Ministerul Educatiei Nationale
Universitatea “1 Decembrie 1918” Alba Iulia
Catedra de Matematica, Informatica, Topografie

Ovidiu Domsa

Programarea calculatoarelor

Curs

Editia a III-a

Alba Iulia

2003

Motto:

“...în toate sa ai în vedere sfârsitul...”

Solon

I. Introducere

Rezolvarea problemelor cu ajutorul calculatorului reprezinta astazi nu numai o necesitate obiectiva ci un mod de supravietuire, în conditiile în care, volumul de informatii si viteza cu care ele se deruleaza sunt din ce în ce mai mari.

Este de înteles faptul ca nu oricine poate sa programeze un calculator astfel încât acesta sa "faca ce vreau eu". Pentru acest lucru sunt necesare, pe lânga bune cunostinte de matematica, logica si informatica, abilitati si aptitudini de a aborda problemele în mod specific. Programarea calculatoarelor este de fapt o arta, care, ca orice lucru de cultura se cultiva, se dezvolta si se perfectioneaza continuu.

Acest curs are rolul de a fundamenta si de a dezvolta cunostintele dobândite si cultivate în cadrul cursului de elaborare a algoritmilor. Se urmareste modelarea unor principii ale programarii calculatoarelor, dezvoltarea si însusirea unor metode si tehnici specifice prelucrarii anumitor tipuri de date respectiv anumitor tipuri de structuri de date si clase de probleme, împreuna cu formarea uni stil în programare apropiat de cerintele practice ale constructiei de soft.

Principalele aspecte sunt legate de metodele clasice si evoluate de sortare, cautare, interclasare si parcurgere pe clase si structuri de date specifice. Unele dintre acestea au fost prezentate pe scurt si în cursul de algoritmica si care trebuie revizuite cu atentie si aplicate corect în noile situatii. Metodele de programare evoluate tratate în acest curs deschid calea spre rezolvarea optima a unor clase de probleme.

Prelucrarea dinamica a datelor prin reprezentarea lor ca liste este tratata prin exemple si modele de lucru. Prin varietatea de probleme, însotite de solutii, prezentate în cadrul metodelor de rezolvare a problemelor si completarea lor cu alte metode, programarea dinamica, tehnica branch and bound si algoritmi specifici grafelor si arborilor, cursul încearca sa contureze principalele metode si tehnici cunoscute, pe care un programator trebuie sa le stapâneasca pentru a putea implementa produse soft de buna calitate.

Recomand în studiul acestui curs utilizarea frecventa a documentatiilor mentionate în bibliografie si bineînteles a calculatorului ca unealta de lucru.

Alba Iulia, 2003

Autorul

II. Principiile programarii calculatoarelor

Modalitatile de rezolvare a unei probleme sunt diverse si depind atât de tipul problemei cât si de programator. În general programarea respecta principiile de programare, respectiv programarea structurata, programarea modulara, programarea orientata obiect, etc. functie de specificul problemei.

Dezvoltarea de software este un lucru mult mai complex si mai complicat. Acesta presupune o serie întreaga de activitati distincte în care programarea ocupa un loc bine definit si se integreaza în lista activitatilor. Aceste activitati cuprind:

- Definirea problemei
- Analiza cerintelor
- Construirea planului de implementare
- Definirea detaliilor si a structurii problemei
- Constructia, respectiv implementarea
- Integrarea modulelor
- Testarea modulelor
- Testarea sistemului în ansamblu
- Întretinerea corectiva
- Dezvoltarea functionala

Intuitiv toate aceste activitati sunt privite în general ca „PROGRAMARE”, delimitând aici partea de constructie (design si structura), de transcriere sub forma de cod a activitatilor (implementare, codificare) si respectiv ansamblul etapelor de testare si întretinere.

Pornind de la idea ca rezolvarea unei probleme presupune cunoasterea în totalitate a cerintelor si datelor de iesire, descrierea corecta a pasilor algoritmilor, forma si metodele de lucru utilizate de catre programator apartin exclusiv acestuia. Programarea industriala presupune însa elaborarea unor coduri dupa specificatii exacte ale altor programatori care au realizat etapa de constructie si arhitectura a programului.

Descrierea programelor, indiferent de limbajul de programare utilizat presupune respectarea unor principii atât în ceea ce priveste descrierea datelor cât si a pasilor de rezolvare. Aceste principii se impun din cel putin urmatoarele motive:

- **Lizibilitatea programului**; aspect care presupune aranjarea liniilor de program astfel încât la citirea, parcurgerea si verificarea acestuia sa se delimiteze clar secventele, structurile si instructiunile cu modul lor de imbricare si succesiune. În acest sens fiecare instructiune este scrisa pe câte un rând, instructiunile imbricate se scriu aliniate mai la dreapta fata de instructiunea “parinte”. Nu se pune problema corectitudinii algoritmului, el fiind considerat a fi corect.

- **Depanarea programului**; scrierea programelor pe module mici, în secvente scurte si clare duce la depistarea usoara a eventualelor erori si totodata la posibilitatea de a verifica usor corectitudinea secventelor, deci principiul urmator:
- **Testarea pe module mici** presupune testarea programelor îndata ce au fost elaborate. Fiecare linie noua de cod adaugata va trebuie testata pentru a vedea corectitudinea si situatiile noi generate de aceasta;
- **Portabilitatea**; reprezinta posibilitatea de a implementa programul pe diverse sisteme si în diverse configuratii, eventualele modificari sa permita flexibilitatea modulelor.
- **Lucrul în echipa**; se cunosc doua principii de programare în echipa:

A) *"principiul cutiei negre"* care reprezinta totala independenta a programatorului în interiorul programului sau, legatura cu exteriorul facându-se strict pe baza formatului datelor de intrare si a celor de iesire. Sub forma

DATE DE INTRARE [redacted] *DATE DE IESIRE*

B) "principul cutie transparente", un principiu modern prin care se impune respectarea unor reguli interne deci a unui standard vizibil si din exteriorul programului.

DATE DE INTRARE | *program* | *DATE DE IESIRE*

Respectarea unor principii, reguli, în programarea calculatoarelor presupune învatarea corecta, în etape, prin pasi marunti a metodelor de rezolvare a problemelor si respectarea unor principii date de experienta în programare a numerosi programatori.

Selectam, din cartile de specialitate, unele dintre cele mai frecvent întâlnite sfaturi si principii ale programarii care va îndruma spre ceea ce trebuie si ceea ce nu trebuie facut atunci când doriti sa elaborati un program:

1) **Defineste complet problema** – definirea completa a unei probleme presupune cunoasterea exacta si în cele mai mici detalii a datelor de intrare, a cerintelor problemei precum si cunoasterea si definirea corecta tuturor notiunilor care intervin în problema.
2) **Proiecteaza structurat algoritmii** – se elaboreaza algoritmii având în vedere secventialitatea operatiilor, posibilitatea de repetitie sau de decizie în fiecare etapa.
3) **Foloseste algoritmii existenti si tehnica programarii modulare** – împartirea unei probleme în subprobleme si rezolvarea acestora folosind algoritmi clasici, cu performante deja demonstrate, usureaza munca de

programare si reduce posibilitatea de eroare în algoritmi. Utilizarea unor algoritmi neconventionali si aglomerarea problemelor într-una singura duce la confuzii si în general la respingerea programelor astfel elaborate.

4) **Foloseste proiectarea orientata pe obiecte** – acest mod de proiectare permite flexibilitatea atât la nivelul datelor cât si al procedurilor.

5) **Gândeste mai întâi, programeaza pe urma** – în general timpul alocat elaborarii strategiei de rezolvare a unei probleme este cel putin dublu fata de timpul necesar implementarii.

6) **Detaliile nesemnificative sunt semnificative**;

7) **Foloseste comentariile în textul sursa si utilizeaza identificatori sugestivi** care sa permita si altor programatori sa interpreteze programul sursa.

8) **Asigura portabilitatea programului** – prin evitarea folosirii specificului calculatorului, cum ar fi elemente de afisaj specific (rezolutii, placi grafice, tipuri de interfete), transmisie de date (retele locale, unitati de disc specifice), dispozitive periferice (tipuri de imprimante) si respectiv medii de programare sau sisteme de operare.

9) **Verifica corectitudinea algoritmului si a programului în fiecare etapa a elaborarii.**

10) **Elaboreaza documentatia programului odata cu elaborarea sa.**

Pe lânga aceste principii se recomanda câteva aspecte care tin de prelucrarea datelor în cadrul unui algoritm si care sunt utile mai ales celor care fac primii pasi în elaborarea programelor:

- Foloseste toate variabilele pe care le-ai definit
- Cunoaste si respecta tipul si semnificatia fiecarei variabile
- Initializeaza variabilele, fie prin citirea lor fie prin atribuirea unei valori initiale.
- Verifica valorile variabilelor imediat dupa calculul lor.
- Indiferent de metodele de programare utilizate evita sa folosesti variabile globale. Fiecare modul este bine definit de variabilele sale locale iar transmiterea valorilor se realizeaza prin lista de parametrii.
- Evita artificiile
- Testeaza programul chiar daca ai demonstrat corectitudinea sa pentru mai multe seturi de date de intrare.
- Foloseste facilitatile de depanare puse la dispozitie de mediul de programare (vizualizarea valorilor variabilelor, a stivei de lucru, a pasilor în executia unui program)

III. Structuri de date

1. Notiuni si concepte preliminarii

Limbajele de programare dispun de modalitati de agregare a datelor care permit apoi tratarea globala a acestora. Este vorba în general de date care corespund nivelului de abstractizare al limbajului, deci care nu au corespondent direct în tipurile masina. Pentru ca aceste date definite de utilizator conform nevoilor sale concrete sa poata fi integrate în mecanismul de tipuri al limbajului, acesta din urma pune la dispozitia programatorului constructorii de tipuri. în acest capitol se vor discuta tipurile de date structurate.

Tabelul de mai jos prezinta o clasificare a tipurilor de date împreuna cu modalitatile de organizare a acestora.

Structura	**Tip de date**	**Numar Componente**	**Tip componente**	**Mod de organizare**
A) simple	Numeric	Una	Numeric	Static
	Alfanumeric	Una	Caracter	Static
	Logic	Una	Logic	Static
B)structurate	Tablou	Mai multe	Acelasi tip	Static
	Articol	Mai multe	Tipuri diferite	Static/Dinamic
	Fisier	Mai multe	Acelasi tip	Static
	Liste	Mai multe	Acelasi tip	Static/Dinamic
	Pointer	Mai multe	Tipuri diferite	Dinamic

Spre deosebire de datele simple, care sunt atomice, indivizibile, datele structurate (compuse, agregate) se descompun în componente sau elemente, fiecare de un tip precizat (simplu sau structurat). O data structurata poate fi accesata fie ca întreg (global), fie pe componente. Structura unei date stabileste relatiile care exista între componentele acesteia.

Exista patru tipuri de legaturi structurale fundamentale:

- **multime** (nici o legatura între componente), fisier
- **liniara** (legatura 1:1), tablou unidimensional (vector), liste liniare, articol
- **arbore** (legatura 1:n), liste dublu înlantuite
- **graf** (legatura m : n), tablouri bidimensionale (mxn).

Din punctul de vedere al uniformitatii structurale, datele structurate se împart în:

- **omogene** (toate componentele au acelasi tip); tipurile de date aferente sunt numite tablou (engl. *array*), multime (engl. *set*) si fisier (engl. *file*);

- **heterogene** (elementele unei date au de obicei componente diferite ca tip); ele apartin tipului de date înregistrare (engl. *record*).

Tablourile, fisierele si înregistrarile au structura liniara: exista o prima si o ultima componenta, iar toate celelalte au fiecare atât predecesor, cât si succesor. Prin urmare, un element (al tabloului), o înregistrare (din fisier) sau un câmp (al înregistrarii) se pot localiza. Un tablou este un agregat de elemente de acelasi tip, un element fiind localizat prin pozitia pe care o ocupa în cadrul acestuia (indicele elementului de tablou). Un fisier este constituit si el din elemente (înregistrari) de acelasi tip, localizate tot dupa pozitia ocupata în fisier. Deosebirea dintre un tablou si un fisier consta în aceea ca tabloul este memorat în memoria interna a calculatorului, iar fisierul este memorat în memoria externa (pe un suport magnetic sau magneto-optic). O înregistrare este un agregat care grupeaza de obicei elemente de tipuri diferite numite câmpuri si localizate prin numele lor.

Multimea are o structura amorfa: ea contine elemente de acelasi tip, care însa nu pot fi localizate explicit, neexistând o ordine în care sa fie considerate elementele din ea.

2. Tipuri de date structurate. Definitii si clasificari.

2.1. Tablouri

Tablourile se folosesc pentru a grupa variabile de tipuri identice si a le manipula prin operatii. daca fiecare variabila ar fi declarata individual, atunci fiecare operatie ar trebui specificata separat, fapt care ar duce la programe lungi.

Un tablou ne permite sa grupam variabilele de acelasi tip sub un singur nume si sa putem referi fiecare variabila (numita element al tabloului) asociind numelui un indice (care o va identifica unic). Prin aceasta se reduce considerabil dimensiunea unui program care efectueaza operatii similare asupra mai multor elemente din tablou, folosind indicii si instructiunile de ciclare.

In Pascal, declaratia de tip tablou are sintaxa:

Type tip_tablou = Array[tip_index] Of tip_element;

unde:

- Array si Of sunt cuvinte rezervate

- tip_element, numit tipul componentei, poate fi orice tip recunoscut de sistemul de tipuri

- tip_index este o lista de tipuri de indici; numarul elementelor din aceasta lista denota numarul de dimensiuni al tabloului, care nu este limitat.

Tipurile indicilor trebuie sa fie ordinale (cu exceptia lui Longint si a subdomeniilor de Longint). Daca tipul componentei este tot un tip tablou, rezultatul acestei declaratii de tip poate fi tratat fie ca un tablou de tablouri, fie ca un tablou mono dimensional. De exemplu, a doua declaratie:

type D = 20..30;
type T = array[boolean] of array[1..10] of array[D] of real;

este identica (structural) cu declaratia:

type T1 = array[boolean, 1..10, D] of real;

iar referiri corecte de elemente sunt:

```
var
  A: T;
 A1: T1;

A[ false] s i A1[ false] sunt de tip array[ 1..10] of array[ D] of real;
A[ false,5] s i A1[ false,5] sunt de tip array[ D] of real;
A[ false][ 5] si A1[ false][ 5] sunt de tip array[ D] of real;
A[ false,5,30] si A1[ false,5,30] sunt de tip real;
A[ false][ 5][ 30] si A1[ false][ 5]30] sunt de tip real;
```

Declararea de variabile de tip tablou nu trebuie neaparat sa contina numele unui tip de date tablou. Constructorul de tip se poate include în declaratia de variabila. Astfel,

```
Var
  V: Array[ 1..10] Of Integer;
```

este o declaratie valida de variabila. în aceasta declaratie de variabila, numele tipului tablou este referit chiar prin constructorul de tip, adica

```
Array[ 1..10] Of Integer;
```

Numarul de elemente al unui tablou este dat de produsul numarului de elemente în fiecare dimensiune.

Accesarea (referirea) unui element de tablou se face prin precizarea numelui tabloului urmat de o expresie de indice. Expresia de indice contine, în paranteze drepte valorile efective ale indicilor tabloului (ce trebuie, de obicei, sa concorde ca numar si tip cu declararea acestuia). Exista în general doua modalitati de referire:

- punctuala
- de subtablouri.

Ambele modalitati se bazeaza pe calculul de adresa.

Pentru fiecare tablou declarat, se memoreaza în descriptorul de tablou urmatoarele informatii:

- nume = numele tabloului;
- tip = tipul elementului de tablou;
- lung = lungimea reprezentarii unui element de tablou (în unitati de alocare);
- adrs = adresa de unde începe memorarea tabloului;
- nrd = numarul de dimensiuni al tabloului;
- pentru fiecare dimensiune i, limitele $linf_i$ si $lsup_i$ (i=1,nrd)

Tablourile bidimensionale (numite si matrice în limbajul curent) au linii si coloane. Am vazut în paragraful precedent cum se memoreaza aceste tablouri. Dam aici un exemplu care foloseste doua modalitati diferite (structural echivalente) de declarare a tablourilor bidimensionale.

Exemplu:

```
Type
  DomeniuLinii = 1..25;
  DomeniuColoane = 1..80;
  Element = Record
    Car: Char;
    Atr: Byte
  End;
  TabEcran1 = Array[ DomeniuLinii, DomeniuColoane] Of Element;
  TabEcran2 = Array[ DomeniuLinii] Of Array[ DomeniuColoane] Of
          Element;

Var
  EcranColor1: TabEcran1 Absolute $B800:0000;
  EcranMono1: TabEcran1 Absolute $B000:0000;
  EcranColor2: TabEcran2 Absolute $B800:0000;
  EcranMono2: TabEcran2 Absolute $B000:0000;

Begin
  EcranColor1[ 12, 33].Car := 'A';
  Write(EcranColor2[ 12][ 33].Car);
  EcranMono2[ 11][ 25].Car := 'B';
  Write(EcranMono1[ 11,25].Car);
End.
```

Prima modalitate de declarare permite accesarea

- punctuala: EcranColor1[12, 33]
- globala: EcranColor1

A doua modalitate de declarare a tabloului permite accesarea
- punctuala: EcranColor2[12, 33] sau EcranColor2[12][33]
- a unei linii: EcranColor2[12] (care va fi de tipul Array[DomeniuColoane] Of Element)
- globala: EcranColor2

Tablourile multidimensionale au mai mult de doua dimensiuni. Limbajul Pascal nu impune o regula privind numarul maxim de dimensiuni al unui tablou. Trebuie avuta în vedere doar restrictia ca spatiul alocat unei variabile sa nu depaseasca dimensiunea segmentului de date sau de stiva.

Operatii globale pe tablouri

Operatiile definite pe tipul tablou sunt atribuirea si testul de egalitate. De asemenea, variabilele de tip tablou se pot transmite ca parametrii în subprograme.

2.2. Multime

Limbajul Pascal este unul din primele limbaje care a introdus tipul multime. Fiind dat un tip ordinal B, numit tip de baza, tipul multime T se declara folosind sintaxa:

Type tip_multime = Set Of tip_de_baza;

Domeniul tipului tip_multime T este multimea partilor (submultimilor) domeniului tipului tip_de_baza. Astfel, daca tipul B are domeniul {a, b, c}, atunci domeniul lui T va fi

{{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}.

Tipul T se considera structurat deoarece fiecare element al sau contine elemente (posibil nici unul) din domeniul tipului de baza B.

Implementarea tipului multime se face pe siruri de biti. Daca domeniul tipului de baza B are n elemente, atunci domeniul tipului multime T va avea 2^n (2 la puterea n) elemente, deci o variabila de tip T se poate reprezenta pe un sir de n biti. Bitul de pe pozitia i din sirul respectiv (1<=i<=n) va fi setat pe 1 daca al i-lea element din domeniul lui B (ordinal) apartine multimii si 0 în caz contrar. Operatiile tipului multime se implementeaza eficient prin operatii pe siruri de biti. în cazul limbajului Pascal, n = 256.

Multimile reprezinta un instrument matematic elegant. Prezenta lor în limbajele de programare îmbogateste expresivitatea acestuia. Din pacate, limitarile impuse asupra domeniului tipului de baza (numar de elemente si tip) restrâng utilizarea

multimilor. în astfel de situatii, utilizatorul poate sa-si defineasca propriile sale tipuri multime.

Înainte de a opera cu variabile de tip multime, ele trebuie construite. Regulile de construire a variabilelor de tip multime în Pascal sunt urmatoarele:

- o multime Pascal este formata dintr-o lista de elemente separate prin virgula si incluse în paranteze drepte;
- multimea vida se marcheaza prin [ ];
- toate elementele din lista trebuie sa fie de acela si tip, tipul de baza al multimii
- toate elementele sunt diferite
- în loc sa se enumere toate elementele dintr-un subdomeniu, se poate preciza subdomeniul pe post de element în lista (primul si ultimul element cu **..** între ele)

Ordinea în care se executa operatiile:

- declararea tipului multime T
- declararea variabilei V de tipul T
- construirea (initializarea) variabilei V

Exemple:

```
Type
 Luni = (Ian, Feb, Mar, Apr, Mai, Iun, Iul, Aug, Sep, Oct, Nov, Dec);
 Anotimp = Set Of Luni;
 Caractere = Set Of Char;
Var
 Iarna, Primavara, Vara, Toamna: Anotimp;
 Litere, Cifre: Caractere;

Begin
  Iarna := [ Dec, Ian, Feb, Mar];
  Vara := [ ];
  Toamna := [ Aug, Sep, Oct, Nov];
  Primavara := [ Apr, Mai, Iun, Iul];
  If Vara = [ ] Then WriteLn('Anul asta n-a fost vara!');
  Litere := [ 'A'..'Z', 'a'..'z'];
  Cifre := [ '0'..'9']
End.
```

Compararea multimilor si testul de apartenenta

Operator	Descriere	Exemplu de folosire	Rezultat
=	egalitate de multimi	[Ian, Feb] = [ ]	False
<>	diferit	[Ian, Mar] <> [Apr]	True
<=	incluziunea A <= B	[Ian, Mar] <= [Ian, Aug, Mar]	True
	înseamna ca orice element	[Ian, Mar] <= [Feb]	False
	din A apartine si lui B		
>=	include: A >= B înseamna	[Ian, Aug, Mar] >= [Ian, Apr]	False
	ca orice element din B	[Ian, Mar] >= [Mar]	True
	apartine si lui A		
in	x în A înseamna x	Ian în [Aug, Feb]	False
	apartine lui A	Mar în [Ian, Aug, Mar]	True
	x trebuie sa fie din	'P' în ['A'..'Z']	True
	tipul de baza		

Operatiile proprii tipului multime sunt cele cunoscute:
a) operatii binare cu rezultat multimi
- reuniunea “+”,
- intersectia “*”,
- diferenta “-“

b) operatii binare cu rezultat boolean
- incluziunea , testul de egalitate “=”
- apartenenta IN,

c) atribuirea “:=”.

2.3. Articol (înregistrare)

Elementele definitorii ale unei înregistrari sunt:
- numele tipului înregistrare (optional),
- numarul de câmpuri,
- numele si tipul fiecarui câmp.

Înregistrarea este o modalitate de agregare (punere împreuna a unor date de tipuri (în general) diferite. Numele tipului de data înregistrare este un identificator. Numarul de câmpuri este dedus din lista de declarare a câmpurilor. Câmpurile unei înregistrari se memoreaza în zone adiacente de memorie. Informatia de tip a fiecarui câmp serveste la stabilirea lungimii de reprezentare a acestuia, iar numele câmpului se foloseste pentru a accesa valoarea lui (prin operatia de accesare). De obicei, sunt accesibile atât data compusa (înregistrarea), cât si componentele (câmpurile) acesteia.

Exista doua clase de înregistrari: cu structura fixa si cu variante. Înregistrarile cu structura fixa au o singura definitie, iar înregistrarile cu variante au o parte fixa, un discriminant si mai multe definitii de variante.

Lungimea unei înregistrari este suma lungimilor câmpurilor componente. Se poate folosi *functia SizeOf* cu parametru tipul de data înregistrare sau variabila de tip înregistrare.

Elementele de discutie privitoare la tipul de date înregistrare sunt urmatoarele:

- maniera de declarare a tipurilor înregistrare;
- maniera de accesare a câmpurilor unei înregistrari;
- definirea înregistrarilor cu structura variabila;
- initializarea unei variabile înregistrare;
- operatiile permise pe variabile de tip înregistrare.

Declararea tipului înregistrare

In Pascal, de exemplu, tipurile de date DataC (data calendaristica), Timp si ElementEcran se declara astfel:

```
Type
  DataC = Record
     zi : 1..31;
     lu : 1..12;
     an : 1800..2000;
  End;
  Timp = Record
    ora: 0..23;
    min: 0..59;
    sec: 0..59
  End;
  Element = Record  { definitia unui element pe ecranul text }
    Car: Char;
    Atr: Byte
  End;
```

Se observa ca aceste tipuri de date au componente (câmpuri) diferite ca tip de date. Odata declarate aceste tipuri, se pot declara variabile de tipurile respective:

```
Var
  DataNasterii, DataCasatoriei: DataC;
  OraDeIncepere: Timp;
  Ecran: Array[ 1..25] Of Array[ 1..80] Of ElementEcran;
```

Doar în momentul declaratiei de variabila se poate stabili dimensiunea de alocare pentru variabilele de tipurile respective.

Referirea componentelor sau selectarea (accesarea) câmpurilor unei înregistrari se face în Pascal în doua moduri:

- folosind calificarea cu punct
- folosind instructiunea With

Calificarea cu punct permite accesarea sau modificarea valorii unui câmp dintr-o variabila înregistrare. Ea se face în forma:

nume_variabila.nume_câmp

Variabilele de tip înregistrare se pot accesa si global, prin numele lor.

Este permisa de asemenea operatia de atribuire, când variabilele implicate sunt de acelasi tip:

```
Var
  Data1, Data2: DataC;
-----------------------------------------------------
Atribuire globala          Echivalenta cu
-----------------------------------------------------
Data1 := Data2             Data1.zi := Data2.zi
                           Data1.lu := Data2.lu
                           Data1.an := Data2.an
-----------------------------------------------------
```

Câmpurile unei înregistrari pot fi de orice tip recunoscut de sistemul de tipuri, în particular si tip înregistrare.

```
Type
  DataC = Record
     zi : 1..31;
     lu : 1..12;
     an : 1800..2000;
  End;
  A40 = Array[ 1..40] Of Char;
  TipSex = (masc,fem);
  StareCiv = (necas,casat,vaduv,divortat);
  Persoana = Record
    Nume : String[ 30];
    NrId   : Longint;
    Adr    : A40;
    Sex    : TipSex;
    StareC : StareCiv;
    DataN  : Datac;
    inalt  : Real
  End;
```

Tipurile câmpurilor sunt predefinite (String, LongInt, Real în Turbo Pascal), simple utilizator (TipSex si StareCiv), respectiv structurate (A40 si DataC).

Prin urmare, cu astfel de notatii de tip se pot declara tipuri oricât de complexe. daca se declara o variabila de tip Persoana:

```
Var p : Persoana;
```

ea va putea fi tratata fie global (atribuire, test de egalitate), fie selectiv (p.Inalt va semnifica câmpul inalt din înregistrarea p). Tabelul urmator prezinta exemple de accesare (cu punct si cu With).

```
          With P Do
                Begin
P.Nume   := 'IONESCU';      Nume   := 'IONESCU';
P.Sex    := masc;           Sex    := masc;
P.StareC := casat;          StareC := casat;
P.Inalt  := 1.80;           Inalt  := 1.80;
P.DataN.zi := 23;           DataN.zi := 23
                End
```

Alte exemple de structuri complexe

- înregistrari cu câmpuri tablou
- tablouri de înregistrari

```
Const
  Max_Persoane = 100;
  Max_Discipline = 20;
Type
  Personal: Array[ 1..Max_Persoane] Of Persoana;
  Medie = Record     {facem economie: Medie: Real ocupa 6 bytes }
    Parte_Int: Byte;  {SizeOf(Medie) = 2}
    Parte_Zec: Byte
  End; { Medie }
  Medii = Array[ 1..Max_Discipline] Of Medie;
  Trimestru = 1..3;
  Elev = Record
    Nume: String[ 40];
    DataN: DataC;  {declarat în exemplul de mai sus}
    SitScolara: Array[ Trimestru] Of Medii;
  End; { Elev }
```

2.4. Fisier

Notiunea de fisier sta la baza multor concepte în informatica, cum ar fi sisteme de operare DOS sau WINDOWS etc.

Notiunile de data si nume de data sunt considerate ca notiuni primare si le vom numi câmpuri.

In prelucrarea datelor a aparut necesitatea considerarii unor grupari de date.

Exemplu: *nume, prenume, adresa*

Datele asociate celor trei nume de data, împreuna, constituie un articol (înregistrare).

Exemplu:

Gestiunea marfurilor dintr-un magazin se realizeaza pe baza unor date care contin: denumirea produsului, cantitatea, pretul unitar, codul de bara. Aceste date care nume si tipuri diferite le numim câmpurile articolului. Articolul corespunzator unui produs îl vom numi PRODUS, iar câmpurile fiecaruia sunt DEN, CANT, PRET, COD. La un moment dat ne referim la denumirea unui produs prin PRODUS.DEN. Pentru un sir de produse referinta este PRODUS(i).DEN, adica denumirea produsului de pe pozitia i.

Definitie.

Numim fisier o multime de realizari (înregistrari) ale unuia sau mai multor articole.

EVIDENTA = {r1,r2,…rn} sau o colectie de înregistrari de acelasi tip.

Accesul la datele dintr-un fisier

Accesul la datele dintr-un fisier poate fi:

- *secvential*; pentru a ajunge la o anumita data dintr-un fisier este necesar sa parcurgem realizari de articol una dupa alta, pâna la acea data pe care o caut.

Toate suporturile de date pe care se gasesc fisiere admit acest mod de acces.

- *direct*, înseamna posibilitatea de acces direct la o realizare a articolului din fisier, fara a mai fi nevoie de consultarea celorlalte articole. Exista posibilitatea de identificare a înregistrarilor prin intermediul unor chei. Acest mod de acces este posibil numai pe anumite suporturi pe care exista mecanisme fizice care dau posibilitatea unui astfel de acces. Exista posibilitatea de acces direct în memoria calculatoarelor si pe discuri magnetice.

Organizare

Modurile de organizare a fisierelor sunt :

a) Organizarea secventiala presupune aranjarea înregistrarilor una dupa alta. Este posibila pe toate suporturile de date, accesul este numai cel secvential.

b) Organizarea secvential-indexata presupune asocierea la fisierul de aceasta organizare a unui tabel de index care contine 2 coloane (câmpuri); un câmp contine *cheile de articol* care identifica înregistrarile fisierului; al doilea câmp contine adresele fizice ale înregistrarilor fisierului de aceasta organizare. Consultarea unui astfel de fisier se face folosindu-ne de tabela de index, întâi se consulta tabela de index, apoi din tabela gasim înregistrarea care ne intereseaza, apoi având adresa acestei înregistrari ne ducem direct la înregistrarea pe care o cautam. Avem posibilitatea de acces direct la aceste înregistrari. Este posibila o astfel de organizare pe suporturi care admit un acces direct.

c) Organizarea selectiva, foloseste niste câmpuri *cheie* ale fisierului pe care dorim sa-l organizam în acest fel. Facem o distribuire a realizarilor de articol în asa numitele *casete* care au câte un numar de ordine, folosind un algoritm special în acest scop; într-o caseta se gasesc un anumit numar de înregistrari. Consultarea unui fisier selectiv presupune identificarea casetei în care se gaseste acea înregistrare, parcurgerea secventiala a înregistrarilor din caseta respectiva admit accesul direct. Distribuirea în casete a înregistrarilor se face pe baza unor algoritmi numiti algoritmi de randomizare.

Un astfel de algoritm poate fi: aleg un câmp al fisierului numeric si numarul casetei în care se include o realizare de articol poate fi restul împartirii la *n* a valorii numerice din câmpul cheie ales. Vor fi *n* casete numerotate: 0,1,…,n

Operatii cu fisiere

Operatiile cu fisiere presupun:

- *creare* de fisier (a scrie într-un fisier, înregistrari);
- *exploatate* sau *consultate* (a citi) realizari de articol.
- *actualizarea* poate fi :

? *modificare*, adica citesc o înregistrare, din ea modific o anumita data.
? *adaugare*, adaug o noua înregistrare
? *stergere*, stergerea unei înregistrari.

Actualizarea presupune o citire a înregistrarii, apoi modificarea, scrierea unei înregistrari sau stergerea unei înregistrari.

La nivelul programului sursa Pascal, un fisier este referit printr-o variabila fisier, ceea ce am numit anterior identificator logic de fisier. Deoarece Pascal este un limbaj puternic tipizat, identificatorul logic de fisier trebuie sa fie de una dintre urmatoarele tipuri:

- **text** (pentru fisiere text)
- **file of tip_componenta** (fisierul este un fisier record în care fiecare înregistrare este de tipul tip_componenta)
- **file** (fisierul este nedefinit, fiind considerat ca o succesiune de octeti - stream)

Declararea variabilelor fisier se face uzual, folosind cuvântul rezervat var. Variabilele fisier nu se pot folosi decât în operatiile specifice lucrului cu fisiere. Mai exact, o variabila fisier se initializeaza prin operatia de deschidere a fisierului si îsi pierde valoarea la închiderea acestuia.

```
var
 fis_text: Text;                  { fisier text }
 fis_intregi = File of Integer;   { fisier de intregi }
 fis_nedefinit: File;             { fisier nedefinit }
```

Deschiderea unui fisier

Operatia de deschidere a unui fisier este prima operatie care se efectueaza asupra acestuia, înaintea prelucrarilor propriu-zise la care acesta este supus. Menirea acestei operatii este de a stabili

- modul în care este folosit fisierul (citire sau scriere)
- alocarea buffer-ului si initializarea contorului de pozitie.

Deschiderea fisierelor

In Turbo si Borland Pascal, operatia de deschidere a unui fisier se realizeaza în doi pasi:

- initializarea variabilei fisier;
- deschiderea propriu-zisa a fisierului.

Initializarea variabilei fisier se face cu procedura standard Assign, care are declaratia:

```
procedure Assign(var var_fis; nume_fis:String);
```

unde:

- var_fis este o variabila fisier de oricare tip (identificatorul logic de fisier),
- nume_fis este un sir de caractere ce desemneaza numele extern al fisierului (specificatorul de fisier)

Dupa executia procedurii, var_fis va fi initializata, fiind asociata fisierului extern nume_fis; cu alte cuvinte, orice operatie pe var_fis va însemna de fapt lucrul cu fisierul nume_fis. Asocierea ramâne valabila pâna când se închide fisierul referit de var_fis sau pâna când var_fis apare într-o alta procedura Assign.

Daca nume_fis este sirul de caractere vid, var_fis va fi asociata unuia dintre fisierele sistem. Daca nume_fis este numele unui fisier deja deschis, se produce o eroare de executie.

Deschiderea propriu-zisa a unui fisier se face în mod obisnuit prin apelul uneia dintre procedurile standard Reset sau Rewrite.

Procedura standard Reset realizeaza deschiderea unui fisier (de obicei) în citire. Declaratia sa este:

procedure Reset(var var_fis);

unde var_fis este o variabila fisier de oricare tip (identificatorul logic de fisier), asociat în prealabil unui fisier extern prin apelul procedurii Assign.

Reset deschide fisierul extern asociat variabilei var_fis. daca fisierul respectiv este deja deschis, el se închide în prealabil si apoi are loc deschiderea. Contorul de pozitie al fisierului se seteaza pe începutul fisierului. Reset produce o eroare de intrare-iesire daca fisierul extern nu exista. în cazul fisierelor text, Reset produce deschiderea acestora numai în citire.

Pentru fisierele non-text, Unit-ul System contine variabila FileMode, care contine informatia privitoare la modul de deschidere a acestora prin Reset:

- 0 - Read only (numai citire)
- 1 - Write only (numai scriere)
- 2 - Read/Write (valoare implicita).

Aceasta variabila se poate seta de catre programator prin operatia de atribuire.

Procedura standard Rewrite realizeaza deschiderea unui fisier în scriere.

Declaratia sa este:

procedure Rewrite(var var_fis);

unde var_fis este o variabila fisier de oricare tip (identificatorul logic de fisier), asociat în prealabil unui fisier extern prin apelul procedurii Assign.

Rewrite creeaza fisierul extern asociat variabilei var_fis. Daca fisierul respectiv exista, atunci el se sterge si se creeaza un fisier vid. Daca el este deja deschis, atunci se închide în prealabil si apoi este re-creat.

Contorul de pozitie al fisierului se seteaza pe începutul fisierului. în cazul fisierelor text, Rewrite produce deschiderea acestora numai în scriere.

Închiderea unui fisier

Operatia de închidere a unui fisier semnifica terminarea lucrului cu acesta. Prin închidere se realizeaza urmatoarele:

- transferarea informatiei din buffer pe suport (în cazul fisierelor deschise în scriere)
- distrugerea asocierii între variabila fisier si numele extern al fisierului.

Dupa închidere, fisierul se poate folosi din nou, respectând etapele descrise anterior. Variabila fisier devine disponibila, ea putând fi folosita într-un alt apel al procedurii Assign.

Pentru un fisier deschis în citire, prelucrarea sa se termina de regula atunci când s-a ajuns la sfârsitul sau. Detectarea sfârsitului de fisier se face cu functia standard EOF. Declaratia acesteia este:

function Eof(var var_fis): Boolean; *{ fisiere non-text }*

function Eof [(var var_fis: Text)]: Boolean; { fisiere text }

In cazul fisierelor text, daca var_fis lipseste se considera ca EOF se refera la fisierul standard de intrare. Altfel, var_fis va referi un fisier deschis în prealabil prin Assign si Reset/Rewrite. EOF întoarce:

- True daca contorul de pozitie este dupa ultimul octet din fisier sau daca fisierul este vid
- False în toate celelalte cazuri.

Exemplu: citirea unui fisier text si afisarea acestuia pe ecran;

```
Program CitireT;
 var f: text;
    linie: string;
    numeFis: String;
    este: Boolean;

 function Exista(NumeFisier: String): Boolean;
 { intoarce
    True  daca fisierul extern reprezentat de NumeFisier exista
    False altfel
 }
  var
    f: File;
  begin
    {$I-}                          { comuta pe off indicatorul $I }
    Assign(f, NumeFisier);
    FileMode := 0;                 { deschide în citire }
    Reset(f);
    Close(f);
    {$I+}
    Exista := (IOResult = 0) and (NumeFisier <> '')
  end; { Exista }

begin
  WriteLn('CitireT - afisarea unui fisier text pe ecran');
  Repeat
    Write('Dati numele fisierului: ');
    ReadLn(numeFis);
    este := Exista(numeFis);
    if not este then WriteLn('Fisier inexistent!');
  Until este;
```

```
Assign(f, numeFis);          { asociaza f la numeFis }
Reset(f);                    { deschide f în citire }
While not Eof(f) do begin  { cat timp nu s-a ajuns la sfarsit }
  Readln(f, linie);          { citeste o linie din fisier }
  Writeln(linie)             { scrie linia la iesirea standard }
end;
Close(f);                    { inchide fisierul }
end. { CitireT }
```

Pozitionarea

Operatia de pozitionare se refera la fisierele non-text si are ca efect modificarea contorului de pozitie. Modificarea se face în doua moduri:

- implicit, prin operatiile de citire si de scriere; orice operatie de transfer modifica contorul de pozitie, acesta avansând spre capatul fisierului
- explicit, prin procedura standard Seek.

Mediile Borland pun la dispozitia programatorului o functie si o procedura care îi permit acestuia sa acceseze si sa modifice contorul de pozitie:

FilePos si *Seek*. De asemenea, functia *FileSize* determina dimensiunea unui fisier.

Functia *FileSize* întoarce numarul de componente dintr-un fisier. Declaratia sa este urmatoarea:

```
function FileSize(var var_fis): Longint;
```

în care var_fis este o variabila fisier, asociata unui fisier deschis în prealabil. *FileSize* întoarce numarul de componente al fisierului. daca fisierul este vid, *FileSize* întoarce 0. Semantica exacta a lui *FileSize* pentru fiecare clasa de fisiere poate fi studiata folosind exemplele mediului.

Functia FilePos întoarce valoarea contorului de pozitie al unui fisier non-text. Declaratia sa este:

```
function FilePos(var var_fis): Longint;
```

în care var_fis este o variabila fisier, asociata unui fisier deschis în prealabil. daca contorul de pozitie este pe începutul fisierului, *FilePos* va întoarce 0, iar daca contorul de pozitie este la sfârsitul fisierului, atunci *FilePos(var_fis)* va fi egal cu *FileSize(var_fis)*.

Contorul de pozitie al unui fisier non-text se exprima în unitari de masura proprii tipului de fisier. El semnifica înregistrarea curenta din fisier, luând valori de la 0 (începutul fisierului, prima înregistrare din el) la *FileSize(var_fis)* - 1 (ultima înregistrare din fisier). Procedura standard *Seek* realizeaza modificarea contorului de pozitie la o noua valoare, specificata ca parametru al acesteia. Declaratia sa este:

procedure Seek(var var_fis; pozitie: Longint);

în care:

- var_fis este o variabila fisier, asociata unui fisier deschis în prealabil
- pozitie este un întreg, cu urmatoarele valori valide:
  - ? între 0 si *FileSize(var_fis) - 1*: contorul de pozitie al fisierului var_fis se va seta la valoarea pozitie
  - ? FileSize(var_fis): în fisierul var_fis se va adauga (la sfârsit) o noua înregistrare

Exista, de asemenea, procedura Truncate, care permite trunchierea unui fisier non-text. Declaratia sa este:

procedure Truncate(var var_fis);

în care var_fis este o variabila fisier, asociata unui fisier deschis în prealabil. Truncate pastreaza în fisier numai înregistrarile de la 0 si pâna la FilePos(var_fis) - 1, eliminând celelalte înregistrari din el (daca exista) si setând EOF(var_fis) pe True.

Citirea

Operatia de citire înseamna transferarea de informatie din fisierul extern în memoria interna a calculatorului. Mai exact, citirea realizeaza initializarea unor variabile din programul Pascal cu valori preluate din fisierul extern.

Citirea se face diferit pentru fiecare clasa de fisiere Borland (Turbo) Pascal. Ea poate sau nu sa fie însotita de conversii.

Scrierea

Operatia de scriere înseamna transferarea de informatie din memoria interna a calculatorului în fisierul extern. Mai exact, scrierea realizeaza memorarea valorilor unor variabile din programul Pascal în fisierul extern. Scrierea se face diferit pentru fiecare clasa de fisiere Borland (Turbo) Pascal. Ea poate sau nu sa fie însotita de conversii.

2.4.1. Fisiere text

Fisierele text sunt fisiere speciale, care se pot citi sau edita de orice editor de texte standard. Un fisier text este o succesiune de caractere ASCII organizate în linii. Numarul de linii este variabil, iar fiecare linie contine un numar variabil de caractere. O linie se termina cu o combinatie speciala de caractere (de regula CR+LF, adica ASCII 10 + ASCII 13), iar sfârsitul de fisier poate fi determinat cu ajutorul functiei EOF. Aceasta poate folosi:

- functia *FileSize* (care determina numarul de caractere din fisier)

- un caracter special de sfârsit de fisier (cu codul ASCII 26, recunoscut prin combinatia CTRL+Z de la tastatura).

Mediile Borland si Turbo Pascal permit specificarea unui fisier text prin cuvântul cheie text, care are declaratia:

type text = file of char;

In Borland si Turbo Pascal sunt disponibile urmatoarele functii si proceduri specifice lucrului cu fisierele text:

- *Append (procedura)*
- *EOLN (functie)*
- *Flush (procedura)*
- *Read (procedura)*
- *ReadLn (procedura)*
- *SeekEOF (functie)*
- *SeekEOLN (functie)*
- *SetTExtBuf (procedura)*
- *Write (procedura)*
- *WriteLn (procedura)*

In cele ce urmeaza, cu exceptia locurilor unde se face o referire explicita, prin var_fis vom desemna o variabila fisier de tip text, deschis în prealabil.

Procedura Append este specifica fisierelor text. Ea permite deschiderea unui fisier în adaugare, adica:

- deschide fisierul în scriere
- seteaza contorul de pozitie la sfârsitul fisierului.

Declaratia sa este:

procedure Append(var var_fis: Text);

unde var_fis este o variabila fisier de tip text, asociata în prealabil printr-o procedura Assign unui fisier extern existent. daca fisierul extern nu exista, se produce o eroare de intrare-iesire. daca var_fis desemneaza un fisier deja deschis, acesta se închide în prealabil si apoi se executa operatiile specifice lui Append.

Daca terminatorul de sfârsit de fisier CTRL+Z este prezent, contorul de pozitie se seteaza pe pozitia acestuia. Prin urmare, dupa fiecare scriere în fisier acesta va contine la sfârsitul sau terminatorul de fisier.

Functia EOLN întoarce statutul de sfârsit de linie, adica EOLN

- întoarce True daca :
- caracterul curent din fisier (specificat prin contorul de pozitie) este un terminator de linie sau de fisier
- între caracterul curent si sfârsitul de linie nu exista decât spatii
- întoarce False altfel.

Declaratia aceste functii este:

function EOLN [(var var_fis: Text)]: Boolean;

Daca var_fis lipseste, se considera ca EOLN se refera la fisierul standard de intrare.

Procedura Flush realizeaza transferul fizic de informatie din bufferul unui fisier deschis în scriere (cu Rewrite sau Append) pe suport. Declaratia sa este:

procedure Flush(var var_fis: Text);

Scrierea pe un fisier text se realizeaza prin intermediul unui buffer. în mod normal, scrierea fizica se efectueaza numai atunci când bufferul este plin.

Apelând Flush, ne asiguram ca se efectueaza scrierea si când bufferul nu este plin.

Procedura standard Read citeste valori dintr-un fisier text într-una sau mai multe variabile. Citirea se efectueaza începând de la contorul de pozitie, avansându-se spre sfârsitul fisierului. Se pot efectua conversii, în functie de tipul variabilelor prezente ca parametri ai lui Read. Declaratia procedurii este:

procedure Read([var var_fis: Text;] V1 [, V2,...,Vn]);

unde V1, V2, ..., Vn sunt variabile pentru care Read este în domeniul lor de vizibilitate. Citirea se opreste la sfârsitul de linie sau de fisier, fara a se citi si aceste caractere speciale.

Procedura standard ReadLn este similara procedurii Read, cu exceptia faptului ca dupa terminarea citirii trece la linia urmatoare: la întâlnirea caracterelor de sfârsit de linie le sare, setând contorul de pozitie dupa ele.

Declaratia procedurii este:

procedure ReadLn([var var_fis: Text;] V1 [, V2,...,Vn]);

Dupa executarea lui ReadLn, contorul de pozitie va fi setat pe începutul unei noi linii.

Functia SeekEOF întoarce True daca s-a ajuns la sfârsitul de fisier si False altfel. Declaratia sa este:

function SeekEOF [(var var_fis: Text)]: Boolean;

Functia SeekEOLN întoarce True daca s-a ajuns la sfârsit de linie si False altfel. Declaratia sa este:

function SeekEOLN [(var var_fis: Text)]: Boolean;

Procedura SetTextBuf atribuie unui fisier text un buffer de intrare-iesire.

Uzual, bufferul de I/E pentru fisierele text este de 128 de octeti. Cu cât bufferul este mai mare, cu atât operatiile de citire si de scriere se executa mai rapid si printr-un numar mai mic de accese la suportul fizic. Declaratia este urmatoarea:

procedure SetTextBuf(var var_fis: Text; var Buf [; Lung: Word]);

unde:

- Buf este numele unei variabile (de obicei de tipul unui tablou de caractere) care va fi folosita pe post de buffer

- Lung este dimensiunea bufferului (în octeti)

SetTextBuf trebuie apelata imediat dupa deschiderea fisierului (urmatoarea instructiune dupa Reset, Rewrite sau Append).

Procedura Write scrie valoarea uneia sau mai multor variabile într-un fisier text. Scrierea se efectueaza începând de la contorul de pozitie, avansându-se spre sfârsitul fisierului. Se pot efectua conversii, în functie de tipul variabilelor prezente ca parametri ai lui Write. Declaratia procedurii este:

procedure Write([var var_fis: Text;] P1 [, P2,...,Pn]);

unde P1, P2, ..., Pn sunt expresii de formatate, formate din nume de variabile sau expresii (de una din tipurile Char, Integer, Real, String si Boolean) ce contin variabile împreuna cu specificatori de lungime si de numar de zecimale. Pentru tipurile numerice, se face conversia la string înainte de scrierea în fisier. Fisierul trebuie sa fie deschis cu Rewrite sau Append. Contorul de pozitie se va mari cu lungimea stringurilor scrise.

O expresie de formatare P are formatul: V:l[:z], unde

- v este o variabila de tip numeric

- l este lungimea stringului generat (daca partea întreaga a lui v are lungimea mai mare ca l, atunci l se va seta la lungimea partii întregi a lui v - luând în considerare si semnul)

- z este numarul de cifre la partea zecimala (numai pentru variabile reale).

Procedura WriteLn este similara cu Write, scriind la sfârsit un terminator de linie. Declaratia procedurii este:

procedure WriteLn([var var_fis: Text;] P1 [, P2,...,Pn]);

iar semantica ei este:

Write(var var_fis, P1 [, P2,...,Pn]);
Write(var var_fis, Chr(13), Chr(10)); { terminator de linie }

Procedurile ReadLn si WriteLn sunt specifice fisierelor text, pe când procedurile Read si Write se folosesc si în cazul fisierelor cu tip.

Exemplu:

```
Program CitireT;
  { exemplu de citire a unui fisier text cu afisarea lui pe ecran }
  uses
    UFile{, Crt};
  var
```

```
  f: text;
  linie: string;
  numeFis: string;
  nrLinie: integer;                          { contor de linie }
  nrPagina: integer;
  este: boolean;
 begin
  Repeat
   {ClrScr;}
   WriteLn('CitireT - afisarea unui fisier text pe ecran');
   Repeat
    Write('Dati numele fisierului (ENTER la terminare): ');
    ReadLn(numeFis);
    If numeFis = '' then Exit;
    este := Exista(numeFis);
    if not este then WriteLn('Fisier inexistent!');
   Until este;
   nrLinie := 0;
   nrPagina := 1;
    WriteLn('Continutul fisierului ', numeFis, ' este:                ',
'Pag. ', nrPagina:3);
   Assign(f, numeFis);                  { asociaza f la numeFis }
   Reset(f);                            { deschide f în citire }
   While not EOF(f) do begin            { cat timp nu s-a ajuns la sfarsit
}
    ReadLn(f, linie);                   { citeste o linie din fisier }
    Inc(nrLinie);                       { incrementeaza contorul }
    WriteLn(nrLinie:5, ' ', linie);
    If nrLinie mod 20 = 0 then begin
     Asteapta;
     {ClrScr;}
     nrPagina := nrPagina + 1;
        WriteLn('Continutul fisierului ', numeFis, ' este:                ',
'Pag. ', nrPagina:3);
    end
   end;
   if nrLinie mod 20 <> 0 then Asteapta;
   Close(f);
   {Fictiva;}
  Until False
 end.                                   { CitireT }
```

2.4.2. Fisiere cu tip

Fisierele cu tip (pe care le numim si fisiere record) sunt accesate prin intermediul unei variabile fisier declarata astfel:

```
var
  var_fis: file of tip_componenta;
```

sau, mai elegant:

```
type
  tip_componenta = ... { definitia tipului componentei }
  tip_fisier = file of tip_componenta;
var
  var_fis: tip_fisier;
```

Fisierele cu tip respecta definitia generala a fisierului, precizata la începutul acestei sectiuni: ele contin componente de acelasi tip, notat mai sus prin tip_componenta. Tipul componentei poate fi orice tip recunoscut de sistemul de tipuri al limbajului, cu exceptia tipului file.

Spre exemplu, urmatoarele tipuri de fisiere sunt tipuri valide:

```
type
  fisier_integer = file of Integer;
  fisier_boolean = file of Boolean;
  fisier_persoane = file of Persoana; { tipul Persoana descris mai sus }
```

Bufferul este o zona de memorie speciala, fara nume, care o vom nota în cele ce urmeaza cu var_fis^(notiunea de pointer este tratata în capitoul urmator). Initializarea variabilei fisier var_fis se face prin apelul procedurii standard Assign; deschiderea fisierului se face folosind procedurile standard Reset (în citire sau în scriere si citire) si Rewrite (în scriere). Odata cu deschiderea, devine accesibil si bufferul fisierului, adica variabila var_fis^.

Bufferul trebuie considerat ca zona de memorie în care se citesc informatiile din fisierul extern, înainte ca acestea sa fie atribuite variabilelor din program; similar, în buffer sunt depozitate informatiile care se doreste a fi scrise în fisierul extern, înainte ca scrierea sa aiba loc. Ratiunea de a fi a bufferului este aceea de a optimiza (în general de a micsora) numarul de accese la suportul extern, pe care se gaseste fisierul supus prelucrarii.

De obicei, operatiile fizice de citire si scriere pe suport extern realizeaza transferul unei cantitati fixe de informatie, numita bloc. Dimensiunea blocului depinde de caracteristicile fizice ale suportului si perifericului pe care se memoreaza fisierul. Din punct de vedere logic, adica al fisierului prelucrat în programul Pascal,

o citire sau scriere logica realizeaza transferarea unei cantitati de informatie egala cu dimensiunea unei componente a fisierului, adica *SizeOf(tip_componenta)*. De regula, dimensiunea bufferului este un multiplu al dimensiunii bloc

Pentru a simplifica lucrurile, consideram ca dimensiunile blocului, bufferului si componentei sunt egale. Folosind declaratia:

```
type
  tip_componenta = Integer;
  tip_fisier = file of tip_componenta;
var
  var_fis: tip_fisier;
  componenta: tip_componenta;
```

vom deschide acum în citire fisierul TEST.DAT si vom putea accesa deja prima componenta a lui:

```
Begin
  Assign(var_fis, 'TEST.DAT');
  Reset(var_fis);
  componenta := var_fis^
End.
```

Deschiderea provoaca si setarea contorului de pozitie pe prima înregistrare. Pentru a trece la urmatoarea componenta (adica pentru a aduce în buffer urmatoarea componenta) se foloseste o procedura speciala, numita get.

Semantica acesteia este:

- aducerea în buffer a urmatoarei componente din fisier
- marirea cu 1 a contorului de pozitie.

Prin urmare, daca dorim o prelucrare completa a fisierului (de exemplu afisarea fiecarei componente pe ecran), atunci programul de mai sus s-ar scrie astfel:

```
Begin
  Assign(var_fis, 'TEST.DAT');
  Reset(var_fis);
  While not Eof(var_fis) do begin
    componenta := var_fis^;
    get(var_fis);
    WriteLn(componenta)
  end;
  Close(var_fis)
end.
```

Am vazut însa ca în Pascal exista procedura standard Read pentru citirea din fisier. De fapt, semantica exacta a procedurii Read este data în tabelul urmator:

Procedura standard	Este echivalenta cu

Read(var_fis, componenta) componenta := var_fis^;
                           get(var_fis);

Procedura get detecteaza sfârsitul de fisier: când nu mai exista o urmatoare înregistrare de citit, ea nu va întoarce nimic, iar functia Eof(var_fis) va întoarce True, deci citirea fisierului se termina.

Folosind procedura standard Read, programul de mai sus se scrie astfel:

```
Begin
  Assign(var_fis, 'TEST.DAT');
  Reset(var_fis);
  While not Eof(var_fis) do begin
    Read(var_fis, componenta);
    WriteLn(componenta)
  end;
  Close(var_fis)
end.
```

Scrierea în fisier

Folosind aceleasi declaratii:

```
type
  tip_componenta = Integer;
  tip_fisier = file of tip_componenta;
var
  var_fis: tip_fisier;
  componenta: tip_componenta;
  i: Integer;
```

vom deschide acum în scriere fisierul TEST.DAT si vom pune prima componenta a lui în buffer:

```
Begin
  Assign(var_fis, 'TEST.DAT');
  Rewrite(var_fis);
  var_fis^ := componenta
End.
```

Deschiderea în scriere provoaca stergerea fisierului (daca acesta exista) si setarea contorului de pozitie pe 0. Pentru a scrie o componenta în fisier este nevoie ca aceasta sa fie trecuta prima data în buffer (lucru realizat de ultima instructiune

din program), dupa care se foloseste o procedura speciala, numita put. Semantica acesteia este:

Procedure put(var_fis)

- scrierea continutului bufferului în fisier
- marirea cu 1 a contorului de pozitie.

Prin urmare, daca dorim o prelucrare completa a fisierului (de exemplu scrierea fiecarei componente în el), atunci programul de mai sus s-ar scrie astfel:

```
Begin
  Assign(var_fis, 'TEST.DAT');
  Rewrite(var_fis);
  For i := 1 to 10 do begin
    componenta := i;
    var_fis^ := componenta;
    put(var_fis)
  end;
  Close(var_fis)
end.
```

Am vazut însa ca în Pascal exista procedura standard Write pentru scrierea în fisier. De fapt, semantica exacta a procedurii Write este data în tabelul urmator:

Procedura standard	Este echivalenta cu
Write(var_fis, componenta)	var_fis^ := componenta; put(var_fis);

In scriere, sfârsitul de fisier trebuie marcat. Acest lucru este efectuat de procedura standard Close.

Folosind procedura standard Write, programul de mai sus se scrie astfel:

```
Begin
  Assign(var_fis, 'TEST.DAT');
  Rewrite(var_fis);
  For i := 1 to 10 do begin
    componenta := i;
    Write(var_fis, componenta)
  end;
  Close(var_fis)
end.
```

2.4.3. Fisiere fara tip (stream)

Fisierele fara tip pot fi considerate ca fiind fisiere cu tip în care o înregistrare are un octet, si tipul ei este Byte

```
type
  file = file of byte;
```

Deoarece este greoaie manipularea înregistrarilor de 1 byte, la fisierele fara tip se foloseste un parametru nou, dimensiunea înregistrarii (bufferului), numit în engleza RecSize. Acest parametru se precizeaza la deschiderea fisierului. daca nu este precizat, se considera implicit valoarea 128. Procedurile standard Reset si Rewrite au forma generala:

procedure Reset(var var_fis s: File; Recsize: Word]);
procedure Rewrite(var F: File s; Recsize: Word]);

Pentru citirea si scrierea din fisierele fara tip se folosesc proceduri specifice, BlockRead si BlockWrite.

Procedura BlockRead are sintaxa:

procedure BlockRead(var F: File; var Buf; Count: Word [; var Result: Word]);

unde:

F Variabila fisier fara tip

Buf variabila de orice tip (de obicei un tablou de byte), de lungime cel putin egala cu RecSize

Count expresie de tip Word

Result variabila de tip Word

Semantica acestei proceduri este urmatoarea: se citesc cel mult Count înregistrari (de lungime RecSize fiecare) din fisierul F în variabila Buf (care joaca rolul bufferului). Numarul efectiv de înregistrari citite este întors în parametrul optional de iesire Result. daca acesta lipseste, se declanseaza o eroare de intrare-iesire (detectabila prin folosirea lui IOResult) daca numarul de înregistrari citite este mai mic decât Count.

Pentru ca operatia sa aiba sens, dimensiunea bufferului Buf trebuie sa fie cel putin egala cu Count * RecSize octeti, dar nu mai mare decât 64K:

65535 > SizeOf(Buf) > Count * RecSize

Daca Count * RecSize > 65535 se declanseaza o eroare de intrare-iesire.

Parametrul de iesire Result va avea valoarea (daca este prezent în apel) - egala cu Count daca din fisier s-a putut transfera numarul de octeti precizat

- mai mic decât Count daca în timpul transferului s-a detectat sfârsitul de fisier; Result va contine numarul de înregistrari complete citit.

Dupa terminarea transferului, contorul de pozitie al acestuia se mareste cu Result înregistrari.

Procedura BlockWrite are sintaxa:

procedure BlockWrite(var F: File; var Buf; Count: Word [; var Result: Word]);

unde:

F	Variabila fisier fara tip
Buf	variabila de orice tip (de obicei un tablou de byte), de lungime cel putin egala cu RecSize
Count	expresie de tip Word
Result	variabila de tip Word

Semantica acestei proceduri este urmatoarea: se scriu cel mult Count înregistrari (de lungime RecSize fiecare) din variabila Buf în fisierul F.

Numarul efectiv de înregistrari scrise este întors în parametrul optional de iesire Result. Daca acesta lipseste, se declanseaza o eroare de intrare-iesire (detectabila prin folosirea lui IOResult) daca numarul de înregistrari scrise este mai mic decât Count.

Pentru ca operatia sa aiba sens, dimensiunea bufferului Buf trebuie sa fie cel putin egala cu Count * RecSize octeti, dar nu mai mare decât 64K:

65535 > SizeOf(Buf) > Count * RecSize

Daca Count * RecSize > 65535 se declanseaza o eroare de intrare-iesire.

Parametrul de iesire Result va avea valoarea (daca este prezent în apel)

- egala cu Count daca s-a putut transfera în fisier numarul de octeti precizat

- mai mic decât Count daca suportul pe care se face transferul se umple; Result va contine numarul de înregistrari complete scris

Dupa terminarea transferului, contorul de pozitie al acestuia se mareste cu Result înregistrari.

3. Structuri statice si structuri dinamice. Liste

3.1. Liste. Structuri statice. Structuri dinamice.

Lista este o structura liniara si dinamica de date.

Spunem ca o structura este ***liniara*** daca ea este formata din elemente care au o pozitie bine determinata în cadrul structurii:

- exista un unic element numit primul;
- exista un unic element numit ultimul;
- orice element (cu exceptia primului) are un predecesor
- orice element (cu exceptia ultimului) are un succesor.

Spunem ca o structura este ***dinamica*** daca numarul de elemente al ei se modifica în timp. Cu alte cuvinte, spatiul de memorie alocat pentru respectiva structura are dimensiunea variabila în timp: în fiecare moment este alocata doar atâta memorie câta este necesara pentru a pastra elementele curente ale structurii.

Spunem ca o structura este ***statica*** daca numarul de elemente al ei este fix în timp. Cu alte cuvinte, spatiul de memorie alocat pentru respectiva structura are dimensiunea fixa în timp.

Putem face o comparatie între structurile de date tablou si lista. Ambele structuri sunt structuri liniare. Tabloul este o structura statica, pe când lista este o structura dinamica.

Tabloul (asa cum este el definit în Pascal) are un numar de elemente cunoscut dinainte. Spunem ca tabloul este o structura statica. Examinând definitia structurii liniare, constatam ca tabloul verifica aceasta definitie:

- fiecare element este memorat în tablou într-o singura locatie, definita de valoarea unui 'indice'
- pentru un tablou cu n elemente, indicii variaza de regula de la 1 la n
- exista un unic element numit primul, acela cu indicele cel mai mic
- exista un unic element numit ultimul, acela cu indicele cel mai mare
- primul element al tabloului are indicele 1 si nu are predecesor
- ultimul element al tabloului are indicele n si nu are succesor
- orice alt element de indice i (1 < i < n) are
  - ✍ un predecesor: elementul de indice i-1
  - ✍ un succesor: elementul de indice i+1

Tabloul este o structura statica pentru ca înainte de alocarea lui în memorie trebuie cunoscut n - numarul de elemente al sau. Prin urmare, indiferent câte elemente din tablou vor fi folosite efectiv, se va aloca spatiu pentru toate.

Lista este o structura dinamica: initial ea este vida, deci nu contine nici un element, iar spatiul de memorie alocat are dimensiunea 0. Pe masura ce apar (se produc) elemente care trebuie incluse în lista, se aloca spatiu pentru ele si se face includerea, stabilindu-se prin aceasta pozitia pe care elementul inclus o ocupa în cadrul listei. Nu exista teoretic nici o restrictie privitoare la locul unde se include un nou element în lista; includerea se poate face fie la începutul ei (înaintea primului element din ea), fie undeva la mijlocul ei (intre primul si ultimul element), fie la sfârsitul listei, dupa ultimul element.

În mod analog, din lista se pot scoate elemente. Scoaterea unui element din lista înseamna refacerea legaturilor între elementul dinaintea si cel de dupa elementul care se scoate.

Elementele listei se numesc ***noduri***.

Lista este astfel organizata încât fiecare nod al ei contine:

- informatia utila din nod
- adresa nodului precedent (predecesorul nodului curent)
- adresa nodului urmator (succesorul nodului curent).

În terminologia listelor, operatiile de mai sus se numesc:

- inserare: adaugarea unui nod nou la lista
- stergere: stergerea unui nod din lista
- parcurgere (traversare): inspectarea nodurilor din lista, începând cu primul si terminând cu ultimul.

Adresele nodului precedent si urmator formeaza ceea ce numim „informatie de înlantuire", iar structura de date *lista* cu nodurile definite ca mai sus se numeste în practica lista ***dublu înlantuita***. Aceasta înseamna ca, daca ne aflam într-un nod, putem sa parcurgem lista fie spre începutul acesteia (folosind adresa nodului precedent) fie spre sfârsitul acesteia (urmând adresa nodului urmator).

3.2. Lista simplu înlantuita

Lista simplu înlantuita este un caz particular de lista, în nodurile careia se pastreaza fie adresa nodului precedent, fie adresa nodului urmator.

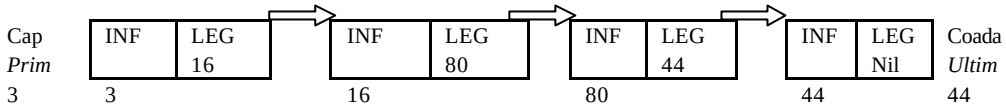
O **lista simplu înlantuita** (*lista liniara simplu înlantuita*) este un sir de elemente dintr-o anumita multime. Fiecare element al unei liste ocupa o zona de memorie (locatie) formata din doua componente: INFORMATIA si LEGATURA.

Memorarea listelor se face prin alocarea unor zone de memorie pentru elementele acesteia în doua moduri:

a) **Alocarea secventiala** consta în memorarea elementelor lstei în locatii aflate la adrese succesive în memorie, pornind de la o adresa de baza, adresa primului element din lista. Fiecare element din lista poate fi identificat conform ordinii în lista.

Un exemplu, pe care l-am studiat, de astfel de liste, sunt tablourile. Fiecare element se identifica prin indicele sau în tablou, iar informatia se afla în locatia T(i1, i2,…,in).

b) **Alocarea înlantuita** presupune ca fiecarui element din lista sa i se aloce o zona corespunzatoare în care se pastreaza INFormatia si adresa de LEGatura la care se afla urmatorul element, sub forma:



Sub fiecare locatie am precizat adresa la care se afla si valorile corespunzatoare pentru primul si ultimul element. Se observa ca locatiile de memorie nu au adrese succesive. Avantajul alocarii înlantuite este ca se foloseste doar atâta zona de memorie cât este necesara si se poate folosi orice zona libera la un moment dat.

Operatii elementare asupra listelor:

Inserarea unui element, într-un anumit loc, în lista (la cap, la coada, în interiorul listei);

Stergerea unui element din lista (eliminarea din lista);

Parcurgerea elementelor unei liste, deci accesul la elementele sale, fie prin pozitia în lista fie prin valoarea informatiei.

Concatenarea listelor.

Tipuri de liste simplu înlantuite

În functie de operatiile pe care le putem aplica listelor liniare, precum si a locului în care acestea se efectueaza, distingem urmatoarele tipuri de liste:

a) ***lista liniara simplu înlantuita*** – inserarea si stergerea elementelor se face oriunde în lista.

b) ***Stiva*** (lista **LIFO, Last în First Out,** ultimul intrat va fi primul iesit, inserarea si stergerea se realizeaza la acelasi capat, *Cap* sau *Prim***)**

c) ***Coada*** **(**lista **FIFO, First în First Out,** primul intrat va fi primul iesit, inserarea si stergerea se realizeaza la capete diferite, *Coada* sau *Ultim* respectiv *Cap* sau *Prim***)**

d) ***Lista circulara*** (lista nu are capete, legatura ultimului element se face spre primul element)

3.2.1. Specificarea structurii de date lista simplu înlantuita

Structura de date lista simplu înlantuita pe care o prezentam în continuare are operatii proprii în functie de tipul structurii:

- structurilor liniare: pozitionare (primul, ultimul, precedentul, urmatorul), element curent, cautare, traversare

- structurilor dinamice: creare, inserare, stergere, eliberare

Specificarea unei structuri de date cuprinde urmatoarele sectiuni:

- *Elemente*: precizeaza informatii despre elementele continute în structura
- S*tructura*: precizeaza informatii despre legaturile dintre elemente
- *Domeniu*: precizeaza câmpurile structurii
- *Operatii*: precizeaza operatiile efectuate asupra structurii

Specificarea unei operatii se face prin:

Nume: numele operatiei

Parametri: parametrii operatiei, numele lor

Preconditie: predicat referitor la datele de intrare conditiile în care se poate executa operatia

Postconditie: predicat referitor la datele de iesire conditiile pe care le satisfac acestea, legatura dintre datele de intrare si datele de iesire

Specificarea structurii de date lista simplu înlantuita

Elemente: Toate elementele sunt de acelasi tip, desemnat prin TNod si se numesc noduri. Un nod contine informatie utila, de tipul TInfo

Structura: Lista simplu înlantuita are o structura liniara. Exista un nod numit primul si un nod numit ultimul. Fiecare nod (cu exceptia ultimului) contine în el adresa nodului urmator.

Domeniu: Lista are doua câmpuri, Cap si Cursor. Cap refera primul element al sau, iar Cursor refera elementul curent.

Operatii:

Creeaza(L) - creeaza o lista L vida

Pre: True

Post: L este vida (L.Cap si L.Cursor nu refera nimic)

L.Cap = L.Cursor = Nil

Vida(L) - testeaza daca L este vida

Pre: True

Post: vida = True daca L este vida si vida = False în caz contrar

Insereaza(L, I) - insereaza în L un nod nou (care contine informatia utila I), pe pozitia cursorului

Pre: True

Post: L.Cap[Info] = I si L.Cursor = L.Cap (daca L.Cursor = Nil) sau daca L.Cursor <> Nil, atunci L.Cursor[Urm] = nodul nou inserat si apoi L.Cursor := nodul nou inserat

Sterge(L) - sterge din L nodul referit de L.Cursor

Pre: L nu este vida si L.Cursor refera un nod existent în L
Post: nodul referit de L.Cursor este sters si nodul din capul listei devine nod curent (este referit de L.Cursor)

Modifica(L, I) - modifica informatia utila din nodul referit de cursor
Pre: L este nevida
Post: informatia utila din nodul referit de cursor are valoarea I

Extrage(L, I) - extrage informatia utila din nodul referit de cursor
Pre: L este nevida
Post: I primeste valoarea informatiei utile din nodul curent

Elibereaza(L) - sterge din L toate nodurile
Pre: True
Post: L este vida

Cauta(L, I) - cauta în L nodul cu informatia utila I
Pre: True
Post: Informatia utila din nodul curent este I si Cauta = True
(daca exista în L un nod cu informatia utila I)
sau
L.Cursor ramâne nemodificat si Cauta = False
(daca nu exista în L un nod cu informatia utila I)

Primul(L) - pozitioneaza L.Cursor pe primul element din lista
Pre: L este nevida
Post: L.Cursor := L.Cap

Ultimul(L) - pozitioneaza L.Cursor pe ultimul element din lista
Pre: L este nevida
Post: L.Cursor refera ultimul element din lista

Precedentul(L, Esec)
Pre: L este nevida
Post: L.Cursor refera nodul precedent lui L.Cursor si Esec = False
(daca L.Cursor nu refera primul nod din lista)
sau
L.Cursor ramâne nemodificat si Esec = True
(daca nu exista un nod precedent celui curent)

Urmatorul(L, Esec)

Pre: L este nevida
Post: L.Cursor refera nodul urmator lui L.Cursor si Esec = False
(daca L.Cursor nu refera ultimul nod din lista)
sau
L.Cursor ramâne nemodificat si Esec = True
(daca nu exista un nod urmator celui curent)

Traverseaza(L)
Pre: True;
Post: Parcurge toate nodurile listei, începând de la primul (L.Cap) si pâna la ultimul. L.Cursor nu se modifica.

3.3. Tipuri de liste simplu înlantuite

3.3.1. Stiva

Un exemplu clasic de stiva este "stivuirea unor cutii care au un anumit continut". Este evident ca doar asupra cutiei din vârful stivei putem sa actionam.

Stivele permit trei operatii elementare: inserarea la vârf, stergerea de la vârf si parcurgerea.

În descrierile algoritmilor, pentru operatiile pe stiva, vom folosi în reprezentarea generala a unui element dintr-o lista tipul de date articol care are structura:

Element = **articol**
leg: adresa_element;
inf: informatia_de_un_anumit_tip;
unde
Adresa_element = adresa_unei_zone_ de_memorie
Vom folosi variabilele
p, q, vârf : adresa_element;
Prin ***leg [p]*** si ***inf [p]*** vom întelege legatura si informatia aflata la adresa p.
leg [p] este de tip adresa_element, iar ***inf [p]*** este de tip informatia_de_un_ anumit_tip.
Pentru stiva vida avem ***vârf = nil***.
Alocarea unei zone noi de memorie se realizeaza cu **`aloc`** (adresa_element)
Disponibilizarea unei zone de memorie se realizeaza cu **`disp`** (adresa_element).
Inserarea unui element în vârful stivei (deci **vârf** are o anumita valoare)

```
Aloc (p);                    {alocarea memorie pentru noul element}
inf [p] = <expresie>;        {atribuirea valorii zonei de informatie}
leg [p] = vârf;              {adresa_de_legatura a noului element este vârf}
vârf    = p;                 {noul vârf se afla acum la noua adresa p }
```

Stergerea unui element din vârful stivei (deci **vârf** are o anumita valoare)

```
P = vârf;          {se salveaza adresa vârfului pentru a nu se pierde}
Vârf = leg[p];     {noua adresa a vârfului este legatura celui vechi}
Disp ( p );        {se disponibilizeaza (sterge) zona de adresa p}
```

Parcurgerea elementelor unei stivei pornind din **vârf.**

```
P = vârf;          {se salveaza adresa vârfului pentru a nu se pierde}
Cât timp p <> nil executa{conditia poate fi schimbata dupa cerintele de parcurgere}
   {Operatie asupra elementului de la adresa p}
   p = leg[p];     {noua adresa a elementului p}
sf cât timp;
```

Exercitii:

Pentru exemplificare studiati problemele de la sectiunea de verificare si solutiile din anexa.

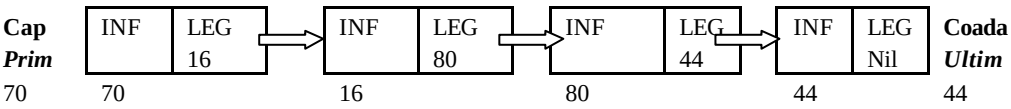
3.3.2. Coada

Coada (coada de asteptare) este o lista liniara simplu înlantuita cu proprietatile:

Inserarea unui element se face numai la coada stivei, numit **ultim element al stivei**.

Stergerea unui element se face numai la capul stivei, numit **prim element al stivei**.

Legatura elementului din coada stivei, numit **ultim,** este ***nil*** (nimic).



Un exemplu clasic de coada este “Trecerea vagoanelor, pe o linie de cale ferata, printr-un tunel, într-un singur sens”. Este evident ca vagoanele intra si ies în aceeasi ordine. Deci locomotiva, vagonul 1, 2, etc.

Cozile permit trei operatii elementare: inserarea la coada (ultim), stergerea de la cap (prim) si parcurgerea.

În descrierile algoritmilor, pentru operatiile asupra cozilor, vom folosi în reprezentarea generala aceleasi notatii ca în cazul stivelor:

Vom folosi variabilele

p, q, prim, ultim : adresa_element;
Pentru initializarea cozii avem:

aloc (***prim);***
leg [prim] = nil;
ultim = prim;.

Alocarea unei zone noi de memorie se realizeaza cu **`aloc`** (adresa_element)
Disponibilizarea unei zone de memorie se realizeaza cu **`disp`** (adresa_element).
Inserarea unui element în coada, deci **ultim** are o anumita valoare.

Aloc (p); {alocarea memorie pentru noul element}
inf [p] = <expresie>; **{**atribuirea valorii zonei de informatie}
leg [p] = nil; **{**p fiind ultimul element, adresa de legatura este nil}
leg [ultim] = p; **{**adresa_de_legatura a ultimului element este noul p}
ultim = p; {noul ultim se afla acum la noua adresa p }

Stergerea unui element din coada, deci **prim** are o anumita valoare)

P = prim; {se salveaza adresa lui **prim** pentru a nu se pierde}
prim = leg[prim]; {noua adresa a lui **prim** este legatura sa}
Disp (p); {se disponibilizeaza (sterge) zona de adresa p}

Parcurgerea elementelor unei cozi pornind de la **prim.**

P = prim; {se salveaza adresa **prim** pentru a nu se pierde}
Cât timp p <> nil executa {conditia poate fi schimbata dupa cerintele de parcurgere}
{**Operatie asupra elementului de la adresa p**}
p = leg[p]; {noua adresa a elementului p}
sf cât timp;

Exercitii:
Pentru exemplificare studiati problemele de la sectiunea de verificare si solutiile din anexa.

3.3.3. Lista circulara

Lista circulara este o lista liniara simplu înlantuita, definita în mod similar cu coada. În plus are proprietatea ca:

leg [ultim] = prim;

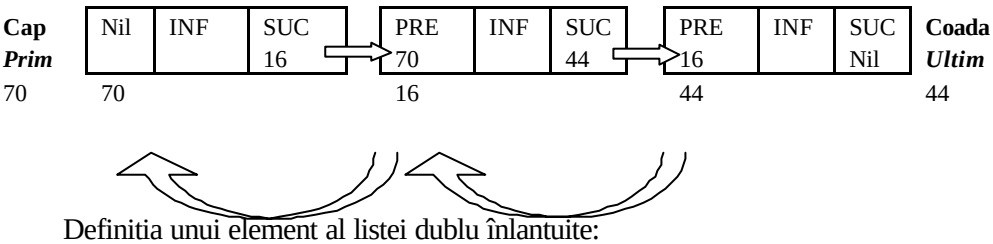
Pe lânga operatia de parcurgere care este similara cu cea de la coada, operatiile de inserare si stergere se pot efectua la orice adresa **q** convenabila.

Inserarea dupa elementul de la adresa **q**.

Aloc (p); {alocarea memorie pentru noul element}
inf [p] = <expresie>; **{**atribuirea valorii zonei de informatie}
leg [p] = leg [q] ; **{**adresa de legatura a lui p este cea a lui q}
leg [q] = p; **{**adresa_de_legatura a lui q este noul p}
Stergerea elementului de dupa elementul de la adresa q (leg [q]).
P = leg[q]; {se salveaza adresa lui **leg [q]** pentru a nu se pierde}
leg[q] = leg[p]; {noua legatura a lui **q** este legatura lui **p**}
Disp (p); {se disponibilizeaza (sterge) zona de adresa p}

3.4. Lista dublu înlantuita

Asa cum am definit *lista dublu înlantuita* în paragraful 3.1., este o lista în care un element este format din INFormatie si doua adrese de legatura, înspre elementul PREdecesor si respectiv SUCcesor. Lista dublu înlantuita are doua capete pe care le vom nota la fel ca în cazul cozii cu prim si respectiv ultim.



Definitia unui element al listei dublu înlantuite:

Element = **articol**
pre: adresa_element;
suc: adresa_element;
inf: informatia_de_un_anumit_tip;

Pe lânga operatia de parcurgere care este similara cu cea de la coada sau stiva, cu diferenta ca deplasarea se face spre PRE sau spre SUC, operatiile de inserare si stergere se pot efectua la orice adresa **q** convenabila.

Inserarea înaintea elementului de la adresa **q**.

Aloc (p); {alocarea memorie pentru noul element}
inf [p] = <expresie>; **{**atribuirea valorii zonei de informatie}

pre [p] = pre [q] ; **{**legatura pre a lui p este legatura pre a lui q}
suc [p] = q ; **{**suc lui p este q}
suc [pre [p]] = p; **{**predecesorul lui p are ca succesor pe p}
pre [q] = p; **{**adresa_de_legatura a lui q este noul p}

Exercitiu:

Verificati pentru lista din figura de mai sus presupunând noua adresa a lui **p=60 si q=16.**

Stergerea elementului aflat la adresa succesorului lui **q (deci de la suc [q]).**

P = suc[q]; {se salveaza adresa lui **suc [q]** pentru a nu se pierde}
suc[q] = suc[p]; {noul succesor a lui **q** este lui **p**}
pre [suc[p]] = q; { succesorul lui **p** are ca predecesor pe **p**}
Disp (p); {se disponibilizeaza (sterge) zona de adresa p}

Exercitiu:

Verificati pentru lista din figura de mai sus presupunând **q=70**.

Listele dublu înlantuite se folosesc la reprezentarea arborilor binari, despre care vom vorbi în continuare.

4. Structuri de date dinamice

4.1. Tipuri de variabile (dupa durata de viata)

1) *globale*
- declarate în programul principal
- alocate în segmentul de date
- durata lor de viata este durata de executie a programului
- se initializeaza la declarare cu valori identice lui 0 (0 pentru numere, stringul vid pentru string-uri, multimea vida pentru multimi, etc) - în Borland (Turbo) Pascal 7.0

2) *locale unui subprogram*
- sunt formate din
- variabilele declarate în subprogramul respectiv
- parametrii transmisi prin valoare
- sunt alocate în stiva de executie
- alocarea se face automat, când subprogramul de care ele tin este lansat în executie = înregistrarea de activare a acestuia este pusa în stiva de executie (vezi CTRL+F3)
- dealocarea se face automat, când subprogramul îsi termina executia

- în stiva de executie pot fi mai multe subprograme
- cel din vârful stivei este cel activ (care se executa curent)
- celelalte au apelat fiecare subprogramul de deasupra lor din stiva
- nu sunt initializate automat

3) *dinamice*

- nu au un nume propriu (se numesc 'anonime')
- sunt referite prin intermediul pointerilor
- sunt alocate si dealocate de catre programator
- sunt alocate în memorie în "heap"

heap = gramada, ansamblu, zona de memorie dinamica

= împreuna cu stiva de executie ocupa portiunea de memorie ramasa ne ocupata de programul executabil (cod + segmentul de date), fiecare pornind de la un capat spre mijloc

Deosebirea dintre stiva si heap consta în:

- **stiva** este o zona de memorie de dimensiune fixa, numit segment de memorie, (in Pascal minim 16384, maxim 65536 bytes) care permite stocarea programelor si a variabilelor la executarea acestora. Apelarea subprogramelor permite alocarea sau dealocarea de memorie pentru variabilele locale. Alocarea pe stiva se face static la executarea programelor.

- heap-ul este o zona de memorie formata din totalitatea segmentelor de memorie disponibile pe un sistem de calcul. În Pascal pentru Windows dimensiunea maxima ce poate fi alocata pe heap este de 655360 bytes, iar în UNIX (FreePascal) dimensiunea depinde doar de memoria RAM a sistemului, disponibila la momentul alocarii. Alocarea se face dinamic.

- dimensiunea creste sau descreste dupa cum apar instructiuni de alocare sau dealocare de variabile dinamice
- este format din blocuri de memorie (segmente)
- are o lista de blocuri libere
- pe masura ce se fac alocari si dealocari, heap-ul este tot mai fragmentat
- functii Turbo (Borland) Pascal pentru heap:
  - MaxAvail - dimensiunea celui mai mare bloc liber
  - MemAvail - suma dimensiunilor blocurilor libere

4.2. Tipul pointer

Este un tip de date definit de utilizator si are la baza urmatoarele elemente:

a) domeniu

- valorile sale sunt adrese de memorie ale altor obiecte din program (variabile, subprograme)
- mai exact: multimea valorilor adreselor din memoria RAM disponibila programului, plus o valoare particulara notata (în Pascal) cu Nil

b) operatii
- de atribuire: (:=)
- de comparare: (= si <>)
- de alocare a variabilei dinamice referite de pointer:
  New (Pascal standard) si *GetMem* (in Turbo si Borland Pascal)
- de dealocare a variabilei dinamice referite de pointer:
  Dispose (Pascal standard) si *FreeMem* (in Turbo si Borland Pascal)

4.2.1. Declararea tipului pointer

Variabilele de tip pointer refera alte obiecte. Deoarece Pascal este un limbaj puternic tipizat, tipul unui pointer (TP) va indica tipul obiectelor referite de variabilele de tipul pointer TP:

```
type
  tip_pointer = ^tip_de_baza
```

va indica faptul ca orice variabila de tipul "tip_pointer" va avea ca valoare o adresa a unei variabile (obiect) de tipul "tip_de_baza". Aceasta declaratie introduce un nou tip de date, cu numele "tip_pointer" în sistemul de tipuri al programului în care ea apare, dupa toate regulile cunoscute.

Tipul de date "tip_de_baza" poate fi
- predefinit
- definit deja (declaratia lui apare înaintea declaratiei lui "tip_pointer"
- definit dupa declaratia de mai sus (numai în cazul declaratiei tipului pointer se permite ca un nume (in cazul nostru "tip_de_baza" sa fie referit înainte de a fi declarat

Exemple: *(vom prefixa cu P numele tipurilor pointer si cu T numele celorlalte tipuri de date definite)*

```
type
  PInteger = ^Integer;  { variabilele de tip PInteger vor fi pointeri la
  întregi, adica vor contine adresa unor variabile de tip Integer }
  TVector = Array[1..10] of Real;
  PVector = ^TVector;      { TVector a fost definit înainte de PVector }
  PMatrice = ^TMatrice;            { TMatrice este definit dupa }
  TMatrice = Array[1..5, 1..10] of Integer;
```

Limbajele Borland si Turbo Pascal au un tip pointer predefinit cu numele Pointer. Acest tip se refera la pointeri fara tip si se poate folosi în conversii sau în explicarea polimorfismului.

Exemplu:

```
var
  p: PMatrice;
  q: Pointer;
begin
  New(p);                  { aloca p^ }
  q := p;                  { atribuire corecta }
  p := q;            { eroare de compilare, atribuire de tip incorecta }
  p := PMatrice(q);        { conversie explicita de pointeri }
end.
```

4.2.2. Declararea de variabile de tip pointer

Odata declarat tipul pointer, se pot declara variabile de tipul sau, respectiv declaratia normala a variabilelor.

Exemple:

```
var
  pi: PInteger;
  pi2: ^Integer;    { tipul ^Integer este un tip anonim }
  v: PVector;
  m: PMatrice;
```

4.3. Semantica variabilelor de tip pointer. Operatii

4.3.1. Variabile pointer si variabile dinamice asociate

Ori de câte ori lucram cu variabile de tip pointer, trebuie sa stim ca o astfel de variabila are o dubla semnificatie (sau ca avem de-a face cu doua variabile, total diferite una de alta). Cele doua variabile au sensul:

1) variabilei pointer obisnuita, notat prin numele variabilei, care are
   - un domeniu de vizibilitate (dat de locul în care apare declaratia sa)
   - o durata de viata (variabila poate fi globala, locala sau dinamica)
   - o valoare (adresa unui alt obiect din memorie)

2) obiectului pe care îl refera variabila pointer (a carui adresa o contine ca valoare), notat prin numele variabilei pointer urmat de ^

Exemplu:

```
var
  pi: PInteger;
```

- *pi* este o variabila pointer de tipul PInteger, deci ea poate contine adresa unei variabile de tip Integer din program
- *pi^* este variabila de tip Integer a carei adresa este continuta de pi.

Terminologie
pi este *variabila pointer*
*pi^* este *variabila dinamica referita de pi*

4.3.2. Operatia de atribuire. Semantica.

Având de-a face cu doua variabile distincte, semantica operatiei de atribuire trebuie discutata pentru fiecare caz în parte:

a) Atribuirea de pointeri
Pentru declaratia
```
var
  pi1, pi2: PInteger;
```
atribuirea
```
  pi1 := pi2;
```
are semnificatia
- se atribuie lui pi1 valoarea lui pi2, adica
- pi1 va contine aceeasi valoare ca si pi2
- ele vor referi aceeasi variabila dinamica, adica dupa atribuire pi1^ si pi2^ vor referi acelasi obiect

b) Atribuirea de valori (ale variabilelor dinamice)
Pentru declaratia
```
var
  pi1, pi2: PInteger;
```
atribuirea
```
  pi1^ := pi2^;
```
are semnificatia
- se atribuie lui pi1^ valoarea lui pi2^, adica
- pi1^ va contine aceeasi valoare ca si pi2^
- pi1 si pi2 vor referi variabile dinamice diferite;
- valoarea variabilei dinamice pi2^ este atribuita variabilei dinamice pi1^
- pi1^ si pi2^ ramân în continuare obiecte diferite, însa ele vor avea aceeasi valoare

c) Lucrul cu obiecte existente în program prin intermediul pointerilor

Pentru declaratiile
var
pi: PInteger;
i: Integer;

instructiunea de atribuire
pi := Addr(i)

are efectul unei atribuiri de pointeri
- functia Addr(o) întoarce adresa unui obiect o din memoria programului
- în cazul nostru, Addr(i) întoarce adresa variabilei i

Exemplu: Programul PtrDemo1.PAS

4.3.3. Alocarea variabilelor dinamice si initializarea pointerilor

In lucrul cu pointeri, distingem în mod normal urmatoarele momente:
- declararea tipurilor pointer pe care vrem sa le folosim
- declararea variabilelor de tip pointer, acolo unde vrem sa le folosim
- alocarea variabilelor dinamice referite de pointeri
- lucrul propriu-zis cu pointeri
- dealocarea variabilelor dinamice referite de pointeri

Atribuirea de valori unei variabile pointer se face în Borland Pascal prin:
- folosirea procedurilor standard New si GetMem
- folosirea operatorului @ ('at')
- folosirea functiei Addr
- folosirea functiei Ptr.

Dintre acestea, New si GetMem realizeaza si alocarea în heap a variabilei dinamice referita de pointer.

Exista o valoare constanta, notata cu Nil, care semnifica faptul ca pointerul care are aceasta valoare nu refera nimic (nu exista înca variabila dinamica referita de el).

Procedura standard New are declaratia:

Procedure New(var P: Pointer);

si care realizeaza urmatoarele:

- aloca variabila dinamica P^
- atribuie ca valoare lui P adresa variabilei dinamice P^.

Tipul Pointer din declaratie este unul general. Procedura primeste ca parametru o variabila P de orice tip pointer recunoscut de sistemul de tipuri. Alocarea variabilei dinamice P^ se va face în functie de tipul lui P.

Fata de limbajul Pascal standard, exista în Borland si Turbo Pascal extensii ale procedurilor de alocare si dealocare New si Dispose, care vor fi discutate la programarea orientata pe obiecte (GetMem, FreeMem si @).

Lucrul cu variabile pointer înseamna operatii de atribuire si comparare.

Trebuie avute în vedere urmatoarele:

- prima operatie asupra unei variabile pointer este initializarea acesteia, care se poate face:
  - prin New, GetMem, @, Addr sau Ptr
  - prin atribuire de pointeri
  - prin atribuire de pointeri se pierde vechea valoare a variabilei dinamice care apare în partea stânga a operatorului de atribuire

Exemplu: la atribuirea

pi1 := pi2;

vechea valoare a lui pi1 (deci adresa variabilei dinamice pi1^) se va pierde; în locul ei se va trece (scrie peste) adresa variabilei dinamice pi2^).

- durata de viata a variabilei p^ poate sa depaseasca domeniul de vizibilitate al variabilei p.

Exemplu: Programul DuV_Ptr;

4.3.4. Dealocarea variabilelor dinamice

Dupa ce am terminat lucrul cu o variabila dinamica, ea trebuie dealocata explicit. Dealocarea se face în functie de modul în care s-a facut alocarea dupa cum urmeaza

Alocarea s-a facut cu	Dealocarea se face cu
New	Dispose
GetMem	FreeMem

Procedura standard Dispose are declaratia

Procedure Dispose(var P: Pointer);

ea realizeaza

- dealocarea din heap a variabilei dinamice referite de P, adica a lui P^
- zona ocupata de P^ este pusa în lista de blocuri libere a heap-ului
- deinitializarea valorii lui P: referirea lui P dupa Dispose(P) provoaca o eroare de executie.

Exemplu complex de lucru cu Pointeri: COMPLEXP.PAS, unit-ul UCOMPL_P.PAS

Aplicatie (p434.pas)***:***

Sa se verifice daca trei puncte date prin coordonatele lor carteziene sunt vârfurile unui triunghi echilateral.

```
Program triunghi_echilateral;
uses crt;
TYPE punct=record
          a,o:real;
      end;
Var A,B,C :^punct;
   Lab,Lac,Lbc :real;
Begin
clrscr;
New(A);
Write('A.a=');readln(A^.a);
Write('A.o=');readln(A^.o);
New(B);
Write('B.a=');readln(B^.a );
Write ('B.o=');readln(B^.o);
New(C);
Write('C.a=');readln(C^.a );
Write('C.o=');readln(C^.o);
LAB:=sqrt(sqr(A^.a-B^.a)+sqr(A^.o-B^.o));
LBC:=sqrt(sqr(B^.a-C^.a)+sqr(B^.o-C^.o));
LAC:=sqrt(sqr(A^.a-C^.a)+sqr(A^.o-C^.o));
Dispose(A);Dispose(b);Dispose(C);
If (LAB=LBC)and(LBC=LAC) then
    Writeln('ABC- echilateral')
                  Else
    Writeln('ABC- nu este echilateral');
Readln
End.
```

4.3.5. Probleme propuse

1. Se dau coordonatele carteziene a patru puncte. Sa se verifice daca ele sunt vârfurile unui dreptunghi. Se va folosi tipul punct definit în aplicatia anterioara.
2. Sa se afiseze ora exacta si data exacta folosind variabile alocate dinamic.
3. Sa se afiseze cel mai mare divizor comun si cel mai mic multiplu comun a doua numere naturale memorate dinamic.
4. Se dau 4 puncte prin coordonatele lor carteziene si un cerc prin centrul sau si raza. Sa se afiseze numarul de puncte interioare si exterioare cercului.
5. Se dau 2 numere complexe. Sa se scrie un program care permite efectuarea urmatoarelor operatii: suma, diferenta, modul, conjugat, produs, cât si putere.

Se va folosi tipul complex definit astfel:

```
TYPE complex=^element;
     Element=Record
          Re, Im:real;
          End;
```

4.4. Structuri de date dinamice. Aplicatii.

Structurile date dinamice sunt structuri de date cu numar variabil de componente. Aceste structuri folosesc alocarea dinamica fiind mult mai avantajoase din punct de vedere al utilizarii judicioase a memoriei în comparatie cu cele statice. În general o componenta a unei structuri dinamice are forma:

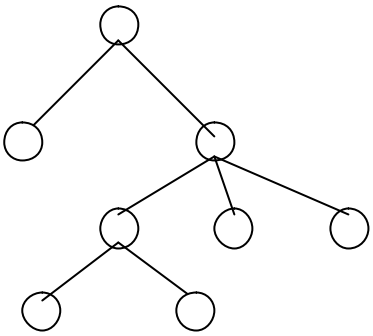
Informatie	adrese de legatura cu componenta urmatoare

Clasificarea structurilor de date dinamice:

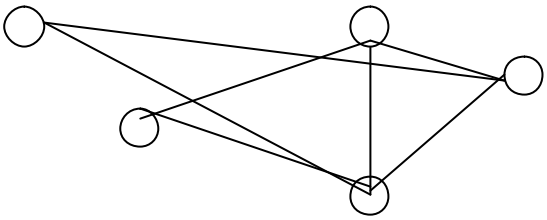
a) În functie de tipul de legatura

liste liniare - componentele structurii se afla pe un singur nivel, legatura 1:1.

arbori – componentele structurii sunt grupate pe niveluri.



retele – între componentele structurii pot exista legaturi între oricare elemente



în general asupra (nodurilor) componentelor unei structuri se fac urmatoarele operatii:

- introducere de componente
- stergeri de componente
- parcurgere a componentelor si prelucrarea informatiilor asociate

În continuare vom prezenta câteva cazuri speciale de liste liniare (stiva, coada, liste simplu înlantuite, liste dublu înlantiute si liste circulare) alocate dinamic.

În TurboPascal un nod din lista va fi utilizat prin variabile de tip reper definit astfel:

```
TYPE reper=^Element;
     Element=record
                Informatie:tip;
                Leg:reper;
             End;
```

Observatie:

În TurboPascal orice tip folosit trebuie declarat anterior. Pentru tipul reper este permisa folosirea unui tip si declararea lui ulterioara. Aici este singurul loc unde TurboPascal face aceasta concesiune.

4.4.1. Aplicatia stiva (p441.pas):

a) Sa se construiasca o stiva cu numere naturale, citirea numerelor terminându-se cu 0 (care nu face parte din stiva);

b) Sa se afiseze numerele din stiva;

c) Sa se stearga componenta din vârful stivei;

d) Sa se afiseze numerele din stiva dupa stergerea componentei de la pct. c).

```
program stiva;
uses crt;
type reper=^element;
element=record
inf:word;
leg:reper;
end;
var
vf:reper;

procedure creare(var vf:reper);
var n:word;
    p:reper;
begin
vf:=nil;
write('numar=');readln(n);
while n<>0 do
begin
new(p);
p^.inf:=n;
p^.leg:=vf;
vf:=p;
write('numar=');readln(n);
end;
end;
```

```
procedure afisare(vf:reper);
begin
if vf=nil then
writeln('stiva vida')
else
repeat
write(vf^.inf,' ');
vf:=vf^.leg;
until vf=nil;
end;
procedure stergere(var vf:reper);
var p:reper;
begin
p:=vf;
vf:=vf^.leg;
dispose(p);
end;

begin{p.p.}
writeln('a) creare stiva');
creare(vf);
writeln('b) afisare');
afisare(vf); writeln;
writeln('c) stergere...');
stergere(vf);
writeln('d) afisare dupa stergere');
afisare(vf);
end.
```

4.4.2. Aplicatia coada (p442.pas):

a) Sa se construiasca o coada cu numere naturale, citirea numerelor se termina o data cu introducerea numarului 0 (care nu face parte din coada).

b) Sa se afiseze numerele din coada.

c) Sa se stearga o componenta din coada.

d) Sa se afiseze componentele din coada dupa operatia de la punctul c)

```
program coada;
uses crt;
type reper=^element;
     element=record
               inf:integer;
               leg:reper;
             end;
var p,u:reper;

procedure creare(var p,u:reper);
var q:reper; n:integer;
begin
write('nr=');readln(n);
p:=nil;  {coada vina}
while n<>0 do
 begin
  new(q);
  q^.inf:=n;
  q^.leg:=nil;
  if p=nil then
   begin
    p:=q;
    u:=q;
   end
          else
   begin
    u^.leg:=q;
    u:=q;
   end;
  write('nr=');readln(n);
 end;
end; {creare}

procedure afisare(p,u:reper);
var q:reper;
begin
q:=p;
if q=nil then
 writeln('coada vida')
        else
begin
writeln('coada contine:');
  repeat
  write(q^.inf,' ');
  q:=q^.leg
  until q=nil
 end;
writeln;
end;{afisare}

procedure stergere(var p,u:reper);
var q:reper;
begin
if p=nil then
 writeln('Coada vida!')
        else
 begin
  q:=p;
  if p=u then
   p:=nil
        else
   p:=p^.leg;
  dispose(q);
 end;
end;{stergere}


begin {progr. principal}
clrscr;
writeln('a) creare coada');
creare(p,u);
writeln('b) afisare');
afisare(p,u);
writeln('c) stergere');
stergere(p,u);
writeln('d) afisare dupa stergere');
afisare(p,u);
readln;
end.
```

4.4.3. Aplicatia lista simplu înlantuita (p443.pas):

a) Sa se construiasca o lista simplu înlantiuta cu numere naturale, citirea numerelor se termina o data cu introducerea numarului 0 (care nu face parte din coada).

b) Sa se insereze o componenta în lista dupa o componenta data prin informatia ei.

c) Sa se insereze o componenta în lista inaintea unei componente data prin informatia ei.

d) Sa se stearga o componenta data prin informatia ei.

Dupa fiecare subpunct se va afisa lista.

```
program l_s_i;
uses crt;
type reper=^element;
     element=record
               inf:integer;
               leg:reper;
             end;
var s1,s2:reper; info:integer;

procedure creare(var s1,s2:reper);
var q:reper; n:integer;
begin
 new(s1);new(s2);
 s1^.leg:=s2;
 s2^.leg:=nil;
 write('nr=');readln(n);
 while n<>0 do
  begin
   new(q);
   s2^.inf:=n;
   q^.leg:=nil;
   s2^.leg:=q;
   s2:=q;
   write('nr=');readln(n);
  end;
end;{creare}

function adresa(s1,s2:reper;info:integer):reper;
```

```
{Aceasta functie returneaza adresa nodului cu
informatia egala cu  cea a parametrului info}
var q:reper; sw:boolean;
begin
 q:=s1^.leg;
 sw:=false;
 while (q<>s2) and (not sw) do
  if q^.inf=info then
   sw:=true
                else
   q:=q^.leg;
 adresa:=q;
end;{adresa}

procedure inserare_i(info:integer);
var w,q,r:reper; n:integer;
begin
 r:=adresa(s1,s2,info);
 if r=s2 then
  write('nu exista nodul inaintea caruia sa inseram')
        else
  begin
   q:=s1;
   while q^.leg<>r do
    q:=q^.leg;
   write('info din nodul inserat=');readln(n);
   new(w);
   w^.inf:=n;
   w^.leg:=r;
   q^.leg:=w;
  end;
end;{inserare_i}

procedure inserare_d(info:integer);
var r,w:reper; n:integer;
begin
 r:=adresa(s1,s2,info);
 if r=s2 then
  write('nu ex. nod dupa care sa inserez')
        else
  begin
   new(w);
```

```
   write('info=');readln(n);
   w^.inf:=n;
   w^.leg:=r^.leg;
   r^.leg:=w;
  end;
end;{inserare_d}

procedure stergere(info:integer);
var q,r,w:reper;
begin
 r:=adresa(s1,s2,info);
 if r=s2 then
  write('nu ex. nodul')
        else
  begin
   q:=s1;
   while q^.leg<>r do
    q:=q^.leg;
   q^.leg:=r^.leg;
   dispose(r);
  end;
end;{stergere}
procedure afisare(s1,s2:reper);
var q:reper;
begin
 q:=s1^.leg;
 if q=s2 then
  write('lista vida')
        else
   repeat
    write(q^.inf,' ');
    q:=q^.leg;
   until q=s2;
writeln;
end;{afisare}

begin{p.p.}
clrscr;
writeln('a) creare lista simplu înlantuita');
creare(s1,s2);
writeln('Afisare:');
afisare(s1,s2);
```

```
writeln('b) inserare inainte');
write('informatia no inaintea caruia se insereaza=');
readln(info);
inserare_i(info);
writeln('Afisare:');
afisare(s1,s2);
writeln('c) inserare dupa');
write('informatia nodului dupa care se insereaza=');
readln(info);
inserare_d(info);
writeln('Afisare:');
afisare(s1,s2);
writeln('d) stergere');
write('informatia din nod ce va fi sters=');
readln(info);
stergere(info);
writeln('Afisare:');
afisare(s1,s2);
readln;
end.
```

4.4.4. Aplicatia Lista dublu înlantuita (p444.pas):

a) Sa se construiasca o lista dublu înlantiuta cu numere naturale, citirea numerelor se termina o data cu introducerea numarului 0 (care nu face parte din coada).

b) Sa se insereze o componenta în lista dupa o componenta data prin informatia ei.

c) Sa se insereze o componenta în lista inaintea unei componente data prin informatia ei.

d) Sa se stearga o componenta data prin informatia ei.

e) Folosind interschimbari de adrese ale componentelor sa se formeze o lista cu numere ordonate crescator.

f) Sa se afiseze lista parcurgând-o de la stânga la dreapta si de la dreapta la stânga.

Dupa fiecare subpunct se va afisa lista.

```
program l_d_i;
uses crt;
type nod=^elem;
    elem=record
         st,dr:nod;
```

```
          inf:integer;
         end;
var s1,s2:nod; n,m:integer;

procedure creare(var s1,s2:nod);
var n:integer; p:nod;
begin
 new(s1);
 new(s2);
 s1^.dr:=s2;
 s2^.st:=s1;
 write('numar=');readln(n);
 while n<>0 do
  begin
   s2^.inf:=n;
   new(p);
   p^.st:=s2;
   s2^.dr:=p;
   s2:=p;
   write('numar=');readln(n);
  end;
end;{creare}

function caut(s1,s2:nod;n:integer):nod;
{functia returneaza adresa nodului cu
informatia egala cu parametrul n}
var p:nod;
begin
 p:=s1^.dr;
 while(p<>s2) and(p^.inf<>n) do
 p:=p^.dr;
 if p=s2 then
  caut:=nil
        else
  caut:=p;
end;{caut}

procedure stergere(s1,s2:nod;n:integer);
var q,r,p:nod;
begin
 p:=caut(s1,s2,n);
 if p<>nil then
```

```
  begin
   q:=p^.dr;
   r:=p^.st;
   r^.dr:=q;
   q^.st:=r;
   dispose(p);
  end
          else
  writeln('nu exista nod cu inf=',n);
end;{stergere}

procedure afisaresd(s1,s2:nod);
{afisarea informatiilor în sensul
s1--->s2}
var p:nod;
begin
 p:=s1^.dr;
 while p<>s2 do
  begin
   write(p^.inf,' ');
   p:=p^.dr;
  end;
 writeln;
end;{afisaresd}

procedure afisareds(s1,s2:nod);
{afisarea informatiilor în sensul
s2--->s1}
var p:nod;
begin
 p:=s2^.st;
 while p<>s1 do
  begin
   write(p^.inf,' ');
   p:=p^.st;
  end;
 writeln;
end;{afisareds}

procedure inserare_dupa(s1,s2:nod;n,m:integer);
var p,q:nod;
begin
```

```
 p:=caut(s1,s2,n);
  if p= nil then
   writeln ('nu exista nod cu inf=',n)
            else
   begin
    new(q);
    q^.inf:=m;
    q^.dr:=p^.dr;
    p^.dr:=q;
    q^.st:=p;
   end;
end;{inserare_dupa}

procedure inserare_inainte(s1,s2:nod;n,m:integer);
var p,q:nod;
begin
 p:=caut(s1,s2,n);
 if p=nil then
  writeln('nu exista nod cu inf=',n)
          else
  begin
   new(q);
   q^.inf:=m;
   q^.st:=p^.st;
   q^.dr:=p;
   (p^.st)^.dr:=q;
   p^.st:=q;
  end;
end;{inserare_inainte}

procedure interschimbare(s,p,q,r:nod);
begin
 s^.dr:=q;
 q^.st:=s;
 q^.dr:=p;
 p^.st:=q;
 p^.dr:=r;
 r^.st:=p;
end;{interschimbare}

procedure ord_cresc(s1,s2:nod);
var p:nod; sw:boolean;
```

```
begin
 sw:=true;
 while sw do
  begin
   sw:=false;
   p:=s1^.dr;
   while(p^.dr<>s2) do
     if p^.inf>(p^.dr)^.inf then
      begin
       interschimbare(p^.st,p,p^.dr,(p^.dr)^.dr);
       sw:=true;
      end
                        else
      p:=p^.dr;
  end;
end;{ord_cresc}

begin {p.p.}
clrscr;
writeln('a) creare lista dublu înlantuita');
creare(s1,s2);
writeln('afisare st-dr');
afisaresd(s1,s2);
writeln('afisare dr-st');
afisareds(s1,s2);
writeln('b) inserare dupa');
write('inserare dupa nodul cu info=');
readln(n);
write('informatie nod inserat=');
readln(m);
inserare_dupa(s1,s2,n,m);
writeln('lista dupa inserare=');
afisaresd(s1,s2);
writeln('c) inserare inainte');
write('inserare inaintea nodului cu info=');
readln(n);
write('informatie nod inserat=');
readln(m);
inserare_inainte(s1,s2,n,m);
writeln('lista dupa inserare');
afisaresd(s1,s2);
writeln('d) stergere nod');
```

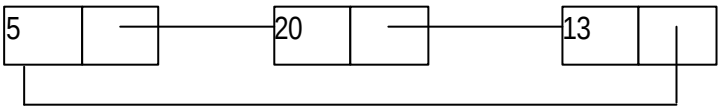
```
write('informatie nod ce va fi sters=');
readln(n);
stergere(s1,s2,n);
writeln('dupa stergere');
afisaresd(s1,s2);
afisareds(s1,s2);
writeln('e) ordonare');
ord_cresc(s1,s2);
writeln('lista dupa ordonare crescatoare:');
afisaresd(s1,s2);
readln;
end.
```

4.4.5. Aplicatia lista circulara (p445.pas):

O lista circulara este o lista simplu înlantuita în care ultima componenta contine adresa primei componente.

Exemplu:

Lista circulara cu numerele 5, 20, 13 poate fi reprezentata astfel:



Pentru a lucra cu liste circulare este nevoie de o singura variabila dinamica în care sa retinem adresa unui nod (spre exemplu adresa primului nod introdus).

Operatiile asupra listei circulare sunt aceleasi ca cele de la lista simplu înlantuita. Aceste operatii le vom prezenta în aplicatia urmatoarea.

a) Sa se construiasca o lista circulara care sa contina n numere naturale distincte citite de la tastatura.

b) Sa se afiseze numerele din lista creata la punctul a).

c) Sa se insereze un numar citit de la tastatura înaintea si dupa cel mai mare numar din lista circulara. Apoi sa se afiseze numerele din lista.

d) Sa se stearga din lista circulara toate numerele prime. Apoi sa se afiseze numerele din lista.

```
program lista_circulare;
uses crt;
type nod=^element;
   element=record
```

```
                inf:byte;
                urm:nod;
              end;
var pr:nod;
    x:integer;
procedure creare(var pr:nod);
var i,x,n:integer;p,q:nod;
begin
 write('n='); readln(n);
 pr:=nil;
 for i:=1 to n do
  begin
   new(p);
   write('numar='); readln(x);
   p^.inf:=x;
   if pr=nil then
    begin
     p^.urm:=p;
     pr:=p;
     q:=p;
    end
            else
    begin
     q^.urm:=p;
     q:=p;
    end;
  end;
  q^.urm:=pr;
end;{creare}

procedure afisare(pr:nod);
var p:nod;
begin
 p:=pr;
 repeat
     write(' ', p^.inf);
     p:=p^.urm;
 until p=pr;
 writeln;
end;{afisare}

function maxim(pr:nod):integer;
```

```
var p:nod; max:integer;
begin
 p:=pr^.urm;
 max:=p^.inf;
 while p<>pr do
  begin
   if p^. inf>max then max:=p^.inf;
   p:=p^.urm;
  end;
 maxim:=max;
end;{maxim}

function adresa_inainte(pr,q:nod):nod;
var p:nod;
begin
 p:=pr;
 while p^.urm<>q do
  p:=p^.urm;
 adresa_inainte:=p;
end;{adresa_inainte}


procedure inserare_i_d(pr:nod; x:integer);
var p,q,r:nod;max:integer;
begin
 p:=pr;
 max:=maxim(pr);
 while p^.urm<>pr do
  if p^.inf=max then
   begin
    {inserarea inainte de nodul p}
    new(q);
    r:=adresa_inainte(pr,p);
    q^.inf:=x;
    q^.urm:=p;
    r^.urm:=q;
    {inserare dupa nodul p}
    new(q);
    q^.inf:=x;
    q^.urm:=p^.urm;
    p^.urm:=q;
    p:=q;
```

```
    end
            else
    p:=p^.urm;
   {ultimul nod il prelucram separat}
   if p^.inf=max then
    begin
     new(q);
     q^.inf:=x;
     q^.urm:=p^.urm;
     p^.urm:=q;
     p:=q;
    end
  end;{inserared}

  procedure stergere(pr:nod);
  {stergem din lista circurara
  toate numerele prime}
  var q,p:nod;
  function prim(a:integer):boolean;
  var i:integer; sw:boolean;
  begin
   if (a=0)or(a=1) then
    sw:=false
            else
    begin
     sw:=true;
     for i:=2 to trunc(sqrt(a)) do
      if a mod i=0 then
       sw:=false;
    end;
   prim:=sw;
  end;{prim}
  begin
   p:=pr;
   if p<>nil then
    repeat
     if prim(p^.inf) then
      begin
       q:=adresa_inainte(pr,p);
       q^.urm:=p^.urm;
       dispose(p);
       p:=q^.urm;
```

```
    end
                else
    p:=p^.urm;
  until p=pr;
 end;{stergere}

 begin{p.p.}
  clrscr;
  writeln('a) creare lista circulara');
  creare(pr);
  writeln('b) elementele din lista circulara: ');
  afisare(pr);
  writeln('c) inserare nr. inainte si dupa max.');
  write('nr. natural ce va fi inserat=');
  readln(x);
  inserare_i_d(pr,x);
  writeln('afisare dupa inserare:');
  afisare(pr);
  writeln('d) stergere numere prime');
  stergere(pr);
  writeln('afisare dupa stergere:');
  afisare(pr);
  readln;
 end.
```

4.4.6. Probleme propuse

1. Sa se creeze o stiva cu primele n (n citit de la tastatura) numere prime. Apoi sa se afiseze numerele din stiva.

2. Sa se ceeze o coada care sa ɔontina primele n numere patrate perfecte. Apoi sa se afiseze numerele din coada.

3. a) Sa se creeze o lista simplu înlantuita cu numele si înaltimea a n elevi dintr-o clasa.

b) Sa se afiseze lista.

c) Sa se insereze înainte si dupa o componenta ce contine elevi cu înaltimea maxima doi elevi cu datele (nume si înaltime) citite de la tastatura.

d) Sa se afiseze lista.

e) Sa se stearga din lista componentele care au înaltimea minima.

f) Sa se afiseze lista.

4. a) Sa se creeze o lista dublu înlantuita care sa contina numere naturale x<1000000000. Citirea numerelor se încheie tastând 0 (care nu face parte din lista).

b) Sa se afiseze lista.

c) Sa se insereze înainte si dupa fiecare componenta din lista dublu înlantuita ce contine un numar format numai din cifre pare, numarul 0.

d) Sa se afiseze lista.

e) Sa se stearga componentele din lista care contin numere ce au în scrierea lor numai cifre impare.

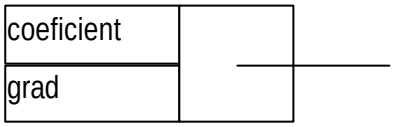
f) Sa se afiseze lista.

5. a) Sa se creeze o lista liniara care sa contina un sir de cuvinte. Citirea datelor se încheie prin tastarea caracterului '*'.

b) Pentru un cuvânt x, dat sa se construiasca doua liste liniare pornind de la lista creata la punctul a), care sa contina cuvintele mai mici în ordine lexicografica, respectiv mai mari decât x.

6. Folosind liste liniare sa se calculeze suma si produsul a doua numere mari.

7. Folosind liste liniare sa se calculeze suma si produsul a doua polinoame. Se va folosi pentru un nod al listei structura:



Exemplu:

Pentru polinomul $P(X)=5X^{100}+20X^{5}-7$ se va folosi lista:

8. Folosind liste liniare (pentru retinerea cifrelor) sa se calculeze puterea a^m, unde a este cifra, iar m este un numar natural mai mic s-au egal cu 1000000000.

9. Folosind liste circulare sa se retina coordonatele vârfurilor unui poligon convex. Apoi sa se calculeze perimetrul si aria poligonului.

IV. Metode si tehnici de programare

Rezolvarea problemelor algoritmice presupune uneori o abordare riguroasa a metodei alese. Acest lucru se impune pentru a reduce complexitatea algoritmului si pentru organizarea eficienta a datelor. Exista astfel clase de probleme rezolvabile si abordabile prin metode specifice.

1. Metoda Greedy

Metoda Geedy (din engleza "lacom") este o metoda care se aplica problemelor în care se da o multime A de n elemente si se cere sa se determine o submultime B a acesteia care sa îndeplineasca anumite conditii, numite *conditii interne*. În general aceste probleme cer ca multimea B sa satisfaca o functie de optim, si se numesc *probleme de optim.*

În majoritatea cazurilor exista mai multe *solutii posibile* ale unei astfel de probleme. Dintre acestea doar unele sunt si *solutii optime*. Tehnica Greedy permite gasirea uneia dintre solutiile optime.

Algoritmul Greedy consta în lacomia sa. La fiecare pas se alege câte un element al multimii A, care satisface conditiile cerute. La alegerea elementului, exista certitudinea ca elementul îndeplineste conditiile problemei si deci face parte din solutia optima.

Pentru a descrie algoritmic metoda consideram algoritmii:

`Alege (A, i, x )` care alege din multimea A un element corespunzator x, aflat pe pozitia i, dintre elementele ramase în A.

`Posibil (B, x, cod)` care verifica daca elementul x este acceptat ca solutie, si îndeplineste conditiile de optim. El poate fi adaugat multimii B. Validarea elementului se face prin variabila cod a carei valoarea de adevar corespunde validitatii (true- elementul este acceptat. False- în caz contrar);

`Adaug (B, x)` care adauga elementul x multimii B si îl elimina din A.

Folosind acesti algoritmi definiti în functie de specificul problemei, algoritmul general al Metodei Greedy este:

```
Algoritm Greedy (A, n, B);
Cod : boolean;

B = {Multimea vida};
Pentru i=1, n,1 executa
```

```
            Alege (A, i, x )
            Posibil (B, x, cod)
            Daca cod atunci
                  Adaug (B, x);
            Sf daca;
Sf pentru;
```

Complexitatea algoritmilor Greedy este în general liniara (O(n)), unde n este numarul elementelor multimii A.

Exemplul 1:

Un exemplu foarte general este problema de determinare a unui numar maxim de elemente dintr-o multime A astfel încât suma lor sa nu depaseasca o valoare data V.

Tehnica Greedy consta în a alege în ordine crescatoare elemente pâna când un nou element adaugat sa depaseasca valoarea ceruta. Pentru a scrie mai usor algoritmul de rezolvare vom apela la algoritmii construiti anterior (ordonarea unui sir).

{tipul **Sir** este un sir de n valori numerice}

```
Algoritm Posibil (x, S, V: numeric, cod : boolean);
Daca S+x<=V atunci
            Cod = adevarat;
Sf daca;

Algoritm Greedy (A: sir, n, V: numeric, B: sir);
Cod : boolean;
s, k: numeric;

BubbleSort (A, n);
k = 0;
S = 0;
Pentru i=1, n,1 executa
            Posibil ( A(i), S, V, cod );
            Daca cod atunci
                  k = k + 1;
                  S = S + A (i);
                  B (k) = A (i);
            Sf daca;
Sf pentru;
```

Exemplul 2

Problema rucsacului

Se da o multime A de obiecte, caracterizate prin utilitate si greutate si un rucsac care poate sa suporte o greutate maxima notata prin G. Se cere sa se stabileasca obiectele din A ce se pot pune în rucsac, astfel ca suma greutatilor lor sa nu depaseasca G, iar utilitatea lor sa fie maxima.

Consideram ca multimea A are n obiecte,

A = {a1, a2, ..., an}

fiecare obiect a_i este caracterizat de

u_i = gradul de utilitate al obiectului

g_i = greutatea acestuia.

O implementare Greedy consta în alegerea elementelor din A în ordinea data de raportul utilitate(a) / greutate(a) cu conditia ca raportul sa fie maximal.

Programul Rucsac.PAS foloseste acesta implementare si respectiv cea în care se ordoneaza elementele lui A. Toate subprogramele necesare sunt incluse in unit-ul URucsac.PAS. Problema este rezolvata în sectiunea Probleme de pe CD la capitolul Metode, Greedy, printr-un algoritm clasic.

Probleme propuse

1. Se dau n bancnote fiecare exprimând o valoare v_i, i=1,n. Sa se determine o modalitate de plata a unei valori cât mai apropiata de o suma S, folosind un numar minim de bancnote.

2. Se dau n tevi de lungimi L_i , i=1,n. Sa se acopere un tronson de lungime M folosind un numar cât mai mare de tevi.

3. Pe n scaune asezate în cerc se afla n-1 copii, cel mult unul pe un scaun. La un moment dat scaunul liber poate fi ocupat de un alt copil a carui scaun devine liber. Dâdu-se o configuratie initiala de asezare a copiilor si una finala, sa se determine modul în care copii se pot reaseza ajungând de la configuratia initiala la cea finala.

2. Metoda Divide et Impera

Divide et Impera (“împarte si stapâneste”) este o metoda care consta în:

- Descompunerea problemei ce trebuie rezolvata într-un anumit numar de subprobleme mai mici, în general de aceeasi natura;

- Rezolvarea succesiva si independenta a fiecarei subprobleme;
- Combinarea tuturor solutiilor fiecarei subprobleme pentru a obtine solutia întregii probleme.

Modelul general al algoritmului Divide et Impera se poate descrie în doua moduri:

A)Recursiv, metoda care în situatiile în care numarul de subprobleme este mare devine ineficient;

B) Iterativ, când pentru fiecare subproblema în parte se aloca, daca este necesar un spatiu de memorie.

Algoritmul Divide et Impera_recursiv, atunci când problema se descompune doar în doua subprobleme, este:

Algoritm divimp (p, q, a);
Daca q – p <= eror **atunci**
prelucrare (p, q, a);
altfel
împarte (p, q, m);
divimp (p, m, b);
divimp (m+1, q, c);
combina (b, c, a);
sf daca

Unde p si q reprezinta capetele multimii unei subprobleme, a, b, c fiind solutiile subproblemelor.

Cazul general presupune împartirea problemei P, a carei solutie S o cautam, în Np probleme ca vor avea fiecare solutiile Sp si o dimensiune Dp. Vom utiliza urmatorii algoritmi:

Rezolva (P, S) care rezolva problema P, atunci când dimensiunea sa este suficient de mica, deci rezolvabila, obtinând solutia S.

Descompune (P, Np, Dp) care descompune problema P în problemele Np de dimensiuni Dp.

Combina (Sp, Dp, S) combina solutiile subproblemelor Sp de dimensiuni Dp si obtine solutia S.

Algoritmul general „Divide et Impera” poate fi descris astfel:

Algoritm DivideEtImpera (P, Dp, S)
Daca „P este rezolvabila” atunci
Rezolva (P, S)
Altfel

Descompune (P, Np, Dp)
Pentru „Fiecare problema Np” executa
DivideEtImpera(Np, Dp, Sp)
SfPentru
Combina (Sp, Dp, S)
SfDaca

În cele ce urmeaza vom prezenta câtiva dintre cei mai reprezentativi algoritmi care folosesc tehnica Divide Et Impera si care fac parte dintr-o categorie importanta, cautarea si sortarea pentru structuri de date mari.

2.1. Cautare si sortare pentru structuri de date

Cautarea si sortarea sunt doua dintre cele mai des întâlnite subprobleme în programare. Ele constituie o parte esentiala din numeroasele procese de prelucrare a datelor. Operatiile de cautare si sortare sunt executate frecvent de catre oameni în viata de zi cu zi, ca de exemplu cautarea unui cuvânt în dictionar sau cautarea unui numar în cartea de telefon.

Cautarea este mult simplificata daca datele în care efectuam aceasta operatie sunt sortate (ordonate, aranjate) într-o anumita ordine (cuvintele în ordine alfabetica, numerele în ordine crescatoare sau descrescatoare).

Sortarea datelor consta în rearanjarea colectiei de date astfel încât un câmp al elementelor colectiei sa respecte o anumita ordine. De exemplu în cartea de telefon fiecare element (abonat) are un câmp de nume, unul de adresa si unul pentru numarul de telefon. Colectia aceasta respecta ordinea alfabetica dupa câmpul de nume.

Daca datele pe care dorim sa le ordonam, adica sa le sortam, sunt în memoria interna, atunci procesul de rearanjare a colectiei îl vom numi sortare interna, iar daca datele se afla într-un fisier (colectie de date de acelasi fel aflate pe suport extern), atunci procesul îl vom numi sortare externa.

Fiecare element al colectiei de date se numeste articol iar acesta la rândul sau este compus din unul sau mai multe componente. O cheie C este asociata fiecarui articol si este de obicei unul dintre componente. Spunem ca o colectie de n articole este ordonat crescator dupa cheia C daca C(I)<=C(j) pentru 1<=i<j<=n, iar daca C(i)>=C(j) atunci sirul este ordonat descrescator.

Algoritmi de cautare
Deci problema cautarii are urmatoarea specificare:

Date a,n,(k,i=1,n);
Preconditia: n din=N, n<=1 si $k_1<k_2<\ldots<k_n$*;*
Rezultate p;

Preconditia: (p=1 si a<=k_1*)sau(p=n+1 si a>*k_n*)*

sau (1<p<=n) si (k_{p-1}<a<=k_p).

Pentru rezolvarea acestei probleme vom descrie mai multi subalgoritmi.

O prima metoda este cautarea secventiala, în care sunt examinate succesiv toate cheile.

Sublagoritmul CautSecv(a,n,k,p) este:
{n din N, n>=1 si k_1<k_2 <…<k_n. Se cauta p astfel ca: (p=1) si a<=k_1) sau (p=n+1 si a>k_n) sau (1<p<=n) si (k_{p-1}<a<=k_p). ?Cazul "înca negasit"}?
Fie p:=0;
Daca a<=k_1 atunci p:=1 altfel
Daca a>k_n atunci p:=n+1 altfel
Pentru i:=2, n executa
Daca (p=0) si (a<=k_1) atunci
p:=i
sfdaca
Sfpentru
Sfdaca
Sfdaca
Sf-*CautSecv*

Se observa ca prin aceasta metoda se vor executa în cel mai nefavorabil caz n-1 comparari, întrucât contorul i va lua toate valorile de la 2 la n. Cele n chei împart axa reala în n+1 intervale. Tot atâtea comparari se vor efectua în n-1 din cele n+1 intervale în care se poate afla cheia cautata, deci complexitatea medie are acelasi ordin de marime ca si complexitatea în cel mai rau caz.

2.2. Cautare binara

Se dau un vector A=(a_1,…,a_n) si o valoare x. Se cere sa se determine daca x se afla printre elementele vectorului A.

Daca A este un vector oarecare, atunci trebuie parcurse secvential elementele vectorului. Aceasta modalitate necesita cel mult n comparatii în cazul unei cautari cu succes si exact n comparatii în cazul cautarii fara succes. Numarul mediu de comparatii în cazul unei cautari cu succes este (1+2+…+n)/2=(n+1)/2.

Daca A are elementele în ordine crescatoare – situatie des întâlnita în practica – atunci exista posibilitatea de a efectua o cautare mai performanta. Deci în continuare vom lucra în ipoteza a_1<=a_2<=…<=a_n.

Vom folosi metoda "Divide et impera", începând prin a compara x cu elementul "din mijloc", adica cu a_i unde i=(n+1)/2. Daca a_i=x, atunci cautarea se încheie cu succes. În caz contrar, vom cauta x în secventa a_1,…,a_{i-1} daca x<a_i sau

în secventa $a_{i+1},\ldots,a_n$ daca x> a. Vom presupune ca dorim ca solutia sa fie exprimata sub forma:

(s,i)=(1,i) daca x=a_i, i =1,n?
(0, i) daca $a_i < x < a_i$, i =1,n+1?unde a_0 = - maxint?? a_{n+1} = +maxint

Cautarea binara se poate realiza practic prin apelul functiei BinarySearch(a,n,k,1,n), deschisa mai jos, folosita în subalgoritmul dat în continuare.

Subalgoritmul CautBin(a,n,k,p) este:

{n din N, n>=1 si $k_1<k_2<\ldots<k_n$. Se cauta p astfel ca: (p=1) si a<=k_1) sau (p=n+1 si a>k_n) sau (1<p<=n) si (k_{p-1}<a<=k_p).}

```
Daca a<=k 1 atunci p:=1 altfel
   Daca a>k n atunci p:=n+1 altfel
         p:=BinarySearch(a,n,k,1,n)
   sfdaca
sfdaca
sfCautBin
Functia BinarySearch(a,n,k,St,Dr) este:
Daca st>dr-1 atunci BinarySearch:=Dr
Altfel m:=(st+dr) div 2;
   Daca a<=k[m]
         Atunci BinarySearch:=BinarySearch(a,n,k,st,m)
         Altfel BinarySearch:=BinarySearch(a,n,k,m,dr)
   Sfdaca
Sfdaca
SfBinarySearch
```

În functia BinarySearch deschisa mai sus, variabilele St si Dr reprezinta capetele intervalului de cautare, iar m reprezinta mijlocul acestui interval. Prin aceasta metoda, într-o colectie având n elemente, rezultatul cautarii se poate furniza dupa cel mult $\log_2 n$ comparari. Deci complexitatea în cel mai rau caz este direct proportionala cu $\log_2 n$. Fara a insista asupra demonstratiei, mentionam ca ordinul de marime al complexitatii medii este acelasi.

Exemplu:

Urmatorul program pascal realizeaza cautarea binara într-un sir de numere întregi, folosind tehnica recursiva:

```
program cautare_binara;  {Solutia Probleme/81Metode/Divide/binara.pas}
type sir=array[1..10] of integer;
var a:sir;
   i,n:byte;
```

```
    x:integer;
procedure   cautare(st,dr:byte);
var mijloc:byte;
begin
if st>dr then  begin
write('el nu apartine sirului');exit;
end;
mijloc:=(st+dr) div 2;
if a[mijloc]=x then
write('valoarea ',x,' se afla pe pozitia  ',mijloc)
            else
if a[mijloc]>x then
             cautare(st,mijloc-1)
                        else
             cautare(mijloc+1,dr);
end;

begin
write('dati nr de valori ');read(n);
write('dati sirul ordonat');
for i:=1 to n do
begin
write('a[',i,']= ');
read(a[i]);
end;
write('dati valoarea ');read(x);
cautare(1,n);
readln;
readln;
end.
```

2.3. Sortarea rapida

O metoda mai performanta de ordonare, care va fi prezentata în continuare, se numeste “Quick Sort” si se bazeaza pe tehnica “divide et impera” dupa cum se poate observa în continuare. Metoda este prezentata sub forma unei proceduri care realizeaza ordonarea unui subsir precizat prin limita inferioara si limita superioara a indicilor acestuia. Apelul procedurii pentru ordonarea întregului sir este: Quick Sort(n,k,1,n), unde n reprezinta numarul de articole ale colectiei date. Deci

Subalgoritmul SortareRapida (n,k) este:

Cheama Quick Sort (n,k,1,n)

SfSortareRapida

Procedura Quick Sort (n,k,St,Dr) va realiza ordonarea subsirului k_{St} , k_{St+1},…, k_{Dr}. Acest subsir va fi rearanjat astfel încât k_{St} sa ocupe pozitia lui finala (când sirul este ordonat). Daca i este aceasta pozitie, sirul va fi rearanjat astfel încât urmatoarea conditie sa fie îndeplinita:

$k_j <= k_j <= k_l$, pentru st <= j < i < l <= dr

Odata realizat acest lucru, în continuare va trebui doar sa ordonam subsirul

k_{St} , k_{St+1} , … , k_{i-1} prin apelul recursiv al procedurii Quick Sort (n,k,st,i-1) si apoi subsirul k_{i+1} , … , k_{Dr} prin apelul Quick Sort (i+1, Dr). Desigur ordonarea acestor doua subsiruri (prin apelul recursiv al procedurii) mai este necesara doar daca acestea contin cel putin doua elemente.

Procedura Quick Sort este prezentata în continuare:

Subalgoritmul Quick Sort (n,k,St,Dr) este:
Fie i:=St; j:=Dr; a:=k;
Repeta
Câttimp k_j >=a si (i<j) executa j:=j-1 sfcât
K_i:= k_j;
Câttimp k_i >=a si (i<j) executa i:=I+1 sfcât
K_j:= k_i;
Pânacând i=j sfrep
Fie k_j:=a;
Daca St<i-1 atunci Cheama Quick Sort (n,k,St,i-1)
Sfdaca
Daca i+1<Dr atunci Cheama Quick Sort (n,k,i+1,Dr)
Sfdaca
Sf Quick Sort

Complexitatea algoritmului prezentat este $O(n^2)$ în cel mai nefavorabil caz, dar complexitatea medie este de ordinul $O(n\log_2 n)$.

Algoritmul QuickSort este disponibil în limbajul Pascal si poate fi utilizat pentru cautari rapide:

```
program QSort;
{$R-,S-}
uses Crt;
{ Acest program aplica metoda Quicksort pentru un sir de 1000 de}
{ numere alese aleator, cu valori intre 0 si 29999. Rezultatul  }
{este afisat direct la ecran pentru a observa viteza de executie}
const
  Max = 1000;
type
  List = array[1..Max] of Integer;
```

```
var
  Data: List;
  I: Integer;
procedure QuickSort(var A: List; Lo, Hi: Integer);
{Procedura QuickSort contine o procedura recursiva }
{care ordoneaza efectiv elemnetele.                }
procedure Sort(l, r: Integer);
var
  i, j, x, y: integer;
begin
  i := l; j := r; x := a[(l+r) DIV 2];
  repeat
    while a[i] < x do i := i + 1;
    while x < a[j] do j := j - 1;
    if i <= j then
    begin
      y := a[i]; a[i] := a[j]; a[j] := y;
      i := i + 1; j := j - 1;
    end;
  until i > j;
  if l < j then Sort(l, j);
  if i < r then Sort(i, r);
end;

begin {QuickSort};
  Sort(Lo,Hi);
end;

begin {QSort}
  Write('Generare 1000 numere ...');
  Randomize;
  for i := 1 to Max do Data[i] := Random(30000);
  Writeln;
  Write('Sortarea numerelor ...');
  QuickSort(Data, 1, Max);
  Writeln;
  for i := 1 to 1000 do Write(Data[i]:8);
end.
```

2.4. Sortare cu numar minim de comparatii

Fie de sortat vectorul $A=(a_1,\ldots,a_n)$ având componentele distincte.

Oricarei strategii de sortare bazata pe comparari ale elementelor lui A i se poate asocia un arbore binar strict în care:

- ✍ fiecare vârf intern (neterminal) este etichetat cu i:j semnificând faptul ca se compara elementele a_i si a_j ;
- ✍ muchia spre descendentul stâng reprezinta ramificarea corespunzatoare $a_i < a_j$,

- muchia spre descendentul drept corespunde situatiei $a_i > a_j$.
- pentru orice vârf terminal, comparatiile reprezentate de vârfuri împreuna cu rezultatele lor, corespunzatoare faptului ca fiecare vârf diferit de radacina este descendentul stâng sau drept al tatalui sau, conduc în mod necesar la inegalitatile:

$$a_{p(1)}<a_{p(2)}<\ldots<a_{p(n)}$$

unde p din S_n este permutarea atasata vârfului extern respectiv.

Un astfel de arbore se numeste arbore de comparare. Aceasta problema va fi discutata în capitolul special destinat grafurilor.

Probleme rezolvate si probleme propuse

1**.** Sa se determine, folosind tehnica Divide Et Impera, maximul (minimul dintre elementele unui sir. (Probleme/81Metode/Divide/maxdivi.pas(mindivi.pas)).
2. Calculati produsul elementelor unui sir folosind Divide Et Impera. (Probleme/81Metode/Divide/prodivi.pas).
3. Calculati cel mai mare divizor comun a **n** numere naturale nenule.
4. Problema turnurilor din Hanoi. Pe o tija sunt asezate n discuri in ordinea descrescatoare a diametrelor lor. Se cere sa se mute toate discurile pe o alta tija, folosind o a treia tija auxiliara, exact în aceeasi ordine. Singura mutare permisa este aceea de a aseza un disc de pe o tija pe alta, respectându-se întotdeauna ordinea descrescatoare a diametrelor discurilor aflate pe fiecare tija. (Probleme/81Metode/ Divide/hanoi.pas).

3. Metoda Backtracking

Vom începe prezentarea metodei Backtracking printr-un exemplu.

Se considera tabla de sah si opt turnuri. Se cere sa se aseze în toate modurile posibile cele opt turnuri pe tabla de sah astfel încât sa nu existe doua care se ataca (nu se afla pe accesi linie respectiv coloana).

Daca luam în fata noastra o tabla de sah si încercam sa asezam turnuri din coltul din stânga sus al tablei vom observa ca o prima asezare, corecta, situeaza turnurile pe diagonala tablei. Fie aceasta solutie 1,2,3,4,5,6,7,8, considerând linia pe care am asezat fiecare turn, coloana fiind pozitia lor în sir. Dorim însa sa obtinem toate posibilitatile fara a pierde nici una. Sa procedam în felul urmator:

Luati de pe tabla ultima piesa, turnul 8.

Încercati sa asezati turnul 7 cu o linie mai jos, pe linia 8 pe tabla si apoi încercati sa reasezati turnul 8. Veti obtine solutia 1,2,3,4,5,6,8,7.

Reluati acelasi algoritm de unde am ramas. Luati turnul 8.

Turnul 7 nu poate fi mutat mai jos, deci luati turnul 7.

Încercati sa asezati turnul 6 cu o linie mai jos, pe linia 7 pe tabla si apoi încercati sa reasezati turnul 7 pe prim pozitie corecta, 6, apoi la fel pentru turnul 8. Veti obtine solutia 1,2,3,4,5,7,6,8.

Reluati acelasi algoritm de unde am ramas. Luati turnul 8. Turnul 7 poate fi mutat mai jos iar asezarea lui 8 va da solutia 1,2,3,4,5,7,8,6.

Continuati în acelasi mod si veti ajunge în final la ultima solutie, asezarea pe diagonala secundara, respectiv 8,7,6,5,4,3,2,1.

Daca ati mânuit corect piesele pe tabla de sah înseamna ca ati înteles metoda Backtracking, cauta cu revenire.

Descrierea metodei.

Metoda Backtracking se aplica problemelor ale caror solutii sunt elemente ale unui produs cartezian A_1xA_2 x…xA_n, care satisfac anumite conditii, numite conditii interne.

Multimea tuturor elementelor produsului cartezian se numeste spatiul solutiilor. Din aceasta multime se determina toate solutiile de forma $(x_1,x_2,\ldots,x_n)$ din A_1xA_2 x…xA_n care satisfac conditiile interne.

Metoda backtracking se foloseste în rezolvarea problemelor care îndeplinesc simultan urmatoarele conditii:

-solutia lor poate fi pusa sub forma unui vector x= $(x_1,x_2,\ldots,x_n)$ dintr-un spatiu A= A_1xA_2 x…xA_n , astfel încât x_1? A_1, x_2 ? A_2 ,…,x_n ? A_n .

-multimile A_i, i=1…n, sunt multimi finite, card $(A_i) = a_i$, iar elementele lor se considera ca se afla într-o relatie de ordine bine stabilita;

-exista anumite relatii între componentele $x_1,x_2,\ldots,x_n$ ale vectorului X, numite *conditii interne*;

-nu se dispune de o alta metoda de rezolvare mai rapida.

Multimea A= A_1xA_2 x…xA_n se numeste *spatiul solutiilor posibile*. Solutiile posibile care satisfac conditiile interne se numesc *solutiile problemei*.

Observatii

-Nu pentru toate problemele n este cunoscut;

-$x_1,x_2,\ldots,x_n$ pot fi la rândul lor vectori;

-în multe probleme multimile A_1xA_2 x…xA_n coincid.

Scopul acestei metode este de a determina toate *solutiile problemei* (cu scopul de a le lista sau de a alege dintre ele una care maximizeaza sau minimizeaza o eventuala functie obiectiv data). O metoda de determinare a solutiilor consta în a genera într-un mod oarecare toate solutiile posibile si de a verifica daca ele satisfac conditiile interne.

Dezavantajul consta în faptul ca timpul cerut de aceasta investigare exhaustiva este foarte mare (chiar pentru card(A_i)=2, timpul necesar este de ordinul 2^n, adica exponential).

Metoda backtracking urmareste sa evite generarea tuturor solutiilor posibile. În acest scop, elementele vectorului x primesc pe rând valori, în sensul ca lui x_k i se atribuie o valoare numai daca au fost atribuite deja valori lui $x_1,x_2,\ldots,x_{k-1}$. O data o valoare pentru x_k stabilita, nu se trece direct la atribuirea de valori lui x_{k+1}, ci se verifica conditiile de continuare referitoare la $x_1,x_2,\ldots,x_k$, care stabilesc situatiile în care are sens sa trecem la calculul lui x_{k+1}.

Concret:

-se alege primul element x_1, ce apartine lui A_1

-presupunând generate elementele $x_1,x_2,\ldots,x_k$ apartinând multimilor A_1xA_2 $x\ldots xA_k$ se alege (daca exista) x_{k+1}, primul element disponibil din multimea A_{k+1}; apar doua posibilitati:

i. nu s-a gasit un astfel de element, caz în care se reia cautarea considerând generate elementele $x_1,x_2,\ldots,x_{k-1}$, iar aceasta se reia de la urmatorul element al multimii A_k, ramas netestat.

ii. a fost gasit, caz în care se testeaza daca acesta îndeplineste anumite conditii de continuare, aparând astfel 2 posibilitati:

a) le îndeplineste, caz în care se testeaza daca s-a ajuns la solutie si apar, din nou, 2 posibilitati:

- s-a ajuns la solutie, se tipareste solutia si se reia algoritmul considerând generate elementele $x_1,x_2,\ldots,x_k$ (se cauta, în continuare, un element al multimii A_{k+1} ramas netestat).

- nu s-a ajuns la solutie, caz în care se reia algoritmul considerând generate elementele $x_1,x_2,\ldots,x_{k+1}$ si se cauta un prim element x_{k+2}? A_{k+2}.

b) nu le îndeplineste, caz în care se reia algoritmul considerând generate elementele $x_1,x_2,\ldots,x_k$, iar elementul x_{k+1} se cauta între elementele multimii A_{k+1}ramase netestate.

Algoritmul se termina atunci când nu mai exista nici un element x_1? A_1, netestat.

Observatie. Tehnica backtracking are ca rezultat generarea tuturor solutiilor problemei. În cazul în care se cere o singura solutie, se poate forta oprirea, atunci când a fost gasita.

Am aratat ca orice solutie se genereaza sub forma de vector. Vom considera ca generarea solutiilor se face într-o stiva. În acest fel, stiva (notata st) va arata ca mai jos:

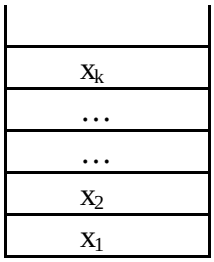


Fig. 2.1.

Nivelul k+1 al stivei trebuie initializat (pentru a alege în ordine elementele multimii k+1). Initializarea trebuie facuta cu o valoare aflata (în relatia de ordine considerata pentru multimea A_{k+1}) înaintea tuturor valorilor posibile din multime.

De exemplu, pentru generarea permutarilor multimii ?1, 2, …, n?, orice nivel al stivei va lua valori de la 1 la n. Initializarea unui nivel (oarecare) se face cu valoarea 0. Procedura de initializare o vom numi init si va avea doi parametri: k, nivelul care trebuie initializat si st (stiva).

Gasirea urmatorului element al multimii A_k, element netestat, se face cu ajutorul procedurii succesor (as, st, k). Parametrul as ?am succesor? este o variabila booleana. În situatia în care am gasit elementul, acesta este pus în stiva si as ia valoarea true, contrar (nu a ramas un element netestat) as ia valoarea false.

Odata ales un element, trebuie vazut daca acesta îndeplineste conditiile de continuare (altfel spus, daca elementul este valid). Acest test se face cu ajutorul procedurii valid (ev, st, k). Variabila ev este booleana.

Testul daca s-a ajuns sau nu la solitia finala se face cu ajutorul functiei solutie (k). Solutia se tipareste cu ajutorul procedurii tipar.

Prezentam rutina backtracking:

```
k:=1; init(1,st);
while k?0 do
        begin
          repeat
            succesor (as, st, k);
              if as then valid (ev, st, k)
           until (not as) or (as and ev);
              if as then
                if solutie (k)
                   then tipar
                 else
                    begin
                       k:=k+1;
```

```
                    init (k, st)
                  end
              else k:=k-1
end;
```

Observatii

Problemele rezolvate prin aceasta metoda necesita timp îndelungat. Din acest motiv, este bine sa utilizam metoda numai atunci când nu avem la dispozitie un alt algoritm mai eficient.

Mentionam ca exista probleme pentru care nu se cunosc algoritmi efecienti de rezolvare, deci backtracking este indicata.

Rutina prezentata corespunde variantei iterative.

Algoritmul de descriere a metodei Backtracking care foloseste tehnica recursiva este prezentat în algoritmul BACK, unde A este multimea A_1xA_2 x…xA_n data ca un sir de siruri, n este numarul de multimi, x sirul solutiei x= (x_1,x_2,…,x_n). Folosim urmatorii sublalgoritmi:

Algoritmul **Posibil (x, A_k(i), cod)** verifica daca elemetul **A_k(i),** elementul i din multimea A_k , poate fi adaugat în solutie, deci daca verifica conditiile interne.

Algoritmul Solutie (x, n) care furnizeaza o solutie corecta a problemei;

Vom utiliza aceeasi variabila k pentru a memora stiva de lucru curenta, respectiv pozitia din sirul x pentru care cautam solutie.

```
Algoritm Back (k: numeric);

Pentru i:=1 , card (Ak)  executa
        x (k) = Ak(i);
        Posibil ( x, Ak(i), cod );
        Daca cod atunci
                Daca k < n atunci
                        Back (k + 1)
                     altfel
                        Scrie_sir ( x, n);
                Sf daca;
        Sf daca;
Sf pentru;
Stop;
```

Exemple

1. Sa se genereze toate permutarile multimii M = {1, 2, 3, …, n}, pentru un n dat. (probcurs\capIV\permutr.pas, permner.pas).

permutr.pas – varianta recursiva

```
type sir=array[1..10] of byte;
var x:sir;
   n,m:byte;

procedure solutie(x:sir;n:byte);
var i:byte;
begin
for i:=1 to n do
write(x[i],',');
writeln;
end;

Function valid(i:byte):boolean;
var cod:boolean;
   j:byte;
begin
cod:=true;
for j:=1 to i-1 do
   if x[i]=x[j] then cod:=false;
valid:=cod;
end;

procedure permut(j:byte);
var  i:byte;

begin
for i:=1 to n do
begin
  x[j]:=i;
  if valid(j) then
    if j<n then permut(j+1)
        else solutie(x,n);

end;
end;

begin
Write('n');readln(n);
permut(1);
end.
```

permner.pas – varianta nerecursiva

```
program bktr_nerecursiv;
uses crt;
type vector=array[1..25] of integer;
var st, v:vector;
   n:integer;

procedure initializari;
var i:integer;
begin
    write('n='); readln(n);
    for i:=1 to 25 do st[i]:=0;
    writeln;
end;

procedure tipar(p:integer);
var i:integer;
begin
    for i:=1 to p do write(st[i]:4,' ');
    writeln;
end;

function valid(p:integer):boolean;
var i:integer; ok:boolean;
begin
    ok:=true;
    for i:=1 to p-1 do
       if st[p]=st[i] then ok:=false;
    valid:=ok;
end;

procedure back;
{implementeaza algoritmul nerecursiv de backtracking}
var p:integer;                                  {varful stivei}
begin
    p:=1; st[p]:=0;                      {initializam primul nivel}
    while p>0 do                {cat timp stiva nu devine din nou vida}
     begin
       if st[p]<n then
         begin
            st[p]:=st[p]+1; {punem pe nivelul p urmatoarea valoare}
            if valid(p) then
                        {daca solutia(st[1],st[2],...,st[p]) este valida}
              if p=n then                {daca solutia este si finala}
```

```
                    tipar(p)
                   else
                    begin
                     p:=p+1; st[p]:=0;
        {trecem la nivelul urmator, pe care il reinitializam}
                    end;
              end
                else
             p:=p-1;          {pasul inapoi la nivelul anterior}
          end;
        end;

        begin
           clrscr;
           initializari;
           back;
           readln;
        end.
```

2. Backtracking în plan.

Pe o suprafata dreptunghiulara de dimensiune m x n, împartita în patratele de dimensiune unu, se afla depozitate rezervoare de combustibil. Un autoturism porneste din unul din patratele si se deplaseaza doar pe orizontala si verticala, consumând o unitate de combustibil la trecerea de la un patratel la altul, pâna când aduna toate rezervoarele de benzina. Sa se determine un punct de plecare si de sosire astfel încât cantitatea acumulata sa fie maxima.

Autoturismul nu se poate deplasa fara combustibil si aduna toata cantitatea la trecerea printr-un patratel.

(probcurs\capIV\auto.pas)

```
program auto;
uses crt;
type    pereche=record
                  x,y:byte;
             end;
       mat=array[0..20,0..20]of integer;
       sir=array[1..40] of pereche;
var f:text;
   a,b:mat;
   c,cmax:sir;
   Sn,Snmax,n,i,j,m,kmax:integer;

Procedure drum(m,n:integer;c:sir);
var l:integer;
```

```
begin

for l:=1 to kmax do
      writeln(c[l].x,',',c[l].y);
end;

Function oprire:boolean;
var i,j:integer;
begin
oprire:=true;
for i:=1 to m do
   for j:=1 to n do
      if b[i,j]>0 then oprire:=false;
end;

Procedure mers(k,i,j:integer);
var l:integer;
begin
if b[i,j]<> -1 then
begin
sn:=sn+a[i,j]-1;
b[i,j]:=-1;
c[k].x:=i;
c[k].y:=j;
if oprire then
       begin
          if SN>Snmax then
            begin
            Snmax:=Sn;
            cmax:=c;
            kmax:=k;
            end;
       end
            else
       begin
       if (SN>0) and (i>1) then mers(k+1,i-1,J);
       if (SN>0) and (j<n) then mers(k+1,i,J+1);
       if (SN>0) and (i<m) then mers(k+1,i+1,J);
       if (SN>0) and (j>1) then mers(k+1,i,J-1);
       end;
sn:=sn-a[i,j]+1;
b[i,j]:=a[i,j];
end;
end;
```

```
begin
assign(f,'mat.in');
reset(f);
read(f,m,n);
for i:=1 to m do
   for j:=1 to n do
      read(f,a[i,j]);
SNmax:=0;
b:=a;
for i:=1 to m do
   for j:=1 to n do
      if a[i,j]>0 then
            mers(1,i,j);
writeln('Cantitate maxima adunata:', SNmax);
drum(m,n,cmax);
close(f);
end.
```

Probleme

P.1. Sa se genereze toate aranjamentele de n luate câte m ale multimii M = {1, 2, 3, …, n}, pentru n si m date. (probcurs\capIV\p1.pas)

P.2. Sa se genereze toate combinarile de n luate câte m ale multimii M = {1, 2, 3, …, n}, pentru n si m date. (probcurs\capIV\p2.pas)

P.3. Într-o biblioteca sunt n titluri de carti. Un raft al bibliotecii poate contine doar m carti, m<n. Descrieti toate modurile de asezare a celor n carti pe raftul de câte m astfel încât titlurile sa fie în ordine alfabetica. (probcurs\capIV\p3.pas)

P.4. Un grup de n persoane contine f femei. Sa se formeze toate delegatiile posibile formate din m persoane care contin exact p femei. (m<=n, p <=f) (probcurs\capIV\p4.pas)

4. Programare dinamica

Metodele studiate pâna acum cautau solutii într-un spatiu de solutii posibile si alegeau, în diverse moduri, multimi ale acestora pe baza unor conditii date. Daca metoda Backtracking are o complexitate exponentiala iar Greedy si Divide et Impera erau polinomiale, rezolvarea unor probleme se poate realiza polinomial daca se cere valoarea optima a solutiei.

Programarea dinamica face parte din categoria metodelor matematice, bazate pe reguli de calcul si deductii, numite în literatura de specialitate metode de

scufundare. Metoda se apica problemelor de optim si consta în determinarea solutiei pe baza unui sir de decizii. Strategia generala de rezolvare a problemelor folosind metode programarii dinamice se bazeaza pe principiul optimalitatii (Bellman R.) si se bazeaza pe descompunerea problemei în subprobleme. Spre deosebire de metoda Divide et Impera în care subproblemele sunt independente, subproblemele rezolvate prin programarea dinamica depind unele de altele unele subprobleme folosind solutiile subproblemelor anterioare. Matoda propgramarii dinamice actioneaza de jos în sus trecînd rezolvarea problemei prin stari succesive. Metoda se caracterizeaza prin:

- evitarea calcularii de mai multe ori a aceleiasi subprobleme si pastrarea rezultatelor intermediare.
- se porneste de la cele mai mici subprobleme si pe baza unor formule recursive se construiesc alte subprobleme.
-respecta principiul optimalitatii: *Oricare ar fi o stare initiala si decizia initiala, deciziile ramase trebuie sa contituie o strategie optima în raport cu starea care rezulta din decizia anterioara.*

Solutia optima nu este în general unica, la fel ca în cazul medodei greedy.

Modelul general al metodei programarii dinamice

Fie D_1 , D_2 , ... , D_n un sir de decizii care transforma problema din starea S_{i-1} în starea S_i , i=0,n. Trecerea de la o stare la alta se face de regula prin aplicarea unei recurente în deciziile care se iau si apoi calculul pe baza unor formule recursive.

De obicei, relatiile de recurenta sunt de doua feluri:

- înainte

decizia D_i depinde de deciziile D_{i+1}, ... , D_n deci, deciziile se iau începând de la n la 1.

- înapoi

decizia D_i depinde de deciziile D_1, ... , D_{i-1} deci, deciziile se iau începând de la 1 la n.

Pasii pe care trebuie sa-i parcurgem pentru a rezolva o problema folosind programarea dinamica se pot descrie prin urmatoarea secventa:

- Defineste structura unei solutii optime
- Cauta o formula recursiva care permite calculul valorii solutiei optime într-o anumita stare.
- Calculeaza valoarea solutiei optime „de jos în sus”
- Daca pe lânga valoarea optima se doreste si o solutie explicita parcurge de sus în jos sirul deciziilor si contruieste solutia.

Problema

Un robot care functioneaza pe o platforma de forma dreptunghiulara (mxn) se afla în coltul din stânga sus si prin deplasare numai spre dreapta si jos aduna piese aflate pe platforma si ajunge în coltul din stânga jos. Daca se dau piesele aflate la intersectia liniilor orizontale si verticale ale platformei si robotul se poate deplasa doar pe aceste linii se cere numarul maxim de piese pa care robotul le poate aduna la o trecere.

Exemplu:

M=6 si N=7, valorile reprezinta numarul de piese, deplasarea se face spre dreapta si jos din fiecare punct.

1 0 3 0 0 2 0
2 0 2 0 3 0 1
0 2 3 0 4 5 6
0 0 0 0 0 0 0
0 9 0 8 0 7 0
4 0 5 0 6 7 8

Numarul maxim de piese adunate este 42.

Problema se poate rezolva folosind metode Backtracking dar eficienta este foarte mica la matrice de dimensiuni mari.

Vom încerca o gândire inductiva pentru a definii structura problemei. Sa presupunem ca matricea are dimensiunea 2. Atunci maximul pieselor se obtine în trei etape. În pozitia (1,2) voi aduna 1 piesa, în (2,1) 3 piese, deci în (2,2) maximul posibil va fi 3. Aceasta valoarea este maximul dintre numarul de piese adunate în pozitiile imediat anterioare pozitiei în care se afla, si din care poate ajunge în pozitia curenta. Vom extinde la m=2 si n=3 problema si vom obtine succesiv

1 1 1

1 1 3 deci maxim 3. Unde fiecare element este maximul dintre elementele aflate pe pozitiile (i-1,j), (i,j-1) la care adaugam valoarea curenta.

În final se obtine:

1 1 1 1 1 1 1
1 1 3 3 6 6 7
1 3 6 6 10 15 21
1 3 6 6 10 15 21
1 12 12 20 20 27 27
1 12 17 20 26 34 42 unde 42 reprezinta maximul cautat.

Sa definim formula de recurenta:

Fie A(i,j), i=1,m, j=1,n, matricea pieselor, iar B(i,j) matricea starilor. Atunci urmatoarea formula de recurenta defineste problema noastra.

B(i,j)= max(B(i-1,j),B(i,j-1))+ A(i,j),
unde i=1,m, j=1,n,
si vom considera B(i,0)=B(0,j)=0.

Calculâd succesiv, pe linii, valorile maxime folosind relatia de recurenta, elementul de pe pozitia (m,n) ne da valoarea maxima cautata.

Încercând sa gasim si un drum pe care sa-l parcurga robotul vom pleca din (m,n) si vom alege, în sens invers pozitia (i-1,j) sau (i,j-1) daca si numai daca diferenta între (B(i,j)-B(i-1,j))=A(i,j) respectiv B(i,j),B(i,j-1))=A(i,j). Deci un drum invers este 42, 34, 27, 20, 12, 12, 3, 3, 1, 1.

Solutia problemei este data în Probleme\41Metode\Dinamica\ robot.pas.

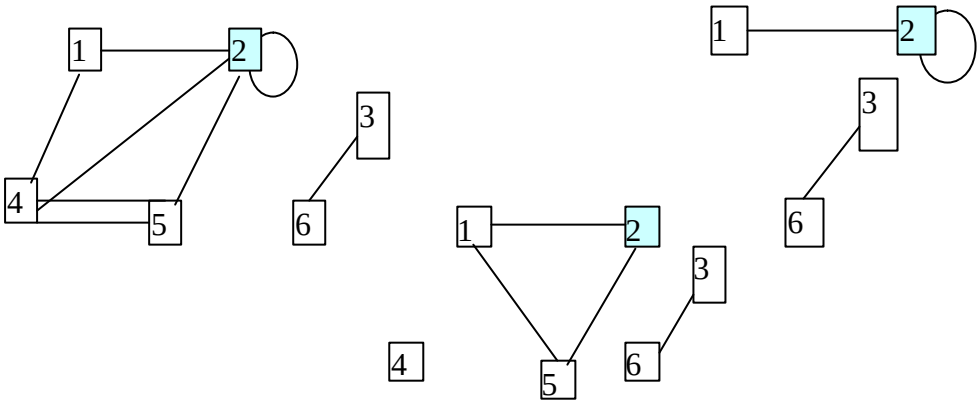
V. Grafuri. Arbori. Reprezentare si algoritmi elementari.

5.1. Grafuri

Definitie

Un **graf orientat (sau digraf)** G este o pereche (V, E), unde V este o multime finita, iar E este o relatie binara pe V, definita prin **perechi ordonate** (x, y).

Multimea V se numeste **multimea vârfurilor** lui G, iar elementele ei se numesc **vârfuri**. Multimea E se numeste **multimea arcelor** lui G, si elementele ei se numesc **arce**.



(a) (b) (c)

Figura 5.1. Grafuri orientate si neorientate

(a) Un graf orientat G = (V, E), unde V = {1, 2, 3, 4, 5, 6}si E = {(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)}. Arcul (2, 2) este o autobucla.

(b) Un graf neorientat G = (V, E), unde V = {1,2,3,4,5,6} si E = {(1, 2),(1, 5),(2, 5),(3, 6)}. Vârful 4 este izolat.

(c) Subgraful grafului de la (a) indus de multimea de vârfuri {1, 2, 3, 6}.

Figura 1 (a) este o reprezentare grafica a unui graf orientat cu multimea de vârfuri {1, 2, 3, 4, 5, 6}. În figura, vârfurile sunt reprezentate prin cercuri, iar arcele prin sageti. Observati ca sunt posibile autobuclele - arce de la un vârf la el însusi.

Definitie

Într-un **graf neorientat** G =(V, E), **multimea muchiilor** E este construita din perechi de vârfuri neordonate, si nu din perechi ordonate. Cu alte cuvinte, **o muchie** este o multime {u, v}, unde u, v apartin lui V si u diferit de v. Prin

conventie, pentru o muchie vom folosi notatia (u, v) în locul notatiei pentru multimi {u, v}, iar (u, v) si (v, u) sunt considerate a fi aceeasi muchie. Într-un graf neorientat, autobuclele sunt interzise si astfel fiecare muchie este formata din exact doua vârfuri distincte.

Figura 1 (b) este o reprezentare grafica a unui graf neorientat având multimea de vârfuri V={1, 2, 3, 4, 5, 6}

Multe definitii pentru grafuri orientate si neorientate sunt aceleasi, desi anumiti termeni pot avea semnificatii diferite în cele doua contexte.

Definitie

Daca (u, v) este un arc într-un graf orientat G = (V, E), spunem ca (u, v) este **incident din** sau **pleaca din vârful u** si este **incident în** sau **intra în vârful** v.

De exemplu, arcele care pleaca din vârful 2, în figura 1, sunt (2, 2), (2, 4) si (2, 5). Arcele care intra în vârful 2 sunt (1, 2) si (2, 2).

Definitie

Daca (u, v) este o muchie într-un graf neorientat G = (V, E), spunem ca (u, v) este **incidenta** vârfurilor u si v.

În figura 1 (b), muchiile incidente ale vârfului 2 sunt (1, 2) si (2, 5).

Definitie

Daca (u, v) este o muchie (arc) într-un graf G = (V, E), spunem ca vârful v este **adiacent vârfului** u. Atunci când graful este neorientat, **relatia de adiacenta este simetrica**. Când graful este orientat, relatia de adiacenta nu este neaparat simetrica. Daca v este adiacent vârfului u într-un graf orientat, uneori scriem

u ? ?**v**.

În partile (a) si (b) ale figurii 1, vârful 2 este adiacent vârfului 1, deoarece muchia (arcul) (1, 2) apartine ambelor grafuri. Vârful 1 nu este adiacent vârfului 2 în figura 1 (a), deoarece muchia (2, 1) nu apartine grafului.

Definitie

Gradul unui vârf al unui graf neorientat este numarul muchiilor incidente acestuia.

De exemplu, vârful 2, din figura 1 (b), are gradul 2. Un vârf al carui grad este 0, cum este, de exemplu, vârful 4, din figura 1 (b), se numeste vârf izolat.

Într-un graf orientat, **gradul exterior** al unui vârf este numarul arcelor ce pleaca din el, iar **gradul interior** al unui vârf este numarul arcelor ce intra în el. Gradul unui vârf al unui graf orientat este gradul sau interior, plus gradul sau exterior.

Vârful 2 din figura 1 (a) are gradul interior 2, gradul exterior 3 si gradul 5.

Definitie

Un **drum de lungime k de la un vârf *u* la un vârf *z*** într-un graf G = (V, E) este un sir de vârfuri (v0, v1, v2,..., vk) astfel încât u = v0, z = vk si (v(i-1), vi) apartin lui E pentru i=1, 2,..., k. **Lungimea unui drum** este numarul de muchii (arce) din acel drum. Drumul contine vârfurile v0, v1, v2,..., vk si muchiile (v0, v1), (v1, v2),..., (vk-1, vk). Daca exista un drum p de la u la z, spunem ca z este accesibil din u prin p.

Definitie

Un **drum este elementar** daca toate vârfurile din el sunt distincte.

În figura 1 (a), drumul (1, 2, 5, 4) este un drum elementar de lungime 3. Drumul (2, 5, 4, 5) nu este elementar.

Un **subdrum** al unui drum p = (v0, v1,..., vk) este un subsir continuu al vârfurilor sale. Cu alte cuvinte, pentru orice 0 ? i ? j ? k, subsirul de vârfuri (v0, v1, v2,..., vk) este un subdrum al lui p.

Definitie

Într-un graf orientat, un drum (v0, v1,..., vk) formeaza un **ciclu** daca v0 = vk si drumul contine cel putin o muchie. **Ciclul este elementar** daca, pe lânga cele mai de sus, v1, v2,..., vk sunt distincte. O autobucla este un ciclu de lungime 1. Doua drumuri (v0, v1,..., v(k-1), v0) si (w0, w1,... w(k-1), w0) formeaza acelasi ciclu daca exista un numar întreg j astfel încât wi = w(i+j) mod k pentru i = 0, 1,..., k-1.

În figura 1 (a), drumul (1, 2, 4, 1) formeaza acelasi ciclu ca drumurile (2, 4, 1, 2) si (4, 1, 2, 4). Acest ciclu este elementar, dar ciclul (1, 2, 4, 5, 4, 1) nu este elementar. Ciclul (2, 2) format de muchia (2, 2) este o autobucla.

Definitie

Un **graf orientat fara autobucle este elementar**.

Într-un graf neorientat, un drum (v0, v1,..., vk) formeaza formeaza un ciclu elementar daca k ? 3, v0 = vk si vârfurile v1, v2,..., vk sunt distincte.

De exemplu, în figura 1 (b), drumul (1, 2, 5, 1) este un ciclu.

Definitie

Un graf fara cicluri este **aciclic**.

Propozitie

Un graf neorientat este **conex** daca fiecare pereche de vârfuri este conectata printr-un drum. Componentele conexe ale unui graf sunt clasele de echivalenta ale vârfurilor sub relatia "este accesibil din".

Graful din figura 1 (b) are trei componente conexe: {1, 2, 5}, {3, 6} si {4}. Fiecare vârf din {1, 2, 5} este accesibil din fiecare vârf din {1, 2, 5}.

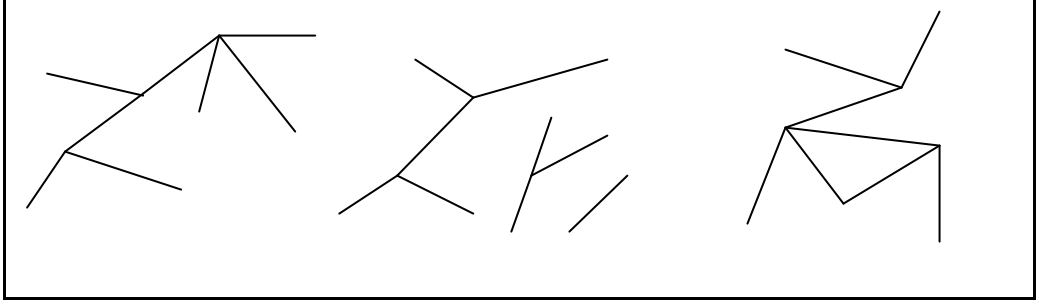
Propozitie

Un **graf neorientat** este **conex** daca are exact o componenta conexa, sau, altfel spus, dupa fiecare vârf este accesibil din fiecare vârf diferit de el.

Un **graf orientat este tare conex** daca fiecare doua vârfuri sunt accesibile din celalalt. Componentele tare conexe ale unui graf sunt clasele de echivalenta ale vârfurilor sub relatia "sunt reciproc accesibile". Un graf orientat este tare conex daca are doar o singura componenta tare conexa.

Graful din figura 5.2. (a) are trei componente tare conexe: {1, 2, 4, 5}, {3} si

Figura 5.2. (a) (b) (c)



{6}. Toate perechile de vârfuri din {1, 2, 4, 5} sunt reciproc accesibile. Vârfurile {3, 6} nu formeaza o componenta tare conexa, deoarece vârful 6 nu este accesibil din vârful 3.

5.2. Arbori

Definitie

Un **arbore** este un graf neorientat, aciclic si conex. Daca un graf neorientat este aciclic, dar s-ar putea sa nu fie conex, el formeaza o **padure**. Multi algoritmi pentru arbori functioneaza, de asemenea, si pentru paduri.

Figura 2 (a) prezinta un arbore liber, iar figura 2 (b) prezinta o padure. Padurea din figura 2 (b) nu este arbore si pentru ca nu este conexa. Graful din figura 2 (c) nu este nici arbore si nici padure, deoarece contine un ciclu. Urmatoarea teorema prezinta într-o forma concentrata multe proprietati importante ale arborilor.

Teorema (Proprietatile arborilor)

Fie G = (V, E) un graf neorientat. Urmatoarele afirmatii sunt adevarate:

1. G este un arbore.
2. Oricare doua vârfuri din G sunt conectate printr-un drum elementar unic.
3. G este conex, dar, daca eliminam o muchie oarecare din E, graful obtinut nu mai este conex.

4. G este conex, si |E| = |V| - 1.

5. G este aciclic, si |E| = |V| - 1.

6. G este aciclic, dar daca adaugam o muchie oarecare în E, graful obtinut contine un ciclu.

Demonstratia acestei teoreme se gaseste în [4] la pag. 79.

Definitie

Un **arbore cu radacina** este un arbore în care unul din vârfuri se deosebeste de celelalte. Acest vârf se numeste **radacina**. Pentru un vârf al arborelui vom folosi uneori denumirea de **nod.**

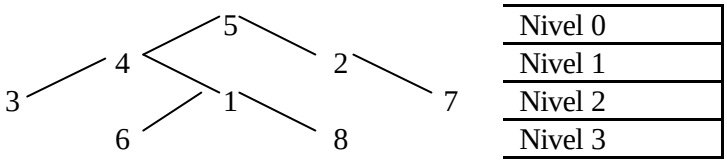


Fig. 5.3. Arbore binar, reprezentat pe nivele.

În exemplul de mai sus nodul 5 este radacina.

Orice nod aflat pe un drum de la radacina spre orice alt nod se numeste **stramos** al radacinii, sau al unui alt nod. Daca se considera un nod x si toti stramosii sai spunem ca avem un **subarbore** de radacina x. De exemplu subarborele lui 4 contine {4, 3, 1, 6, 8}.

Un arbore este definit **pe nivele**, începând cu nivelul 0. O muchie (x, y) în care x este pe un nivel inferior lui y este în relatia parinte copil astfel:

x este parintele lui y, iar **y este copilul lui x.**

De exemplu 2 este parintele lui 7, iar 7 este copilul lui 2.

Definitie

Un nod care nu are copii se numeste **frunza**.

Numarul de copii ai unui nod se numeste **gradul nodului.**

Lungimea drumului de la radacina la un nod x se numeste **adâncimea nodului x.** Cel mai mare grad **n,** dintre gradele tuturor nodurilor, este gradul arborelui si arborele se **numeste arbore n**-ar (pentru n=2, **arbore binar**). De exemplu, arborele din figura este binar.

Definitie

Un arbore binar se defineste în mod recursiv ca fiind o structura definita pe o multime finita de noduri prin:

? Nu contine nici un nod, sau

? Contine trei multimi:

1. un **nod radacina**,

2. un arbore binar numit **subarborele stâng** si
3. un arbore binar numit **subarborele drept.**

5.3. Reprezentarea grafurilor si a arborilor binari

Reprezentarea grafurilor sau a arbori în algoritmi se realizeaza folosind structuri de date statice sau dinamice, respectiv tablouri sau liste.

Un graf G=(X,U) cu ***n*** vârfuri se poate reprezenta prin:

a) Matricea de adiacenta

Matrice patratica de ordin n, definita prin

A(i,j)= 1 , pentru (i,j) din U

A(i,j)=0 , în caz contraj, pentru orice pereche (i,j) din U.

Matricea de adiacenta asociata grafului din Fig. 5.1. (b)este:

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

b) Matricea de incidenta

Se utilizeaza la reprezentarea grafurilor orientate fara bucle, în care multimea arcelor este ordonata si numerotata U={1,2,3,...,m} unde m este numarul arcelor. Astfel matricea are n linii corespuzatoare vârfurilor i=1,n si m coloane corespunzatoare arcelor, definita prin:

B(i,u)= 1 , daca exista j din X astfel încât u=(i,j)

B(i,u)= -1 , daca exista j din X astfel încât u=(j,i)

B(i,u)= 0 , altfel

Aplicatie: Construiti matricea de incidenta asociata grafului din Fig. 5.1. (a), fara bucla nodului 2, cu muchiile numerotate astfel: 1- (1,2), 2- (2,4), 3 - (2,5), 4 - (4,5), 5- (5,4), 6- (4,1).

c) Lista arcelor

Pentru reprezentare se utilizeaza doua tablouri unidimensionale cu m valori care reprezinta extremitatile arcelor din U. Sirurile a(i), b(i), i=1,m, reprezinta extremitatea initiala si respectiv finala a unui arc, u=(a(i), b(i)).

Grafului din Fig. 5.1. (a), fara bucla nodului 2, cu muchiile (1,2), (2,4), (2,5), (4,1), (4,5), (5,4), îi corespund sirurile:

a: 1, 2, 2, 4, 4, 5

b: 2, 4, 5, 1, 5, 4

Obs. Se recomanda ordonarea sirurilor dupa vârfurile initiale si finale.

d) Lista succesorilor (predecesorilor)

Se utilizeaza doua siruri A si B. Sirul A, de lungime n+1 retine pe pozitia i pozitia din sirul B de la care încep succesorii vârfului i. Sirul B contine între pozitiile A(i) si A(i+1)-1 inclusiv, toti succesorii vârfului i. Daca pozitiile A(i) si A(i+1) au aceeasi valoare atunci i nu are succesori.

Grafului din Fig. 5.1. (a), fara bucla nodului 2, cu muchiile (1,2), (2,4), (2,5), (4,1), (4,5), (5,4), îi corespund sirurile:

A: 1, 2, 4, 4, 6

B: 2, 4, 5, 1, 5, 4

(am asezat sugestiv unul sub altul, cele doua siruri. De exemplu i=3 nu are succesori, începând de pe pozitia 4 din B, iar i=4 are 2 succesori începând cu pozitia 4 din sirul B).

Observatie.

Lista predecesorilor se obtine în mod similar precizând pentru fiecare vârf predecesorul sau.

e) *Liste de adiacenta*

Pentru fiecare vârf al grafului neorientat se pastreaza câte o lista care contine toate vârfurile adiacente cu acesta. Numarul listelor va fi egal cu numarul vârfurilor, eventualele vîrfuri izolate vor avea liste vide.

Graful din Fig. 5.1. (b), va contine 6 liste dupa cum urmeaza:

1. 2, 5
2. 1, 5
3. 6
4. lista vida
5. 1, 2
6. 3

Reprezentarea grafurilor orientate folosind liste de adiacenta se realizeaza prin precizarea orientarii, deci lista va contine doar vârfurile adiacente lui i.

f) Reprezentarea arborilor binari

Cea mai frecventa modalitate de reprezentare a arborilor binari se bazeaza pe utilizarea listelor dublu înlantuite.

Definim o structura a listei sub forma:

```
type    arbore=^nod;
        nod=record
                FiuStanga, FiuDreapta:nod;
                inf:TipInformatie;
```

end;

Adresa primului elemnet din lista se considera adresa radacinii.
Arborele din Fig. 5.3. se reprezinta sub forma:

inf	FiuSt	FiuDr
5	4	2
4	3	1
3	nil	Nil
6	nil	Nil

inf	FiuSt	FiuDr
2	nil	7
1	6	8
7	nil	Nil
8	Nil	Nil

Daca în reprezentarea arborilor se doreste deplasarea în structura arborescenta si înspre parinte, se adauga adresa parinteluui fiecarui nod, pentru radacina nil si defintia structurii devine:

```
type    arbore=^nod;
        nod=record
                FiuStanga, FiuDreapta, Parinte:nod;
                inf:TipInformatie;
                end;
```

cu urmatoarea reprezentare pentru arborele de mai sus:

inf	FiuSt	FiuDr	Parinte
5	4	2	nil
4	3	1	5
3	nil	Nil	4
6	nil	Nil	1

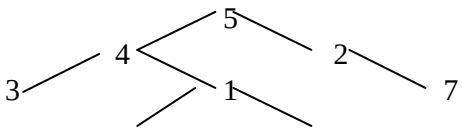
inf	FiuSt	FiuDr	Parinte
2	nil	7	5
1	6	8	4
7	nil	Nil	2
8	Nil	nil	1

Exemple de probleme care folosesc reprezentarea arborilor binari se afla în directorul Probleme\5Grafuri\Arbori.

5.4. Algoritmi elementari

5.4.1. Parcurgerea grafurilor

Sa consideram problema parcurgerii nodurilor unui graf neorientat. Pornind de la un nod oarecare se poate parcurge graful prin alegearea unei muchii incidente în vârful initial, care ne va duce la un nou nod nevizitat. Algoritmul continua prin alegearea din noul vârf a unei muchii incidente cu acesta care sa ne duca într-un vârf nevizitat, si asa mai departe. La epuizarea tuturoe drumurilor dintr-un vârf se revine la vârful anterior si se continua algoritmul pâna la epuizarea vârfurilor. O astfel de parcurgere se numeste parcurgere în adâncime sau „DEPTH-FIRST” (DF).



6 8

Parcurgerea grafului (arborelui) de mai sus folosind algoritmul DF începând cu vârful 1, iar alegearea nodurilor vecine se face în ordine crescatoare, este:

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

Algoritmul care realizeaza parcurgerea „DEPTH-FIRST”, foloseste tehnica Backtracking.

```
Procedura DF ( i )
Vizit (i) = -1
{sirul nodurilor nevizitate se initializeaza cu 0, fiecare nod vizitat e marcat cu -1}
Pentru „orice nod j vecin cu i”  executa
      Daca Vizit (i)=0 atunci
              DF (j)
      Sf daca
Sf Pentru
```

Parcurgerea întregului graf este asigurata prin apelul procedurii DF pentru toate nodurile nevizitate.

```
Pentru i=1,n,1   executa
      Daca Vizit (i)=0 atunci
              DF (i)
      Sf daca
Sf Pentru
```

(Programul Probcurs\capV\df1.pas recursiv, respectiv df.pas varianta nerecursiva)

```
type matrice=array[1..100,1..100] of 0..1;
var a:matrice;
    viz:array[1..100] of 0..1;
    j,n:integer;
    timp:longint absolute $000:$046C;
    timpinit:longint;

Procedure  citiregraf(var a:matrice;var n:integer);
var f:text;
    x,y:byte;
begin
assign(f,'bf1.in');
reset(f);
readln(f,n);
while not (eof(F)) DO
```

```
    begin
    readln(f,x,y);
    a[x,y]:=1;
    a[y,x]:=1;
    end;
close(f);
end;

Procedure back(i:integer);
var j:byte;
begin
write(i,',');
viz[i]:=1;
for j:=1 to n do
    begin
    if (a[i,j]=1) and (viz[j]=0) then
                      back(j);
    end;
end;

begin
timpinit:=timp;
writeln('timpinit: ',timpinit);
citiregraf(a,n);
readln(j);
back(j);
writeln('timp: ',timp);
writeln;
end.
```

Daca la fiecare vârf vizitat se continua cu toate vârfurile adiacente acestuia, vizitate pe rând, acestea în ordine crescatoare, parcurgerea se realizeaza în „latime" sau „BREATH-FIRST" (BF). Parcurgerea grafului de mai sus BF, pornind de la 1 este:

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

Algoritmul care realizeaza parcurgerea „BREATH-FIRST" foloseste acelasi sir Vizit (i) care marcheaza nodurile vizitate dar si o lista de tip coada în care sunt introduse nodurile care au fost vizitate dar ai caror vecini nu au fost inca prelucrati. Procedura BF extrage pe rând câte un element din coada, introduce apoi descendentii sai care nu au fost vizitati si îi viziteaza.

```
Procedura BF ( i )
{se initializeaza coada}
PUSH (i)
```

```
Vizit (i) = 1
{prelucrare nod i}
Cât timp „coada nu este vida” executa
j = POP
     Pentru „toti vecinii k a lui j”  executa
          Daca Vizit (k)=0 atunci
               PUSH (k)
               Vizit (k) = 1
               {prelucrare nod k}

          Sf daca
     Sf Pentru
Sf cât timp
```

(Programul Probcurs\capV\bf1.pas)

```
type matrice=array[1..100,1..100] of 0..1;
     sir=array[1..100] of byte;
var a:matrice;
    x,y:sir;
    i,j,h,ii,p,n,m,t:byte;
    f:text;
    ok:boolean;

Procedure  citire(var a:matrice);
begin
assign(f,'bf1.in');
reset(f);
readln(f,n,m);
for h:=1 to m do
    begin
    readln(f,i,j);
    a[i,j]:=1;
    a[j,i]:=1;
    end;
readln(f,j);
close(f);
assign(f,'bf1.out');
rewrite(f);
end;

begin
citire(a);
y[j]:=1;
t:=1;
```

```
x[t]:=j;
p:=1;
repeat
 for i:=1 to n do
    begin
    if (a[x[p],i]=1) AND (y[i]=0) then
                         begin
                          t:=t+1;
                          x[t]:=i;
                          y[i]:=1;
                         end;
    end;
inc(p);
until (t=n) ;
for i:=1 to n do
    write(f,x[i],' ');
close(f);
end.
```

5.4.2. Conexitatea unui graf

Un graf conex se poate verifica prin apelul procedurilor DF sau BF pornind de la oricare nod si verificând apoi daca toate vârfurile au fost vizitate. În caz afirmativ graful este conex.

Un algoritm similar care verifica daca un graf este conex, pornind de la nodul k, este urmatorul:

```
Procedura Conex ( k )
Pentru i=1,n,1   executa
        Vizit (i)=0
Sf Pentru
BF (k)
Cod = true
I = 1
Cât timp   i<=n and cod   executa
        Daca Vizit (i)=0 atunci
                        Cod=false
        Sf daca
        i=i+1
Sf cât timp
Daca cod atunci
        * Graf conex
                altfel
        * Graf neconex
sf daca
```

(Programul Probcurs\capV\conexit.pas)

```
type matrice=array[1..100,0..100] of byte;
     sir=array[1..100] of byte;
var x:matrice;
    viz,a:sir;
    n,m,t,k,i,j,h:byte;
    f:text;
Procedure citire;
begin
assign(f,'conexit.in');
reset(f);
readln(f,n);
for i:=1 to n do
    begin
    viz[i]:=0;
    x[i,0]:=0;
    end;
while not eof(f) do
      begin
      readln(f,i,t);
      x[i,0]:=x[i,0]+1;
      x[i,x[i,0]]:=t;
      x[t,0]:=x[t,0]+1;
      x[t,x[t,0]]:=i;
      end;
close(f);
end;
begin
citire;
write('introduceti varful a carui vecini ii doriti:
');readln(m);
viz[m]:=1;
a[1]:=m;
k:=1;
for i:=2 to x[m,0]+1 do
    if viz[x[m,i-1]]=0 then
    begin
    k:=k+1;
    viz[x[m,i-1]]:=1;
    a[k]:=x[m,i-1];
    end;
i:=1;
repeat
```

```
i:=i+1;
for j:=1 to x[a[i],0] do
    begin
    if viz[x[a[i],j]]=0 then
       begin
       viz[x[a[i],j]]:=1;
       k:=k+1;
       a[k]:=x[a[i],j];
       end;
    end;
until i=k;
writeln('vecini acestui varf sunt: ');
for i:=1 to k do
    write(a[i],' ');
writeln;
readln;
end.
```

5.4.3 Arbori partiali

Daca unui graf G = (X,U) atasam fiecarei muchii un cost, c(i,j), se pune problema determinarii unui arbore partial, al grafului dat, cu proprietatea ca suma costurilor muchiilor arborelui este minima. Aceasta problema este cunoscuta ca problema determinarii arborelui partial de cost minim într-un graf dat.

Un algoritm care determina acest arbore prin alegerea muchiilor în ordine crescatoare a valorilor costurilor cu conditia ca muchia aleasa sa nu determine un ciclu împreuna cu cele deja alese, Algoritmul lui Kruskal.

```
Procedura Kruskal (G)
{Ordoneaza lista muchiilor crescator dupa cost c(i,j), fie aceasta u_k, k=1,m}
i=1
a(i)=u(1)
Pentru k=2,m,1 executa
Daca „u(k) nu formeaza cicluri” atunci
       i=i+1
       a(i)=u(k)
sf daca
Sf pentru
```

(Programul Probcurs\capV\apm.pas)

```
type sir=array[1..100] of byte;
     as=record
     i,j,k:byte;
```

```
      end;
      asd=array[1..100] of as;
var a:asd;
    x,y:sir;
    ok:boolean;
    d,n,m,i,j,k,t,s,p:byte;

Procedure citire;
var f:text;
begin
assign(f,'apm.in');
reset(f);
readln(f,n);
t:=0;
m:=1;
while not eof(f) do
      begin
      readln(f,i,j,k);
      t:=t+1;
      if i>m then m:=i;
      if j>m then m:=j;
      a[t].i:=i;
      a[t].j:=j;
      a[t].k:=k;
      end;
p:=m;
m:=t;
close(f);
for i:=1 to p do
    x[i]:=i;
end;

Procedure ordonare;
begin
for i:=1 to m-1 do
    for j:=i+1 to m do
        if a[j].k<a[i].k then
        begin
        t:=a[i].i;
        a[i].i:=a[j].i;
        a[j].i:=t;
        t:=a[i].j;
        a[i].j:=a[j].j;
        a[j].j:=t;
```

```
        t:=a[i].k;
        a[i].k:=a[j].k;
        a[j].k:=t;
        end;
end;

begin
citire;
ordonare;
s:=0;
t:=0;
i:=0;
repeat
i:=i+1;
if not (x[a[i].i]=x[a[i].j]) then
    begin
    s:=s+a[i].k;
    t:=t+1;
    y[t]:=i;
    if x[a[i].i]<x[a[i].j] then
            begin
            d:=x[a[i].j];
             for j:=1 to n do
                     if x[j]=d then
                          x[j]:=x[a[i].i];
             end
    else if x[a[i].i]>x[a[i].j] then
                      begin
                      d:=x[a[i].i];
                      for j:=1 to n do
                       if x[j]=d then
                         x[j]:=x[a[i].j];
                      end;
   end;
ok:=false;
for j:=1 to p-1 do
    if not (x[j]=x[j+1]) then ok:=true;
until ((t=p-1) and (ok=false)) or (i=m);
writeln('s= ',s);
for i:=1 to p-1 do
    write('(',a[y[i]].i,',',a[y[i]].j,') ');
writeln;
writeln;
readln;
```

```
    end.
```

VI. Teste

Test 1

1. *"Principiul cutiei negre"* se bazeaza pe urmatoarele principii:

a) programatorul nu cunoaste nimic despre structura interna a programului si despre datele de intrare.

b) reprezinta totala independenta a programatorului în interiorul programului sau, legatura cu exteriorul facâdu-se strict pe baza formatului datelor de intrare si a celor de iesire

c) se impune respectarea unor reguli interne deci a unui standard vizibil si din exteriorul programatorului.

2. Evitarea folosirii specificului calculatorului, cum ar fi elemente de afisaj specific (rezolutii, placi grafice, tipuri de interfete), transmisie de date (retele locale, unitati de disc specifice), dispozitive periferice (tipuri de imprimante) si respectiv medii de programare sau sisteme de operare, reprezinta principiul enuntat de:

a) Verifica corectitudinea algoritmului si a programului în fiecare etapa a elaborarii.
b) Asigurara portabilitatea programului
c) Proiecteaza structurat algoritmii

3. Programul de mai jos realizeaza urmatoarele operatii:

```
Var var_fis:text;
   C: char;
Begin
   Assign(var_fis, 'TEST.DAT');
   Reset(var_fis);
   While not Eof(var_fis) do
   begin
     Readln(var_fis,c)
     WriteLn(c)
    end;
   Close(var_fis)
 end.
```

a) Deschide fisierul TEST.DAT, de tip text, citeste primul caracter de pe fiecare linie din fisier si afiseaza caracterul pe ecran pe câte o linie fiecare.
b) Deschide fisierul TEST.DAT, de tip text, citeste caracter cu caracter din fisier pâna la sfârsitul fisierului si afiseaza caracterele pe ecran pe o singura linie.
c) Deschide fisierul TEST.DAT, de tip text, citeste primul caracter din fisier pâna la sfârsitul fisierului si afiseaza caracterele pe o linie pe ecran.

4. Fie urmatoarele definitii de tipuri si variabile:

```
type    sir=array[0..10] of real;
        mat= array[1..5] of sir;
var     a,b:sir;
        x:mat;
        k:integer;
```

Care din urmatoarele operatii sunt corecte:

a) k:=4; a[1,k]=3.5 b) k:=0; b[k]:=a[7] c) x[3]:=a d) x[1,0]:=b[5]

5. Urmatoarea declaratie de tip corespunde unei structuri dinamice de tip:

```
type nod=^elem;
    elem=record
        st,dr:nod;
        inf:integer;
       end;
```

a) coada b)stiva c) incorecta c) lista dublu înlantuita

6. Pentru polinomul $P(X)=7X^{50}+15X^{5}-4X^{3}+6$ reprezentati coeficientii si gradul lor folosind o lista simplu înlantuita.

7. Descrieti o functie de adaugare a unui element într-o stiva.
Antetul functiei, folosind definitiile de date de la exercitiul 6, este:

```
Function AdaugaStiva(S:nod; e:elem):nod;
```

8. Scrieti procedura de citire a unui graf, dat ca perechi de muchii într-un fisier text, pentru care se creaza matricea de adiacenta. Procedura va avea urmatorul antet:

```
Procedure graf(var f:text;var n:byte;var a:matrice);
```

1. b); 2. b); 3. a); 4. b),d); 5) c.

Test 2

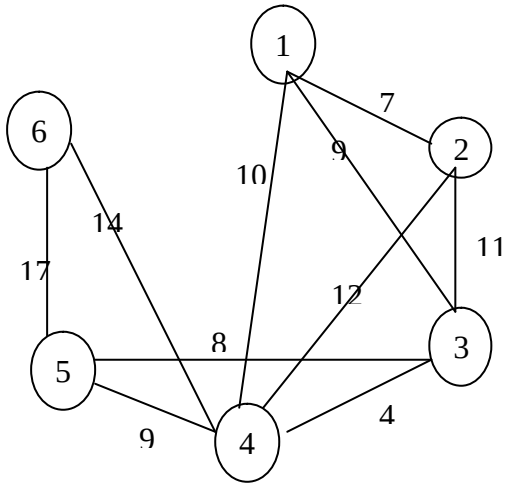
1. Functia Valid împreuna cu procedura backtracking descrise mai jos rezolva problema:

```
 Function valid(i:byte):boolean;
Var cod:boolean;
    j:byte;
begin
cod:=true;
for j:=1 to i-1 do
    if x[i]=x[j] then cod:=false;
valid:=cod;
end;
```

```
procedure Back(j:byte);
var  i:byte;
begin
for i:=1 to n do
begin
  x[j]:=i;
  if valid(j) then
    if j<k then Back(j+1)
         else solutie(x,k);
end;
```

a) Permutarilor b) Combinarilor c) Aranjamentelor

2. Parcurgerea în adâncime (DF), pornind de la vârful 3, a grafului din figura de mai jos este:



a) 3,1,2,5,4,6 b) 3, 1, 2, 4, 5, 6 c) 3,2,4,1,5,6

3. Alegeti corespunzator liniile care lipsesc din urmatorul algoritm pentru a descrie corect tehnica Greedy**:**

```
Algoritm Greedy (A, n, B);
Cod : boolean;
B ? {Multimea vida};
Pentru i=1, n,1 executa
          _____________________
          _____________________
          Daca cod atunci
```

```
                Adaug (B, x);
            Sf daca;
Sf pentru;
```

a) **Alege (A,n,x)** b)**Alege(A,i,x)** c)**Alege(A,i,x)**
Posibil(A,x,cod) Posibil(B,n,cod) Posibil(B,x,cod)

4. Descrieti algoritmul de sortare prin selectie
5. Descrieti algoritmul de cautare binara
6. Arborele partial de cost minim corespunzator grafului (2) are costul:

a) 39 b) 42 c) 38 d) 40

1. c) 2. b) 3. c) 6. b)

VII. Bibliografie

1. Andone R., Gârbacea I., Algoritmi fundamentali o perspectiva C++, Editura Libris, Cluj Napoca, 1995.
2. Böhm C., Jacopini T.C., Flow diagrams, Turing Machines and Languages with only two Formation Rules, CACM 9, 5, 1966.
3. Calude C., Complexitatea calculului, aspecte calitative, Editura Stiintifica si Enciclopedica, Bucuresti, 1982.
4. Cormen T.H., Leiserson E.C., Rivest R.R., Introducere în algoritmi, Editura Libris Agora, 2000 (traducere în limba română).
5. Dahl O.J., Dijkstra E.W., Hoare C.A.R., Structured Programing, Academic Press, 1972.
6. Lazar, M. Frentiu, S. Motogna, V. Prejmerean, Elaborarea algoritmilor, Ed. Presa Universitara Clujeana, Cluj Napoca, 1998
7. Lazar, M. Frentiu, S. Motogna, V. Prejmerean, Programare Pascal, Ed. Presa Universitara Clujeana, Cluj Napoca, 1998
8. Lica D., Onea E., Informatica, Manual pentru clasa a IX-a, Editura L&S, Bucuresti, 1999.
9. Livovschi L., Georgescu H., Sinteza si analiza algoritmilor, Editura Stiintifica si Enciclopedica, Bucuresti, 1986.
10. McConnell S.C., Code Complete: a practical handbook of software construction, Microsoft Press, Washington, 1993
11. Mitrana V., Provocarea algoritmilor, Editura Agni, Bucuresti, 1994.
12. Motogna S.,Metode si tehnici de proiectare a algoritmilor, Universitatea “Babes Bolyai”, Cluj Napoca, 1998
13. Pólya G., How to solve it? A new aspect of mathematical method, Princeton Univerity Press, 1957.
14. Rancea D., Limbajul Pascal. Algoritmi fundamentali., Editura Computer Libris Agora, Cluj Napoca, 1999.
15. Tudor S., Cerchez E., Serban M., Informatica. Manual pentru clasa a IX-a, Editura L&S, Bucuresti, 1999.
16. Logofatu Hrinciuc E., Probleme rezolvate si algoritmi, Editura Polirom, 2001.
17. Patrascoiu O., Marian G., Mtroi N., Elemente de grafuri si combinatorica. Metode, algoritmi si programe, Editura All, Bucuresti, 1994.

Cuprins