va proceda în felul următor: va alege dintre n oferte la care trebuie avansate fondurile f(i) şi care aduc beneficiile b(i), pe acelea pe care le poate onora cu capitalul de care dispune şi care-i aduc beneficiul maxim.

Dintre problemele ale căror soluții se găsesc prin căutare menționăm câteva exemple:

- găsirea unei căi de ieşire dintr-un labirint;
- plasarea pe o tablă de şah a opt dame care nu se atacă între ele;
- găsirea unui traseu care acoperă tabla de şah, generat adoptând săritura calului fără a trece de două ori prin aceeaşi poziție.

În general, aceste probleme pot avea mai multe soluții.

O metodă directă de găsire a soluțiilor este numită *metoda forței brute*, care va căuta să genereze toate submulțimile de oferte. Pentru fiecare din cele 2 la puterea n submulțimi distincte se calculează suma investită şi profitul adus. Se reține cele care nu depaşesc oferta şi aduc profitul maxim.

O soluție a problemei ar fi o selecție de oferte (s_1, s_2, ..., s_n), în care s_i=1 sau s_i=0 după cum oferta este sau nu onorată. Soluțiile acestei probleme au aceeaşi lungime n. Condiția s_i să aparțină lui 0 sau 1 este o restricție explicită. Condiția ca suma investită să nu depăşească capitalul este o funcție de limitare, o restricție implicită, care restrânge numărul soluțiilor.

În metoda căutării cu revenire, soluția este constituită în mod progresiv, prin adăugarea unei componente s_{p+1} la o soluție parțială (s_1, s_2, …, s_p) care reprezintă o selecție din primele *p* oferte din totalul celor *n*, astfel încât (s_1, s_2, …, s_{p+1}) să reprezinte de asemenea o soluție parțială. Soluția parțială mai poate fi întâlnită şi sub numele de soluție *p*-realizabilă.

O soluție finală este obținută în momentul în care a fost făcută o selecție dintre cele n oferte. Aceasta este comparată cu soluția optimă determinată până acum, fiind reținută sau ignorată.

4.1. Problema săriturii calului pe tabla de şah

Se presupune existența unei table de şah de dimensiune 8x8. Trebuie să se găsească toate modalitățile de a deplasa un cal pe această tablă, astfel încât calul să treacă prin toate căsuțele de pe tablă, fără a trece de mai multe ori prin acelaşi loc.

Pentru a parcurge fiecare căsuță de pe tabla de şah exact o dată, calul trebuie trebui să facă exact 8×8=64 de paşi. La fiecare pas el poate alege oricare din cele 64 de căsuțe de pe tablă. Se codifică fiecare dintre căsuțele de pe tabla de şah în modul următor: căsuța de la linia i şi coloana j se notează prin perechea (i,j). Se notează mulțimea tuturor căsuțelor de pe tablă cu C: C = {(0,0), (0,1), ..., (0,7), (1,0), ..., (7,7)}.

O soluție a problemei se poate nota printr-un vector x = (x0, x1, ..., x63), unde x Є S = C×C×C× ... ×C (produs cartezian în care mulțimea C apare de 64 de ori), iar xi Є C, i Є {0, 1, ..., 63}.

Problema săriturii calului pe tabla de şah

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

/* Dimensiunea tablei de sah este definita ca si constanta. */
#define N 8
#define INVALID -1

int main(void)
{
    /* Pentru o tabla de dimensiune N solutiile sunt memorate intr-un
       vector de dimensiune N*N. Fiecare element din vector va fi,
       la randul lui, un vector cu doua elemente; primul element va
       memora linia de pe tabla, iar al doilea element va memora
       coloana de pe tabla. */
    int c[N*N][2];

    int k, i;
    int pe_tabla, continuare;
    int delta_l, delta_c;

    /* Sunt numarate solutiile gasite. */
    int count = 0;

    /* Pentru inceput se marcheaza toate elementele vectorului "c" cu
       INVALID, semn ca nu a fost ales nici un element din multimile
        produsului cartezian. */

```

```
28      for (i=0; i<N*N; i++)
29      {
30          c[i][0] = INVALID;
31          c[i][1] = INVALID;
32      }
33
34      k = 0;
35      while (k >= 0)
36      {
37            /* Se incearca plasarea mutarii "k" a calului in fiecare
38               casuta, pe rand. Se evalueaza posibilitatea continuarii.
39               Procesul se opreste cand au fost incercate toate casutele
40               sau cand este identificata o casuta libera */
41            do
42            {
43                /* Se alege urmatorul element din multimea "C[k]".
44                   Daca elementul "c[k]" este setat pe INVALID,
45                   inseamna ca inca nu a fost ales nici un element din
46                   multimea curenta, prin urmare se alege primul element
47                   calul este plasat pe casuta (0,0) */
48                if (c[k][0] == INVALID)
49                {
50                    c[k][0] = 0;
51                    c[k][1] = 0;
52                    pe_tabla = 1;
53                }
54                /* Daca elementul "c[k]" nu este setat pe INVALID, inseamna
55                   ca deja s-a ales o casuta din multimea "C[k]". Se alege
56                   urmatoarea casuta de pe tabla. Daca este posibil
57                   se ramane pe aceeasi linie, deplasarea realizandu-se
58                   pe coloana spre drepata. */
59                else
60                if (c[k][1] < N-1)
61                {
62                    c[k][1]++;
63                    pe_tabla = 1;
64                }
65                /* Daca este ultima casuta din linie, atunci se trece la
66                   linia urmatoare, cu verificarea sa nu fie ultima linie
67                   caz in care au fost epuizate toate casutele. */
68                else
69                if (c[k][0] < N-1)
70                {
71                    c[k][1] = 0;
72                    c[k][0]++;
73                    pe_tabla = 1;
74                }
75
```

```
76              /* Daca este ultima linie a tablei, atunci se marcheaza
77                 epuizarea casutelor, prin intermediul valorii zero
78                 atribuita variabilei "pe_tabla". */
79              else
80              {
81                  pe_tabla = 0;
82              }
83
84              /* Daca casuta "c[k]" aleasa este valida (se afla pe tabla
85                 de joc),atunci se evaluează posibilitatea continuarii */
86              if (pe_tabla)
87              {
88                  /* Daca este prima mutare, atunci este valida */
89                  if (k == 0)
90                      continuare = 1;
91                  /* Daca nu e prima mutare, se fac o serie de
92                     verificari. */
93                  else
94                  {
95                    /* Se verificam daca de la pozitia precedenta a
96                       calului pe tabla ("c[k-1]") se poate ajunge
97                       in pozitia aleasa. */
98                    delta_l = abs(c[k-1][0]-c[k][0]);
99                    delta_c = abs(c[k-1][1]-c[k][1]);
100                   continuare = (((delta_l == 1) &&
101                                  (delta_c == 2)) ||
102                                 ((delta_l == 2) &&
103                                  (delta_c == 1)));
104
105                   /* Se verifica daca a mai fost aleasa casuta */
106                   for (i=0; continuare && (i<k); i++)
107                   {
108                       if ((c[i][0] == c[k][0]) &&
109                           (c[i][1] == c[k][1]))
110                           continuare = 0;
111                   }
112                 }
113             }
114             /* Daca casuta "c[k]" aleasa este in afara tablei de sah,
115                atunci nu se poate continua */
116             else
117             {
118                 continuare = 0;
119             }
120         }
121         while (!continuare && pe_tabla);
122
123         /* Daca rezultatul este pozitiv in urma verificarilor de
```

```
124             continuare, atunci se considera piesa asezata la pozitia
125             "c[k]" si continuam cautarea. */
126          if (continuare)
127          {
128              /* Daca s-a parcurs toata tabla de sah, atunci solutia
129                 este afisata. */
130              if (k == N*N - 1)
131              {
132                    for (i=0; i<N*N; i++)
133                    {
134                        printf("(%d,%d) ", c[i][0], c[i][1]);
135                    }
136                    printf("\n");
137                    count++;
138              }
139              /* Daca nu a fost parcursa inca toata tabla, atunci se
140                 trece cu un pas inainte pe calea de cautare. */
141              else
142              {
143                  k++;
144              }
145          }
146
147          /* Daca casuta aleasa nu este valida, atunci se marcheaza ele-
148             mentul "c[k]" cu INVALID si se revine la pasul anterior. */
149          else
150          {
151              c[k][0] = INVALID;
152              c[k][1] = INVALID;
153              k--;
154          }
155     }
156
157     printf("%d solutii\n", count);
158     return 0;
159 }
```

Problema celor 8 regine

Considerându-se o tablă de şah de dimensiune 8x8, să se aşeze pe această tablă de şah 8 regine astfel încât să nu existe două regine care se atace între ele.

5. Algoritmi de căutare

O mare gamă a cercetărilor în acest domeniu urmăresc obținerea minimului în strategii minimax. În problemele privind parcurgerea combinațională a arborilor de căutare, au fost dezvoltați un număr mare de algoritmi pentru o căutare cât mai eficientă.

Algoritmii se pot grupa după două aspecte:

- construcția unui arbore de căutare;
- izolarea şi căutarea doar într-o parte a arborelui de căutare.

Cei mai mulți algoritmi de acest tip au caracter euristic. O regulă euristică oferă o metodă de rezolvare a problemelor, sau o metodă de căutare. Ea nu dă rezultate corecte la orice moment de timp şi nu este garantată că găseşte cea mai bună soluție, dar în acelaşi timp poate reduce timpul de căutare. Metodele de acest tip pot fi folosite pentru reducerea mărimii arborilor de căutare în cazurile arborilor foarte mari.

Metodele relaționale sunt metodele prin care cunoaşterea este reprezentată pornind de la relațiile între obiecte sub formă de grafuri şi rețele.

Pornind de la o structură de concepte care poate fi reprezentată arborescent, înaintarea spre frunză reprezintă o specializare, pe când apropierea de rădăcină este o generalizare a conceptului.

5.1. Căutarea binară

Căutarea binară se foloseşte la regăsirea unui element într-un tablou ordonat. Folosind tehnica *divide et impera*, ce presupune împărțirea unei probleme în subprobleme care acceptă acelaşi tip de rezolvare, căutarea binară împarte tabloul original în două. În funcție de valoarea elementului de căutat se selectează unul dintre cele două jumătăți ale tabloului (care conține valoarea căutată). Se repetă procedeul până se găseşte elementul căutat sau dimensiunea subşirului este 1 (ceea ce înseamnă ca elementul nu a fost găsit).

În continuare este prezentată o implementare recursivă a căutării binare.

Căutare binară – implementare în limbajul Java (MainPrg.java)

```
1  package mainprg;
2
3  import java.util.Arrays;
4  import java.util.Scanner;
5
6  public class MainPrg
7  {
8
```

```
 9    public static int cautareBinara(int[] v, int x,
10                                    int startPoz, int stopPoz)
11    {
12      if (startPoz<=stopPoz)
13      {
14        int k = (startPoz+stopPoz)/2;
15
16        if (v[k] == x)
17          return k;
18        else
19          if (x<v[k])
20            return cautareBinara(v, x, startPoz, k-1);
21          else
22            return cautareBinara(v, x, k+1, stopPoz);
23      }
24
25      return -1;
26    }
27
28    public static void main(String[] args)
29    {
30      System.out.println("..:: CAUTAREA BINARA ::..");
31
32      Scanner keyb = new Scanner(System.in);
33      int n;
34      int[] v = null;
35
36      System.out.print("Introduceti numarul de numere din tablou: ");
37      n = keyb.nextInt();
38      v = new int[n];
39
40      for (int i=0;i<n;i++)
41      {
42        System.out.print("Numarul [" + (i+1) + "]: ");
43        v[i] = keyb.nextInt();
44      }
45
46      Arrays.sort(v);
47
48      System.out.print("Introduceti numarul cautat: ");
49      int x = keyb.nextInt();
50
51      int poz = cautareBinara(v,x,0,v.length-1);
52
53      System.out.print("Numarul cautat se afla pe pozitia: " + poz);
54    }
55  }
```

Realizați varianta iterativă a căutării binare într-un program C/C++ sau Java.

6. Algoritmi de sortare

În practica de zi cu zi este des necesar ca un volum de date să fie aranjat într-o anumită ordine (să fie sortat).

În acest scop au fost creați algoritmi de sortare precum:

- BubbleSort (metoda bulelor);
- SelectionSort (sortare prin selecție);
- QuickSort (sortare rapidă), ş.a.

În continuare sunt prezentați doi algoritmi: bubblesort (algoritm de tip *greedy*) şi quicksort (ce utilizează tehnica *divide et impera*).

6.1. Metoda bulelor (Bubblesort)

Ideea generală a acestui algoritm este de a se parcurge lista ce necesită a fi sortată şi de a compara elementele două câte două, interschimbându-le dacă nu sunt în ordinea corectă.

Algoritmul funcționează în modul următor:

- se pleacă de la ipoteza că şirul de sortat este sortat;
- se parcurge şirul şi se verifică dacă fiecare două elemente vecine sunt în ordinea corespunzătoare; dacă se găseşte o neconcordanță cele două elemente în cauză sunt interschimbate şi se memorează faptul că şirul nu era sortat;
- odată ajunşi la sfârşitul şirului, dacă şirul s-a dovedit a nu fi sortat, se reia procedeul; dacă nu a fost semnalată nicio neconcordanță, atunci orice element a_k este în relație corectă cu vecinul său a_{k+1},prin urmare şirul este sortat.

Acest algoritm, deşi simplu, este foarte lent, necesitând multe iterații pentru obținerea soluției.

Implementarea algoritmului de sortare Bubblesort într-un program C/C++

```
 1  #include <conio.h>
 2  #include <iostream.h>
 3
 4  void main()
 5  {
 6    int TAB[100], // elementele ce vor fi sortate
 7        n, // numarul de elemente: este citit de la tastatura
 8        i=0, // variabila folosita pentru ciclari, pe post de contor
 9        sortat; // varibila logica ce va semnaliza daca tabloul este sortat
10
11    // se citeste numarul de elemente
12    cout<<"Numarul de elemente ale vectorului: ";
13    cin>>n;
```

```
14
15    // se citesc elementele
16    for (;i<n;i++)
17    {
18      cout<<"Elementul "<<i+1<<" este: ";
19      cin>>TAB[i];
20    }
21
22    // SORTARE
23    sortat = 0; // initial vectorul nu este sortat
24    while (!sortat) // cat timp vectorul nu este sortat
25    {
26      sortat = 1; // presupunem ca vectorul este sortat
27      for (i=0;i<n-1;i++)
28        if (TAB[i]>TAB[i+1]) // ordinea elementelor nu este corecta
29        {
30          // se inverseaza elementele care nu au fost in ordine
31          int aux = TAB[i];
32          TAB[i] = TAB[i+1];
33          TAB[i+1] = aux;
34          // se semnaleaza faptul ca vectorul nu a fost ordonat, dupa cum
35          // s-a presupus, deci e posibil ca vectorul sa nu fie inca
36          // sortat, deci mai trebuie verificat inca o data
37          sortat = 0;
38        }
39    }
40
41    // se afiseaza vectorul sortat
42    cout<<"Vectorul sortat este: ";
43    for (i=0;i<n;i++)
44      cout<<TAB[i]<<" ";
45
46    getch();
47  }
```

6.2. Metoda rapidă (Quicksort)

Quicksort efectuează sortarea bazându-se pe o strategie divide et impera. Astfel, algoritmul împarte lista de sortat în două subliste, mai ușor de sortat. Pașii algoritmului sunt:

1. Se alege un element al listei, denumit *pivot*
2. Se reordonează lista astfel încât toate elementele mai mici decât pivotul să fie plasate înaintea pivotului și toate elementele mai mari să fie după pivot. După această partiționare, pivotul se află în poziția sa finală.
3. Se sortează recursiv sublista de elemente mai mici decât pivotul și sublista de elemente mai mari decât pivotul.

O listă de dimensiune 0 sau 1 este considerată sortată.

Implementarea algoritmului de sortare Quciksort într-un program C/C++

```
1  #include <conio.h>
2  #include <iostream.h>
3
```

```
 4  int getFixedPos(int *T, int startpos, int stoppos)
 5  {
 6    int i=startpos, j=stoppos, mod=0; // mod=0 - de la dreapta la stanga
 7                                      // mod=1 - de la stanga la dreapta
 8    while (i<j)
 9    {
10      if (T[i]>T[j])
11      {
12        // inversam elementul T[i] cu T[j]
13        int aux = T[i];
14        T[i] = T[j];
15        T[j] = aux;
16        // schimbam modul de lucru
17        mod = !mod;
18      }
19
20      if (mod)
21        i++;
22      else
23        j--;
24    }
25    return i;
26  }
27
28  void quicksort(int *T, int startpos, int stoppos)
29  {
30    if (startpos<stoppos)
31    {
32      int k = getFixedPos(T,startpos,stoppos);
33      quicksort(T,startpos,k-1);//se aplica acelasi algoritm pentru subsi-
34                                // rul dinaintea pozitiei fixate, k
35      quicksort(T,k+1,stoppos); // se aplica acelasi algoritm pentru subsi-
36                                // rul de dupa pozitia fixata, k
37    }
38  }
39
40  void main()
41  {
42    int TAB[100], // elementele ce vor fi sortate
43        n, // numarul de elemente: este citit de la tastatura
44        i=0, // variabila folosita pentru ciclari, pe post de contor
45        k; // pozitia elementului "fixat"
46
47    // se citeste numarul de elemente
48    cout<<"Numarul de elemente ale vectorului: ";
49    cin>>n;
50    // se citesc elementele
51    for (;i<n;i++)
52    {
53      cout<<"Elementul "<<i+1<<" este: ";
54      cin>>TAB[i];
55    }
56
57    // SORTARE
58    quicksort(TAB,0,n-1);
59
60    // se afiseaza vectorul sortat
61    cout<<"Vectorul sortat este: ";
62    for (i=0;i<n;i++)
63      cout<<TAB[i]<<" ";
64
65    getch();
66  }
```

1. Modificați programele de mai sus pentru a afişa numărul de interschimbări efectuate.
2. Realizați o comparație între numărul de interschimbări realizate de fiecare algoritm prezentat pentru un şir de numere sortat crescător.
3. Realizați o comparație între numărul de interschimbări realizate de fiecare algoritm prezentat pentru un şir de numere sortat descrescător.

7. Grafuri

În fața unui mare număr de situații, o veche obişnuință ne îndeamnă să trasăm pe hârtie puncte reprezentând indivizi, localități, corpuri chimice şi altele, legate între ele prin linii sau prin săgeți care simbolizează o anumită relație. Aceste scheme se întâlnesc peste tot şi fără îndoială D. König a fost primul care a propus ca astfel de scheme să se numească *grafuri* şi care le-a studiat sistematic proprietățile.

Printr-un *graf* se înțelege o mulțime de noduri (numite şi vârfuri) şi o aplicație definită pe această mulțime cu valori în aceeaşi mulțime, care face legătura între aceste noduri, legături numite *arce*, care pot fi sau nu orientate.

O *cale* este o succesiune de noduri aleasă astfel încât să existe arce care să reunească nodurile respective. O cale este simplă dacă toate nodurile, cu excepția primului şi ultimului, sunt distincte între ele.

În multe din problemele la a căror rezolvare se foloseşte calculatorul este necesară reprezentarea unor relații generale între obiecte. Un model potrivit în astfel de cazuri este graful orientat (dacă relația este nesimetrică) sau neorientat (dacă relația este simetrică)

Un *graf orientat* sau *digraf* (prescurtare de la directed graf) G=(V,E) constă deci într-o mulțime V de vârfuri (sau noduri) şi o mulțime E de arce. Un arc poate fi privit ca o pereche ordonată de vârfuri (v,w) unde v este baza arcului iar w este vârful arcului. Se spune că w este adiacent lui v.

Un graf neorientat sau graf G=(N,R) este alcătuit dintr-o mulțime de noduri N şi o mulțime R de muchii. O muchie este atunci o pereche ordonată de noduri (v,w)=(w,v).

Un graf este echivalent cu un digraf în care pentru fiecare arc (v,w) există şi perechea lui (w,v).

Se spune că o cale (succesiune de vârfuri) v[1],v[2],....,v[k] conectează v[1] şi v[k].

7.1. Moduri de reprezentare

Pentru grafuri există două moduri de reprezentare mai des utilizate:

- matricea de adiacențe şi
- listele de adiacențe.

Alegerea uneia dintre ele trebuie fi făcută în funcție de frecvența operațiilor de acces la nodurile şi muchiile grafurilor.

Dat fiind G=(V,E) să considerăm mulțimea vârfurilor V={1,2,3,...n} având elementele în ordinea naturală a numerelor prin care sunt reprezentate. Matricea de adiacențe *A* de dimensiune n × n se poate defini prin:

```
A[i,j] =    1 dacă [i,j] aparține lui E
            0 dacă [i,j] nu aparține lui E
```

Reprezentarea prin matrice de adiacențe permite un acces rapid la arcele (muchiile) grafului fiind utilă în algoritmii în care se testează prezența sau absența unui arc oarecare. Ea este dezavantajoasă dacă numărul de arce este mai mic decât n × n, caz în care memoria necesară pentru a păstra matricea este folosită ineficient.

Reprezentarea prin liste de adiacențe foloseşte mai bine memoria, dar determină o căutare mai anevoioasă a arcelor. În această reprezentare, pentru fiecare nod se păstrează lista arcelor către nodurile adiacente. Întregul arbore poate fi reprezentat ca un tablou *Cap*, indexat după noduri, fiecare element *Cap[I]* fiind un pointer spre lista nodurilor adiacente lui *i*. Memoria necesară reprezentării este proporțională cu suma dintre numărul de noduri şi numărul de arce ale grafului.

Scrieți un program C/C++ sau Java care citeşte de la tastatură un graf şi îl memorează folosind reprezentarea prin matricea de adiacență.

7.2. Algoritmi pentru minimizare căi

Dându-se un digraf G=(V,E), în care fiecare arc are ca etichetă un număr ne-negativ (costul său), un vârf este considerat ***sursă***, iar altul ***destinatar***. Problema constă în determinarea căii de cost minim de la sursă la destinatar.

7.2.1. Dijkstra

Rezolvarea acestei probleme se bazează pe o tehnică "greedy" datorată lui E.W. Dijkstra. Ea constă în păstrarea unei mulțimi ***Selectate*** de vârfuri ale căror distanțe minime față de ***sursă*** sunt cunoscute. Inițial, ***Selectate*** conține doar vârful sursă; la fiecare pas, se adaugă la ***Selectate*** un vârf a cărui distanță față de un vârf din ***Selectate*** este minimă. În rezolvare se utilizează:

- un tablou ***Distanță*** al distanțelor minime de la sursă la fiecare vârf;
- matrice ***Cost*** de costuri, în care Cost [i, j] este costul asociat arcului (i,j); dacă nu există un arc (i,j), atunci se consideră pentru Cost [i, j] o valoare *infinit* (practic, foarte mare).

Algoritmul Dijkstra

```
Algoritm GăseşteCăiMinime(sursă)
{
  Selectate ← {sursa}
  pentru toate vârfurile i de la 1 la n
    execută
    {
      Distanţe [ i ] ← Cost [ sursa, i ]
      pentru toate vârfurile i de la 1 la n-1
        execută
        {
          găseşte vârful K neselectat cu Distanţe [ K ] minim
          adaugă K la Selectate
          pentru fiecare vârf j neselectat
            execută
            Distanţe [ j ] ← min ( Distanţe [ j ],
                                   Distanţe [ K ] + Cost [ K, j ] )
        }
    }
}
```

Realizaţi implementarea algoritmului Dijkstra într-un program C/C++ sau Java.

7.2.2. Floyd

Algoritmul de aflare a căilor minime dintr-un punct poate fi repetat luând ca sursă fiecare din nodurile unui graf. Aceasta permite calculul unui tablou al drumurilor minime între toate perechile de noduri ale grafului. O astfel de tehnică este cea datorată lui R. W. Floyd.

Implementarea algoritmului Floyd în C/C++:

```
 1  int floyd(int *A)
 2  {
 3    int k, i, j;
 4
 5    for (k = 1; k <= n; k++)
 6      for (i = 1; i <= n; i++)
 7        for (j = 1; j <= n; j++)
 8          if (A[i][j] > (A[i][k] + A[k][j]))
 9            A[i,j] = A[i][k] + A[k][j];
10  }
```

Această tehnică se bazează pe utilizarea unui tablou *A* al distanţelor minime, ale cărui valori sunt calculate în mai multe etape. Iniţial,

```
A[i,j] =    Cost[i,j] pentru orice i<>j
            0 pentru i = j
            INFINIT (practic valoare foarte mare) dacă nu există arcul (i,j)
```

Calculul distanțelor minime se face în *n* iterații. La iterația *k*, *A*[*i,j*] va avea ca valoare cea mai mică distanță între *i* şi *j*, pe căi care nu conțin vârfuri numerotate peste *k* (exceptând capetele *i* şi *j*).

7.2.3. Arborele de acoperire minim – algoritmul Kruskal

O metodă utilizată în calcule este aceea a arborelui minim de acoperire (*Kruskal*). Algoritmul construieşte treptat mulțimea T a muchilor arborelui minimal adăugând la fiecare pas muchia care nu formează cicluri cu muchiile aflate deja în T.

Algoritmul Kruskal

```
T←{ }
cât timp T nu este arbore de acoperire
  execută
  {
    selectează muchia (w,u) de cost minim din R
    şterge (w,u) din R
    dacă (w,u) nu creează un ciclu în T
     atunci adaugă (w,u) la T
  }
```

O posibilă implementare a algoritmului în limbajul de programare C/C++ este redată mai jos.

Implementarea algoritmului Kruskal în limbajul C/C++

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

void printArray(int a[][100],int n)
{
  int i,j;
  for(i = 0; i < n;i++)
  {
    for(j = 0; j < n;j++)
    {
      printf("%d\t",a[i][j]);
    }
    printf("\n");
  }
}

void GenereazaMatriceaDeAdiacenta(int a[][100], int n)
{
  int i,j;

  for(i = 0;i < n; i++)
  {
    for(j = 0;j < i; j++)
    {
      a[i][j] = a[j][i]= rand()%50;

```

```
27        if(a[i][j]>40)
28        {
29          a[i][j]=a[j][i]=999;
30        }
31      }
32      a[i][i] = 999;
33    }
34    printArray(a,n);
35  }
36
37  int root(int v,int p[])
38  {
39    while(p[v] != v)
40    {
41      v = p[v];
42    }
43    return v;
44  }
45
46  void union_ij(int i,int j,int p[])
47  {
48    if(j > i)
49      p[j] = i;
50    else
51      p[i] = j;
52  }
53
54  void kruskal(int a[][100],int n)
55  {
56    int count, i, p[100], min, j, u, v, k, t[100][100], sum;
57    count = k = sum = 0;
58    for(i = 0; i < n; i++)
59    {
60      p[i] = i;
61    }
62    while(count < n)
63    {
64      min = 999;
65      for(i = 0; i < n; i++)
66      {
67        for(j = 0; j < n; j++)
68        {
69          if(a[i][j] < min)
70          {
71            min = a[i][j];
72            u = i;
73            v = j;
74          }
75        }
76      }
```

```
77      if(min != 999)
78      {
79        i = root(u, p);
80        j = root(v, p);
81        if (i != j)
82        {
83          t[k][0] = u;
84          t[k][1] = v;
85
86          k++;
87
88          sum += min;
89          union_ij(i,j,p);
90        }
91        a[u][v] = a[v][u] = 999;
92      }
93      count++;
94    }
95
96    if(count != n)
97    {
98      printf("Arborele de acoperire minim nu exista!\n");
99    }
100   else
101   {
102     printf("Arborele de acoperire minim este:\n");
103     for(k = 0; k < n-1 ; k++)
104     {
105       printf(" %d -> %d ",t[k][0],t[k][1]);
106     }
107     printf("\nCost = %d \n",sum);
108   }
109 }
110
111 void main()
112 {
113     int a[100][100],n;
114     printf("Introduceti numarul de noduri: ");
115     scanf("%d",&n);
116     GenereazaMatriceaDeAdiacenta(a,n);
117     kruskal(a,n);
118 }
```

Introduceți mesaje de afişare în programul de mai sus pentru a evidenția paşii parcurși în obținerea arborelui de acoperire de cost minim.

7.2.4. Algoritmul Prim pentru aflarea arborelui parţial de cost minim

Un alt algoritm greedy pentru determinarea arborelui parțial de cost minim ale unui graf se datorează lui **Prim** (1957). Deosebirea constă în faptul că, la fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U a vârfurilor pe care le conectează formează un subarbore de cost minim pentru subgraful (U,A) al lui G (şi nu o pădure ca în algoritmul lui Kruskal). Inițial, mulțimea U a vârfurilor acestui arbore conține un singur vârf oarecare din V, care va fi rădăcina, iar mulțimea A a muchiilor este vidă. La fiecare pas se alege o muchie de cost minim, care se adaugă la arborele precedent dând naştere la un nou subarbore de cost minim (deci exact una din extremitățile ale acestei muchii este un vârf în arborele precedent). Arborele de cost minim creşte "natural", cu câte o ramură, până când va atinge toate vârfurile din V, adică până când U = V.

În algoritmul lui Prim, la fiecare pas, (U,A) formează un arbore parțial de cost minim pentru subgraful (U,A) a lui G. În final se obține arborele parțial de cost minim al grafului G.

Descrierea formală a algoritmului este redată mai jos.

Algoritmul Prim

```
Prim (G =(V,M))
{inițializare}
A ← 0 {va conține muchiile arborelui parțial de cost minim}
U ← {un vârf oarecare din V}
{Buclă greedy}
cât timp U ≠ V
execută
      găseşte {u,v} de cost minim astfel ca U ∈ V \ U şi v ∈ U
      A ← A ∪ {{ u , v }}
      U ← U ∪ { u }
returnează A
```

Realizați implementarea algoritmului Prim într-un program C/C++ sau Java.