marin popa mariana popa



ELEMENTE DE ALGORITMI ȘI LIMBAJE DE PROGRAMARE



EDITURA UNIVERSITĂȚII DIN BUCUREȘTI

https://biblioteca-digitala.ro / https://unibuc.ro

https://biblioteca-digitala.ro / https://unibuc.ro

Marin Popa • Mariana Popa

ELEMENTE DE ALGORITMI ŞI LIMBAJE DE PROGRAMARE

https://biblioteca-digitala.ro / https://unibuc.ro

BIBLIOTECA CENTRALĂ UNIVERSITARĂ
BUCUREȘTI
COTA III 479070

578/05

© Editura Universității din București
Șos. Panduri, 90-92, București – 050663; Telefon/Fax: 410.23.84
E-mail: editura_unibuc@yahoo.com
Internet: www.editura.unibuc.ro

B.C.U. Bucuresti
C20053508

Descrierea CIP a Bibliotecii Naționale a României
POPA, MARIN
Elemente de algoritmi și limbaje de programare / Marin Popa, Mariana Popa – București: Editura Universității din București, 2005

Bibliografie

ISBN 973-575-979-9
I. Popa, Mariana

004.43

https://biblioteca-digitala.ro / https://unibuc.ro

**"Pentru Arthur, ale cărui aripi
tocmai se deschid spre viitor..."**

Introducere

Prima maşină inteligentă, acest copil minune al secolului XX fără de care omul nu ar fi putut explora universul – atestată de ştiinţa modernă, a fost gândită şi construită de Blaise Pascal în 1642. Abia peste 300 de ani apare primul sistem de calcul modern de mare capacitate, construit de firma IBM în 1944 şi instalat la Universitatea Harward. Astfel putem spune că istoria sistemelor moderne de calcul se întinde pe numai ultimii 60 de ani.

Începând cu anii '50 industria sistemelor de calcul a cunoscut o dezvoltare rapidă mai ales după crearea circuitelor integrate care au făcut posibilă apariţia calculatoarelor personale.

Primele calculatoare aveau dimensiuni impresionante ocupând camere întregi şi fiind dispuse pe rafturi metalice. Ele erau formate din sute de tuburi cu vid, cuve cu mercur şi panouri de semnalizare optică. Asemănarea acestui conglomerat cu un atelier de fierărie era aşa de mare încat proiectanţii acestor sisteme de calcul vorbeau despre ele numindu-le hardware (articole de fierărie, sau echipament solid).

Sistemele de calcul au avut în timp o evoluţie tehnică spectaculoasă, ultimul deceniu al mileniului trecut fiind reprezentativ pentru acest proces de perfecţionare a ceea ce noi azi numim calculatoare. Ele au devenit indispensabile în orice domeniu de activitate şi astfel studiul modului de operare şi exploatare a lor a devenit o necesitate atât pentru programatori cât şi pentru utilizatorii avansaţi.

Putem considera calculatorul electronic ca pe o maşină automată care are ca funcţie de bază prelucrarea informaţiei şi care este capabilă să execute operaţii complexe folosind un program activ prezent în memoria sa proprie.

În prezent un calculator, cu o putere de calcul mult mai mare decât a celor din anii '50, însă cu aceleaşi principii de operare, poate încăpea într-o servietă, aşa cum ar fi laptop-ul. Calculatoarele au devenit de sute de ori mai rapide decât cele de acum 20 de ani şi de cel puţin două ori mai ieftine.

https://biblioteca-digitala.ro / https://unibuc.ro

În condiţiile în care Internetul şi World Wide Web aduc imense cantităţi de informaţii direct pe calculatorul nostru de birou , evident că nu ne mai putem îndoi de faptul că trăim în era informaticii.

Prin această carte, autorii îşi exprimă concepţia asupra modului de abordare a cursului numit « Informatică » de la Facultăţile de Informatică şi de la cele de Matematică-Informatică pentru specializările Informatică şi Matematică-Informatică. Utilizarea eficientă a calculatorului pentru rezolvarea de probleme din orice domeniu nu este posibilă fără o iniţiere în domeniul algoritmilor, al metodelor şi tehnicilor de elaborare a acestora dar şi al complexităţii lor. Acesta este motivul pentru care prima parte a lucrării dezvoltă metodele de reprezentare a algoritmilor, prezintă câteva metode şi tehnici de elaborare a algoritmilor şi studiază complexitatea algoritmilor elaboraţi prin fiecare din aceste tehnici.

Structurată pe trei părţi, primele două având câte 4 capitole şi ultima două capitole, lucrarea tratează aspectele fundamentale privind reprezentarea elaborarea şi complexitatea algoritmilor (în prima parte), privind limbajul C++, programarea orientată obiect, implementarea structurilor de date de tip stivă, coadă, arbore în C++ şi privind grafica pe calculator în C++ (partea a doua) şi în fine privind modalităţi eficiente de realizare a paginilor Web interactive şi dinamice (partea a treia).

Cartea conţine un mare număr de exemple complete de programare C++ şi JavaScript, introduse sistematic ca o modalitate eficientă de explicare şi însuşire a limbajelor, venind în acelaşi timp şi în sprijinul celor obişnuiţi să lucreze în alte limbaje precum: Fortran, Basic, Pascal, etc.

Prima parte a lucrării reflectă concepţiile cristalizate pe plan mondial în deceniul 8 al secolului trecut, privind metodologia elaborării algoritmilor. Din modul de prezentare a tehnicilor de elaborare a algoritmilor rezultă că a şti să programezi nu înseamnă numai a cunoaşte limbaje de programare şi siteme de operare , ci înseamnă şi a stăpâni perfect metode şi tehnici de reprezentare şi elaborare a algoritmilor. Se evidenţiază de asemenea rolul pe care îl au cunoştinţele matematice în analiza eficienţei spaţiale şi temporale a algoritmilor.

Partea a doua porneşte de la faptul că un limbaj, fie el natural sau artificial, constituie principalul canal de comunicaţie a metodelor de rezolvare a problemelor care apar în diverse domenii ale activităţii umane fie ele teoretice sau aplicative. În categoria limbajelor artificiale pe lângă

https://biblioteca-digitala.ro / https://unibuc.ro

limbajele formale întâlnim şi limbajele de programare a calculatoarelor electronice.

Limbajul C a fost proiectat în 1972 de programatorii Denis Ritchie şi Brian Kernigham de la Bell Laboratories, ca o alternativă la limbajul de asanblare. El este atât un limbaj de sistem, adecvat scrieii de compilatoare, editoare de texte, sisteme de operare, etc. cât şi limbaj de nivel înalt pentru programe de gestiune, calcule ştiinţifice, jocuri cu facilităţi grafice etc. Limbajul C permite utilizarea unui număr mare de funcţii repartizate în diverse fişiere sursă sau biblioteci de funcţii sursă, stimulând astfel programarea structurată şi modulară.

Pe la sfârşitul anilor '80, Bjarne Stroustrup de la Bell Laboratories, pune bazele limbajului C++ pornind de la limbajul C pe care îl dezvoltă şi îl completează cu elemente de programare orientată obiect. C++ păstrează caracterisicile limbajului C referitoare la portabilitate şi flexibilitate, însă extensiile şi completările aduse îl fac să fie universal, mai puternic şi mai riguros. Completările privind elementele de programare orientată obiect conduc la creşterea productivităţii programării unor module reutilizabile, prin creşterea independenţei între componentele programelor şi prin facilitarea depanării şi întreţinerii produselor soft.

Obiectul, element de bază în programarea orientată obiect este un ansamblu format dintr-o structură de date utilizator şi din funcţii (numite şi metode) necesare prelucrării datelor acestei structuri. **Clasa**, de asemenea concept de bază în C++ descrie o suită de obiecte cu aceiaţi structură a datelor şi aceleaşi metode de prelucrare. Ea oferă facilităţi eficiente pentru încapsularea datelor şi metodelor, pentru iniţializarea şi conversia implicită a datelor, pentru supraîncărcarea operatorilor etc. C++ fiind într-o continuă evoluţie a făcut ca în 1996 să fie lansat standardul ANSI C++.

Partea a treia a lucrării a fost introdusă în curs pornind de la observaţia că în ultimii ani un interes deosebit s-a manifestat în scrierea de pagini Web.

HTML este principalul limbaj folosit pentru a crea pagini Web. Documentele HTML sunt fişiere text ce pot fi create cu orice editor , dar care trebuie salvate sub forma de text simplu. Pentru a vedea cum arată documentul creat, el trebuie deschis într-un browser (program de navigare pe Internet) sau chiar în editorul WYSIWYG care permite vizualizarea documentului chiar în timpul editării lui. Acest ultim editor , care permite editarea unui document folosind mijloace vizuale, este de

https://biblioteca-digitala.ro / https://unibuc.ro

mare ajutor pentru documente simple, însă pentru documente complexe, eficiente privind viteza de redare, este necesară cunoaşterea elementelor de bază ale limbajului HTML.

Ca un complement al limbajului HTML, menit să aducă iteractivitate şi dinamism paginilor Web, a fost conceput limbajul de tip script numit JavaScript. El permite accesarea elementelor unui document HTML şi modificarea parametrilor acestora în urma unor acţiuni ale utilizatorilor. Codul JavaScript este inserat într-un document HTML folosind anumite marcaje speciale.

Nu trebuie confundat cu Java, al cărui domeniu de utilizare depăşte cu mult paginile Web permiţând dezvoltarea de aplicaţii sofisticate cu un înalt grad de securitate.

Abordarea graduată a lucrării, pornind de la concepte de bază simple şi mergând până la implementări performante ale unor algoritmi în programarea orientată pe obiecte, în operaţii cu structuri de date dinamice şi în grafica pe calculator, până la realizarea unor pagini Web profesioniste, recomandă această carte unei categorii largi de cititori interesaţi de aspectele conţinute în ea, de la neiniţiaţi în programarea pe calculator până la programatori cu multă experienţă în domeniu.

Mulţumim anticipat, pe această cale, tuturor colegilor, studenţilor, programatorilor profesionişti care, după parcurgerea acestei cărţi, prin observaţiile şi sugestiile lor, vor contribui la o eventuală îmbunătăţire şi completare a ei.

Aşteptăm sugestiile, observaţiile şi recomandările dumneavoastră la adresa de mail: marpopa2002@yahoo.com.

Autorii

https://biblioteca-digitala.ro / https://unibuc.ro

CUPRINS

PARTEA I - ALGORITMICĂ

CAPITOLUL 1. Reprezentarea algoritmilor . . 17

1.1 Noțiunea de algoritm . 17
1.1.1 Scurt istoric . . . 17
1.1.2 Definiția noțiunii de algoritm . . 18
1.1.3 Studiul algoritmilor . . 21
1.1.4 Reprezentarea algoritmilor . . . 22
1.2 Reprezentarea algoritmilor prin scheme logice . 26
1.3 Structuri elementare ce stau la baza programării structurate . . . 32
1.4 Programare structurată 38
1.5 Conceptul NL (Nested Logic) de proiectare și programare 44
1.6 Reprezentarea algoritmilor în pseudocod . . 54
1.7 Reprezentarea algoritmilor prin tabele de decizie . 61

CAPITOLUL II Structuri elementare de date, modularizare si codificarea informatiilor 67

2.1 Structuri statice; tablouri . 67
2.2 Structuri dinamice . . . 73
2.2.1 Operații pe mulțimi dinamice . . 74
2.2.2 Liste înlănțuite. . 75
2.2.3 Stive și cozi . 80
2.2.3.1 Stive . . 80

https://biblioteca-digitala.ro / https://unibuc.ro

2.2.3.2 Cozi 82
2.3 Descompunerea problemelor în subprobleme – modularizare 84
2.4 Codificarea informațiilor în memoria calculatoarelor electronice 92
2.4.1 Sisteme de numerație 92
2.4.2 Conversia unui număr real din sistemul zecimal într-un sistem de bază dată și invers 93
2.4.2.1 Conversia din zecimal în sistemul cu baza b≥2 93
2.4.2.2 Conversia unui număr din baza b≥2 în zecimal 94
2.4.2.3 Conversia binar→octal, hexazecimal și invers a unui numar real . . . 94
2.4.3 Coduri pentru reprezentarea datelor . . 96
2.4.4 Reprezentarea numerelor în calculator . 97
2.4.4.1 Reprezentarea numerelor naturale . 97
2.4.4.2 Reprezentarea numerelor întregi . 98
2.4.4.3 Reprezentarea numerelor reale . . 99
2.4.4.3.1 Reprezentarea numerelor reale în virgulă mobilă cu simplă precizie 100
2.4.4.3.2 Reprezentarea numerelor în virgulă mobilă cu precizie dublă . .101

CAPITOLUL III. Elaborarea algoritmilor . 103

3.1 Introducere 103
3.2 Indicii de abordare a unei probleme prin tehnicile de elaborare a algoritmilor 104
3.3 Privire comparativa a tehnicilor de elaborare a algoritmilor 106
3.4 Descrierea tehnicilor de elaborare a algoritmilor .112
3.4.1 Tehnica backtracking . . .112
3.4.1.1 Prezentare . . .112
3.4.1.2 Backtracking generalizat .117
3.4.1.3 Backtracking recursiv .119
3.4.2 Recursivitatea . .123
3.4.2.1 Prezentare . . .123
3.4.2.2 Tipuri de recursivitate .124

https://biblioteca-digitala.ro / https://unibuc.ro

3.4.2.3 Execuția programelor recursive . .126
3.4.2.4 Algoritm pentru eliminarea recursivității liniare .130
3.4.2.5 Concluzii . .131
3.4.3 Metoda divide et impera .133
3.4.4 Metoda greedy .137
3.4.4.1 Prezentare 137
3.4.4.2 Bazele teoretice ale metodei greedy : .140
3.4.4.2.1 Matroizi 140
3.4.4.2.2 Algoritmi GREEDY pe un matroid ponderat . . .143
3.4.5 Programarea dinamică .149
3.4.5.1 Prezentare .149
3.4.5.2 Concluzii 151
3.4.5.2.1 Substructura optimă. . .152
3.4.5.2.2 Suprapunerea subproblemelor .152
3.4.5.2.3 Memorizarea .152

CAPITOLUL IV Complexitatea algoritmilor . .155

4.1 Generalități . .155
4.2 Noțiuni de bază .159
4.3 Notații asimptotice: 162
4.4 Analiza complexitatii tehnicilor de elaborare a 165 Algoritmilor . . .165
4.4.1 Metoda backtracking . .165
4.4.2 Recursivitatea . .166
4.4.3 Divide et impera .167
4.4.4 Metoda greedy . .169
4.4.5 Programarea dinamică .170

https://biblioteca-digitala.ro / https://unibuc.ro

PARTEA a II-a - LIMBAJE DE PROGRAMARE

CAPITOLUL I Elemente de C şi C++ . . .175

1.1 Introducere . . .175
1.2 Directive de preprocesare . .176
1.3 Operaţii de intrare/ieşire . . .178
1.3.1 Fluxuri de intrare/ieşire. .178
1.3.1.1 Fluxul de ieşire cout . .179
1.3.1.2 Fluxul de intrare cin . .180
1.3.2 Manipulatori .181
1.4 Variabile şi constante . .184
1.5 Tipuri şi structuri de date . .186
1.5.1 Tipuri de date fundamentale .186
1.5.2 Tablouri. . .190
1.5.2.1 Vectori. .190
1.5.2.2 Matrice . .191
1.5.3 Definirea tipurilor de date .192
1.5.4 Structuri. .193
1.5.5 Uniuni . .195
1.6 Operatori . .196
1.7 Instructiuni .200
1.8 Pointeri 208
1.8.1 Pointeri la variabile simple .209
1.8.2 Pointeri la tablouri . .210
1.8.3 Pointeri la şiruri de caractere .212
1.8.4 Pointeri la structuri .214
1.9 Funcţii 215
1.9.1 Definirea funcţiilor .215
1.9.2 Prototipul unei funcţii . .217
1.9.3 Funcţii recursive 218
1.9.4 Funcţii cu număr variabil de parametrii .220
1.9.5 Parametrii cu valori implicite .221

https://biblioteca-digitala.ro / https://unibuc.ro

1.9.6 Supradefinirea funcțiilor .222
1.9.7 Tipuri speciale de parametrii .223
1.9.7.1 Parametrul tablou . .223
1.9.7.2 Parametrul şir de caractere .223
1.9.7.3 Parametrul structură . .224
1.9.8 Transferul parametrilor prin referință . .225
1.9.9 Pointeri la funcții .227
1.9.10 Funcții de biblioteca .230
1.9.11 Funcții inline . .233

CAPITOLUL II Elemente de programare orientată obiect (POO) . . .235

2.1 Concepte de bază . .235
2.2 Declararea claselor .237
2.3 Funcții membre inline .241
2.4 Constructori 242
2.4.1 Declarare şi caracteristici . .242
2.4.2 Supradefinirea constructorilor .245
2.4.3 Constructor de copiere .245
2.5 Destructori . . .247
2.6 Membri statici ai unei clase .249
2.6.1 Date membre statice . .249
2.6.2 Funcții membre statice . .252
2.7 Funcții prietene .253
2.8 Clase prietene . . .256
2.9 Supradefinirea operatorilor .259
2.10 Funcții operator prietene .260
2.11 Funcții operator member . . .262
2.12 Directive condiționale de preprocesare . .262
2.13 Complemente . . .265
2.13.1 . Insertor şi extractor . .265
2.13.2 Stream de intrare .265
2.13.3 Stream de ieşire 265
2.13.4 Funcții la nivel de caracter asociate unui stream de intrare şi de ieşire 266
2.13.5 Funcții de gestiune a ecranului şi tastaturii .268
2.13.6 Vizibilitatea şi durata de viață a variabilelor . .269

https://biblioteca-digitala.ro / https://unibuc.ro

2.13.7. Clase de memorare . .270
2.13.8. Categorii de pointeri . . .271
2.13.9 Alocarea dinamică a memoriei .273
2.13.10 Referințe . .274
2.13.11 Probleme rezolvate 276
2.13.12 Pointeri ce memorează adrese de funcții .279
2.13.13 Supraîncărcarea funcțiilor 280
2.13.14 Clase, obiecte, metode, specificatori de acces .281
2.13.15 Constructori . .283
2.13.16 Destructori 285
2.13.17 Supraîncărcarea operatorilor .286
2.14 Aplicații C si C++ .294
2.14.1 Recursivitate . . .294
2.14.1.1 Probleme rezolvate . .294
2.14.2 Divide et impera .297
2.14.2.1 Descriere . .297
2.14.2.2 Probleme rezolvate: . .298
2.14.3 Backtracking . .309
2.14.3.1 Descriere . .309
2.14.3.2 Probleme rezolvate . .312
2.14.4 Metoda Greedy .325
2.14.4.1 Descriere . . .325
2.14.4.2 Probleme rezolvate: .326
2.14.5 Programarea dinamică .238
2.14.5.1 Descriere . .328
2.14.5.2 Probleme rezolvate . .330

CAPITOLUL III Implementarea în C++ a structurilor de date şi a operațiilor cu fişiere. 337

3.1 Structuri de date de tip listă .337
3.1.1 Stive 338
3.1.1.1 Alocarea secvențială a unei stive .338
3.1.1.2 Alocarea înlănțuită a unei stive .342
3.1.2 Cozi 347
3.1.2.1 Alocarea secventiala a unei cozi .347
3.1.2.2 Alocarea înlănțuită a unei cozi .351

https://biblioteca-digitala.ro / https://unibuc.ro

3.2 Structuri de date de tip arbore .355
3.2.1 Arbori binari .355
3.2.2 Arbori oarecare .360
3.3 Fişiere în C ++ .364

CAPITOLUL IV Elemente de grafică pe calculator .373

4.1 Generalităţi . .373
4.2 Fişierul <graphics.h> 374
4.2.1 Iniţializarea modului de lucru grafic folosind funcţiile 375
4.2.2 Pagini grafice . . .375
4.2.3 Gestiunea culorilor .376
4.2.4 Funcţii pentru ferestre . .377
4.2.5 Relativ la cursorul grafic .378
4.2.6 Relativ la culori 378
4.2.7 Relativ la puncte şi segmente de dreaptă . .379
4.2.8 Relativ la grosime, stil şi mod de desenare a liniilor . 379
4.2.9 Relativ la cercuri, arce şi curbe .380
4.2.10 Relativ la haşuri 381
4.2.11 Privitor la salvări şi restaurări de imagini . 382
4.2.12 Privitor la funcţiile orientate spre texte .382
4.3 Elemente de geometrie .385
4.3.1 Dreapta .385
4.3.2 Cercul 386
4.3.2 Conice pe ecuaţii reduse .387
4.3.3.1 Elipsa . .387
4.3.3.2 Parabola .387
4.3.3.3 Hiperbola 387
4.3.4 Curbe algebrice clasice şi transcendente (curbe nealgebrice) . .387
4.3.5 Curbe şi suprafeţe . .388
4.4 Bazele matematice ale graficii 2D .388
4.5 Bazele matematice ale graficii 3D .394
4.6 Obiecte grafice 403
4.7 Reprezentarea grafică a obiectelor 3D 417
4.8 Aplicaţii. Programe pentru rezolvarea unor probleme de geometrie în spaţiu 419
4.9 Metode şi funcţii pentru realizare de animaţie . .424
4.10 Fractali . .439

https://biblioteca-digitala.ro / https://unibuc.ro

PARTEA a III-a - ELEMENTE DE PROGRAMARE WEB

CAPITOLUL I Limbajul HTML. . . .447

1.1 Introducere (Internet, HTML, Java, URL) .447
1.2. Marcaje . .450
1.3 Stiluri de text . . .453
1.4 Liste si caractere speciale. . .455
1.4.1 Liste neordonate (marcate) .456
1.4.2 Liste ordonate (numerotate) .458
1.4.3 Caractere speciale .460
1.5 Spații de afișare și comentarii .461
1.6 Referințe și legături 462
1.6.1 Referințe interne în cadrul documentelor .466
1.7 Imagini grafice si fragmente audio .469
1.8 Tabele si cadre .476
1.9 Formulare. . .482
1.10 Hărți de imagini . . .487
1.11 Noi marcaje în HTML 4.0 . .488
1.12 Pagini HTML dinamice si JavaScript .490

CAPITOLUL 2. JavaScript 493

2.1 Introducere 493
2.2 Structura programelor JavaScript .497
2.3 Elementele de bază ale limbajului .501
2.4 Obiectele limbajului JavaScript 505
2.5 Definirea și folosirea propriilor obiecte în JavaScript .507
2.6 Obiecte specifice limbajului JavaScript . .509
2.7 Funcții proprii ale lui JavaScript . .512
2.8 Obiectele specifice părții client . .513
2.8.1 Obiectul window .513
2.8.1.1 Metode .513
2.8.1.2 Proprietăți .515
ANEXA 517
BIBLIOGRAFIE 527

https://biblioteca-digitala.ro / https://unibuc.ro

Partea I

Algoritmică



- Reprezentare
- Elaborare
- Complexitate

https://biblioteca-digitala.ro / https://unibuc.ro

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL 1. Reprezentarea algoritmilor

1.1 Noţiunea de algoritm

1.1.1 Scurt istoric [9, 37]

În secolul al VIII-lea d.Hr., un autor persan, ABU JA'FAR MOHAMED IBN AL- KHOWARIZMI a scris o carte de matematică, "Liber algorithmi", unde "algorithm" provine de a al-Khowarizmi ceea ce înseamnă "din oraşul KOWARIZM" azi oraşul KHIVA din UZBECHISTAN. Acest autor, ca şi alţi matematicieni ai evului mediu înţelegeau prin algoritm o regulă pe baza căreia se pot efectua calcule matematice.

Mai târziu prin secolul al XVI-lea noţiunea de algoritm o întâlnim la matematicieni ca Riese, Stifer, Cardano, Leibniz etc, care în lucrările lor foloseau algoritmi pentru dublări, înjumătăţiri, înmulţiri şi împărţiri de numere, etc.

Multă vreme conceptul de algoritm rămane cu o întrebuinţare destul de restrânsă chiar şi în matematică.

Către sfârşitul secolului al XIX-lea (1886-1888) Kronecker şi Dedekind introduc în matematică funcţiile recursive în care conceptul de algoritm este strâns legat de cel de recursivitate. Abia în secolul XX, în deceniile 3 şi 4, prin lucrările matematicienilor Skolem, Ackerman, Sudan, Gödel, Church, Kleene, Turing şi alţii, teoria algorimilor şi recursivităţii se constituie ca atare.

Astăzi, ca rezultat al conexiunii dintre algoritm şi calculator gândirea algoritmică s-a transformat dintr-un instrument matematic particular, într-o modalitate fundamentală de abordare a problemelor din domenii ce aparent nu au nimic comun cu matematica.

Astfel prin algoritm înţelegem azi o metodă generală de rezolvare a unui anumit tip de probleme, metodă care se poate implementa pe calculator prin intermediul unui limbaj de programare.

Evoluţia spectaculoasă a informaticii din ultimele decenii a condus la apariţia a numeroase limbaje de programare şi a numeroase metode de elaborare şi reprezentare a unor algoritmi performanţi.

https://biblioteca-digitala.ro / https://unibuc.ro

Un concept nou, foarte eficient, apărut în ultima vreme este cel de programare orientată pe obiecte, unde prin obiect înțelegem o entitate ce cuprinde atât datele cât și procedurile ce operează cu ele.

Dintre limbajele orientate pe obiecte, C++ permite o exprimare clară și concisă a algoritmilor fapt ce usurează transcrierea în acest limbaj a algoritmilor descriși prin scheme logice, pseudocod sau tabele de decizie.

1.1.2 Definiția noțiunii de algoritm [9, 18, 37]

Noțiunea de algoritm este una primară, ca și noțiunea de mulțime din matematică. Cu toate acestea, ca și multimea din matematică, algoritmii pot fi descriși.

Algoritmul poate fi gândit ca o **succesiune de etape ce se pot aplica mecanic** pentru obținerea unui rezultat.

Exemple din matematică:

- algoritmul lui Euclid pentru aflarea cmmdc a două numere întregi
- algoritmul de înmulțire a două numere
- algoritmul de împărțire a două numere

Generalitatea unui algoritm constă în faptul că el nu operează numai cu numere.

Există algoritmi algebrici dar și algoritmi logici.

O rețetă culinară este în esență tot un algoritm.

Practic nu există domeniu al activității umane în care să nu găsim sectoare ce funcționează algoritmic.

Exemple.

1.**Telefonarea unei persoane** necesită etapele următoare:

- ridicarea receptorului
- așteptarea tonului
- formarea numărului
- transmiterea mesajului
- închiderea telefonului

Aceste etape constituie un algoritm pe care orice persoană îl poate executa mecanic pentru a telefona unei alte persoane.

2.**Gătirea unui fel de mâncare** necesită etapele următoare:

- procurarea ingredientelor necesare (făină, zahăr, ouă, etc.)
- pregătirea acestora
- combinarea lor în anumite proporții
- punerea amestecului în cuptor

https://biblioteca-digitala.ro / https://unibuc.ro

- aşteptarea unui timp pentru fierberea sau coacerea amestecului

Aceste etape constituie un algoritm pe care orice persoana îl poate executa mecanic.

3.**Adunarea a doua fracţii** necesită etapele:

- aducerea fracţiilor la acelaşi numitor
- efectuarea înmulţirilor
- efectuarea adunărilor
- simplificarea fracţiei obţinute, etc.

aceste etape constituie un algoritm pe care orice persoană ce ştie opera cu numere întregi îl poate executa.

Se observă din aceste exemple că toţi algoritmii **se pot descrie în termeni precişi**, astfel încât cel care îi execută să **poată efectua operaţiile mecanic**, adică fără să mai gândească ce are de făcut la un anumit moment.

O analiză sumară a acestor exemple ne conduce la următoarele observaţii:

1.Orice algoritm **porneşte** de la anumite date şi **doreşte** să obţină un anumit rezultat. Astfel:

- In primul exemplu se porneşte de la un anumit **mesaj** şi se doreste transmiterea acestuia unei anumite persoane.
- In exemplul 2 se porneşte de la anumite **produse** şi se doreşte obţinerea mâncării solicitate.
- In exemplul 3 se porneşte de la două **fracţii** şi se doreşte suma lor.

2.Orice algoritm **operează** cu **obiecte** care permit efectuarea anumitor operaţii. Astfel:

- In primul exemplu obiectul este telefonul, iar operaţiile sunt: deschiderea, închiderea telefonului, formarea numărului, etc.
- In exemplul 2 obiectele sunt: farfurii, tăvi, alimente, aragaz, etc. iar operaţiile permise: amestecare, prăjire, fierbere, etc.
- In exemplu 3 obiectele sunt valori numerice iar operaţiile sunt cele descrise de regulile matematice.

3.De cele mai multe ori cel ce elaborează algoritmul este diferit de cel care îl execută. Astfel:

- Pentru primul algoritm a fost necesară inventarea telefonului şi apoi gândirea modului în care se dă uşor un telefon. Deci algoritmul a fost gândit de o mulţime de personalităţi, dar poate fi aplicat (executat) de oricine ştie cifrele 0-9.

https://biblioteca-digitala.ro / https://unibuc.ro

- Pentru algorimul al 2-lea au fost făcute numeroase experimente care au condus în final la o rețeta culinară. Algoritmul poate fi aplicat de orice persoană ce știe să citească rețeta și să folosească un aragaz.
- Pentru algoritmul 3 a fost necesar ca matematica să se dezvolte suficient de mult, cunoscând faptul că mult timp oamenii au lucrat doar cu numere naturale.

Definiție: Numim **algoritm** asociat unui anumit tip de probleme, o metodă generală de rezolvare a problemelor acelui tip, constând dintr-o mulțime finită și ordonată de reguli de calcul care indică **operațiile elementare** necesare rezolvării problemelor precum și ordinea efectuării operatiilor. Regulile de calcul se mai numesc și **pași** ai algoritmului.
Putem spune deci că algoritmul este **o procedură de calcul bine definită** care primește o mulțime de valori ca date de intrare și produce o mulțime de valori ca date de ieșire. Astfel, algoritmul este un **șir de pași** care transformă datele de intrare în date de ieșire.
Algoritmii se caracterizează prin:

1. **Generalitate** (oferă o strategie generală de rezolvare a problemelor de un anumit tip pentru seturi de date de intrare arbitrare).
2. **Finititudine** (permite obținerea datelor de ieșire într-un timp finit și cu un număr finit de pași).
3. **Unicitate** (operația ce succede fiecare pas al algoritmului este determinată în mod unic).
4. **Realizabilitate** (permite numai operații ce se pot executa cu resursele existente).

Pentru ca operațiile unui algoritm să poata fi implementate pe calculator trebuie ca ele să fie:

1. **bine definite** (adică să fie foarte clar ce trebuie executat)
2. **efective** (adică orice persoană ce dispune de un instrument de scris și hârtie să poată efectua operațiie respective într-un timp finit).

Rezolvarea unei probleme necesită parcurgerea următoarelor etape:

1. **Analiza problemei**, adică formularea corectă a ei și aplicarea, adaptarea sau determinarea unei metode de rezolvare.
2. **Programarea**, adică reprezentarea soluției sub o formă ce are în vedere rezolvarea problemei cu ajutorul calculatorului electronic.
3. **Implementarea**, adică aplicarea în practică a soluției.

Persoanele care știu executa cel putin primele doua etape se numesc **programatori** iar cei care știu executa numai etapa a treia se numesc **utiliza-**

https://biblioteca-digitala.ro / https://unibuc.ro

tori. Utilizatorii au nevoie de o pregătire minimă de specialitate în informatică pentru că ei doar utilizează programul şi beneficiază de avantajele lui. Ei sunt consumatori de produse informatice şi beneficiarii acestora. Nu trebuie confundaţi programatorii cu utilizatorii.

Prin **program** înţelegem exprimarea unui algoritm într-un limbaj de programare.

1.1.3 Studiul algoritmilor [14, 18, 37, 69]

Studiul algoritmilor presupune următoarele aspecte:

1. ***Elaborarea algoritmilor.*** Actul de creare a unui algoritm este o artă ce nu va putea fi niciodată pe deplin automatizată. Folosind anumite tehnici de elaborare, intuiţie şi multă eperienţă, se pot concepe algoritmi eficienţi.
2. ***Exprimarea algoritmilor.*** Utilizarea unui anumit stil de programare face ca forma pe care o ia un algoritm într-un program să fie clară şi concisă. Stilul nu este legat neapărat de limbaj, ci mai degrabă de tipul limbajului şi de modul de abordare. Astfel, după anii '80, stilul unanim acceptat este cel de programare structurată, iar în prezent se impune stilul programării orientată pe obiecte.
3. ***Validarea algoritmilor.*** Scris iniţial într-o formă oarecare precisă , nu neapărat codificat într-un limbaj de programare, algoritmul se validează pentru a ne asigura că este corect şi independent de limbajul în care va fi programat.
4. ***Analiza algoritmilor.*** Se defineşte un criteriu de apreciere a unui algoritm care permite obţinerea celui mai bun algoritm dintre mai mulţi care rezolvă o aceiaşi problemă. Criteriul se referă la memoria necesară algoritmului şi la timpul total de calcul necesar efectuării operaţiilor algoritmului.
5. ***Testarea programelor.*** Aceasta constă din depanare şi trasare.

Depanarea este procesul executării unui program pe date de test pentru evidenţierea şi corectarea erorilor (Depanarea evidenţiază doar prezenţa erorilor, nu şi absenţa lor – Dijkstra).
Trasarea este procesul executării unui program corect pe diferite date de test pentru a-i determina timpul de calcul şi memoria necesară.

Rezultatele obţinute după trasare se pot compara cu rezultatele analizei anterioare a algoritmului.

https://biblioteca-digitala.ro / https://unibuc.ro

1.1.4 Reprezentarea algoritmilor [9, 37]

In cele ce urmează ne vom ocupa numai de elaborarea algoritmilor pentru programarea calculatoarelor, adică numai de acei algoritmi care rezolvă probleme ce admit o formulare matematică.

In general, formularea matematică a problemei, algoritmul de rezolvare a problemei şi implementarea acestuia într-un limbaj de programare pentru a fi executat de un calculator sunt trei reprezentări distincte ale aceluiaşi proces de prelucrare.

Descrierea algoritmului este o etapă intermediară între formularea matematică şi codificarea într-un limbaj de programare, necesară mai ales pentru obţinerea unor programe eficiente atât din punct de vedere al memoriei ocupate de program cât şi din punct de vedere al timpului total de execuţie a programului. Printre metodele frecvent folosite în reprezentarea algoritmilor întâlnim:

1. reprezentarea algoritmilor în limbaj natural
2. reprezentarea algoritmilor prin scheme logice
3. reprezentarea algoritmilor în pseudocod
4. reprezentarea algoritmilor prin intermediul tabelelor de decizie.

In **limbaj natural** un algoritm se reprezintă sub forma unui reţetar care pune în evidenţă paşii algoritmului. Acest mod de reprezentare nu necesită pregătiri suplimentare, dar are dezavantajul unei pronunţate neclarităţi pentru probleme complexe.

Prin **schema logică** un algoritm se reprezintă sub o formă grafică în care fiecărui pas al algoritmului i se asociază un simbol numit **bloc**. Blocurile se înlănţuiesc conform logicii algoritmului prin segmente orientate.

În **pseudocod** un algoritm se reprezintă sub o formă clară şi concisă foarte apropiată de codificările în C sau Pascal, formă obţinută prin ignorarea detaliilor de utilizare.

Prin **tabele de decizie** un algoritm se reprezintă scoţând în evidenţă condiţiile şi acţiunile cuprinse în problema de rezolvat şi punând acestea într-un tabel bidimensional cu dublă intrare (intrare condiţii, intrare acţiuni).

Inainte de a detalia fiecare din aceste metode vom prezenta un exemplu pentru care vom folosi toate cele 4 reprezentări.

Exemplu.

Problemă: Pentru $n \in N$, să se calculeze n!.

https://biblioteca-digitala.ro / https://unibuc.ro

Astfel la început f←1
şi apoi pentru i=2,…,n avem f=f*i **(formularea matematică)**
Reprezentarea algoritmului prin cele patru metode cunoscute:

Reprezentarea în limbaj natural:

1. start
2. citeşte n
3. fie f←1
4. dacă n=0, mergi la 10
5. fie i←2 (iniţializarea factorialului)
6. dacă i>n, mergi la 10
7. fie f←f*i
8. fie i←i+1
9. mergi la 6
10. scrie f
11. stop

Reprezentarea prin schemă logică

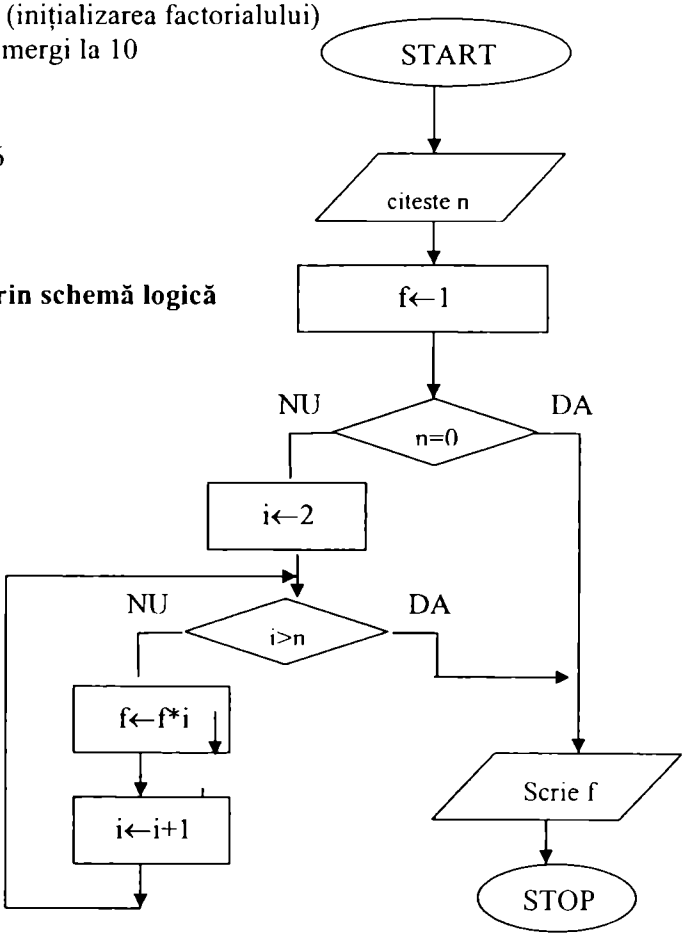


Fig. 1.1

https://biblioteca-digitala.ro / https://unibuc.ro

Reprezentarea în pseudocod

```
procedure factorial
integer n, i, f
read n
f←1
if n≠0 then
   for i=2,n
      f←f*i
   repeat
endif
write f
stop
end
```

Reprezentarea printr-o tabelă de decizie

Condițiile algoritmului: C_1: n=0
C_2: i>n

Acțiunile algoritmului A_1: citește n
A_2: f←1
A_3: i←2
A_4: f←f*i
A_5: i←i+1
A_6: scrie f

reguli de decizie

	R_1	R_2	R_3	R_4
C_1	D	D	N	N
C_2	D	N	D	N
A_1	*	*	*	*
A_2	*	*	*	*
A_3			*	*
A_4				*
A_5				*
A_6	*	*	*	

Definim programele de acțiuni:

$P_1=\{A_1,A_2,A_6\}$, $P_2=\{A_1,A_2, A_3, A_6\}$, $P_3=\{A_1,...,A_6\}$ și observând că P_1 se execută când C_1 este adevărată indiferent cum este C_2, tabela se restrânge la următoarea

https://biblioteca-digitala.ro / https://unibuc.ro

	R_1'	R_2'	R_3'
C_1	D	N	N
C_2	-	D	N
P	P_1	P_2	P_3

unde intrarea "-" se citeşte **indiferent**.
O posibilă implementare a algoritmului în TC, C++ , Pascal şi Qbasic ar fi:

TC:
```
#include<stdio.h>
void main()
{
int i,n;
double f=1.0;
scanf ("%d",&n);
if (n!=0)
        for (i=2; i<=n; i++) f*=i;
printf("Valoarea factorialului este:  %lf",f);
}
```

C++:
```
#include<iostream.h>
void main()
{
int i,n;
double f=1.0;
cin>>n;
if (n!=0)
        for (i=2; i<=n; i++) f*=i;
cout <<"valoarea lui "<<n<<"! este:"<<f<< endl;
}
```

Pascal:
```
Program factorial;
var
i,n:integer;
f:real;
begin
        f:=1.0
        readln (n);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
            if n< >o then
                    for i:=2 to n do
                            f:=f*i;
            writeln ('Valoarea factorialului este: ',f:15:0);
        end.

Qbasic: input "n=";n
        f=1
        if n=0 then goto a
        for i=2 to n
                f=f*i
        next i
    a:  print n;"!=";f
        stop
```

1.2 Reprezentarea algoritmilor prin scheme logice [8, 9, 37, 55]

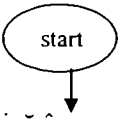
Acest mod de reprezentare a algoritmilor este un mod ilustrativ, util mai ales în faza învățării abecedarului programării şi a familiarizării cu logica algoritmică şi cu structurile de control implicate în această logică.
Definiție: Prin ***schemă logică*** asociată unei metode de rezolvare a unei probleme înțelegem o reprezentare grafică a algoritmului de rezolvare, reprezentare în care fiecărui pas al algoritmului i se asociază un simbol numit ***bloc***. Blocurile sunt conectate între ele prin segmente orientate conform logicii algoritmilui.
Astfel, schema logică este o **reprezentare a fluxului** de prelucrare a datelor algoritmului.
Există trei categorii de scheme logice:

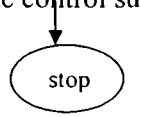
- **scheme de program**, sunt scheme logice ce descriu procesele de prelucrare a datelor pornind de la datele de intrare până la obținerea rezultatelor finale.
- **scheme de detaliere,** sunt scheme logice ce descriu procese complexe la nivel de detaliu, procese ce apar o singură dată în program. Aceste scheme se folosesc şi la modularizarea programelor.
- **scheme de subprogram** sunt scheme logice ce descriu procese repetabile în program.

https://biblioteca-digitala.ro / https://unibuc.ro

Într-o schemă logică blocurile sunt împărțite în clase în funcție de modul lor de utilizare. Avem astfel:

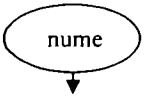
1. **Blocuri de control** care delimitează începutul și sfârșitul unei scheme.
   - pentru schemele de program, blocurile de control sunt:

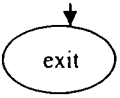




   care indică începutul programului respectiv sfârșitul programului

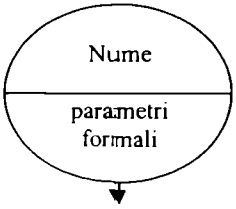
   - pentru schemele de detaliere, blocurile de control sunt:



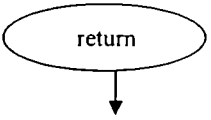


   care indică începutul și sfârșitul schemei de detaliere identificată prin **nume**

   - pentru schemele de subprogram, blocurile de control sunt:

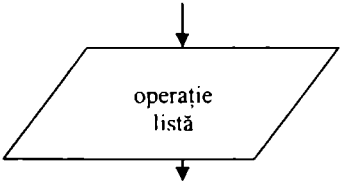


și



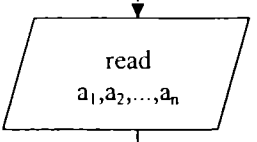
   care permit atât identificarea subprogramului prin nume, cât și identificarea elementelor ce permit comunicarea între subprogram și programul apelant prin parametri formali și respectiv sfârșitul schemei care are sensul de revenire la programul apelant.

2.Blocuri de intrare/ieșire prin care se pun în evidență datele de intrare și datele de ieșire ale algoritmului. Acestea folosesc imaginea comună de forma

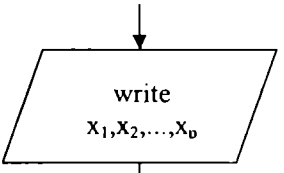


https://biblioteca-digitala.ro / https://unibuc.ro

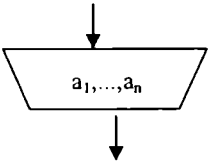
unde operație ∈ {read, write}, iar listă indică elementele implicate în operația realizată.
Astfel,

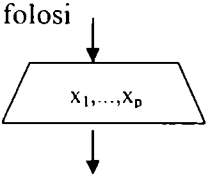


precizează variabilele a_1, a_2, ..., a_n în ale căror locații de memorie asociate se citesc datele de intrare ale algorimului.
Blocul



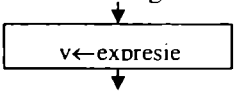
conține constante, variabile, expresii sau șiruri de caractere ce urmează a fi afișate ca rezultate ale prelucrărilor algoritmului.
Pentru citire se mai poate folosi blocul și pentru scriere se mai poate folosi





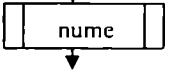
3.Blocurile de prelucrare indică operațiile elementare de calcul sau procesele necesare în prelucrarea informațiilor. Aici găsim:

- **Blocul de atribuire:** are forma



și folosește operatorul de atribuire "←". Interpretarea blocului este următoarea: "se evaluează expresia din dreapta simbolului ← și valoarea obținută se atribuie variabilei v".

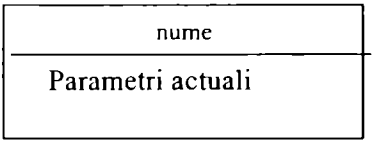
- **Blocul de blocuri** are formatul:



care specifică procesele de prelucrare ce se constituie ulterior într-un subproces descris printr-o schemă de detaliere. Aici "nume" este identificatorul schemei de detaliere folosită. Blocul se folosește și pentru a indica un proces de tip subprogram ce se găsește predefinit în limbajul folosit.

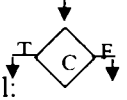
https://biblioteca-digitala.ro / https://unibuc.ro

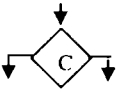
- **Blocul de apel** procedură cu formatul:



şi folosit pentru apelarea subprogramelor. Aici, **nume** identifică subprogramul apelat, iar **parametri actuali** sunt datele pentru care se realizează procesul.

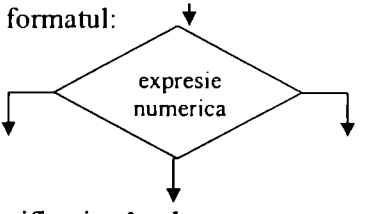
4.Blocurile de decizie pun în evidenţă punctele de ramificare ale algoritmului. Aici întâlnim:

- **blocul de testare logică** cu formatul: 

unde C este o condiţie care poate fi adevărată sau falsă. Dacă C este adevărată (C=True), algoritmul continuă pe ramura corespunzătoare lui T, altfel, continuă pe ramura corespunzătoare lui F.
Putem folosi formatul: 

cu convenţia că ieşirile din dreapta corespund totdeauna îndeplinirii condiţiei C(C=T) iar cele din stânga neîndeplinirii ei (C=F).

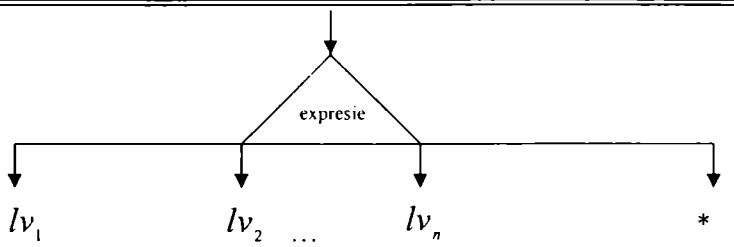
- **blocul de testare algebrică** cu formatul:



unde ieşirile din bloc au semnificaţia: în dreapta avem >0, în stânga <0, şi jos = 0. Se continuă algoritmul după cum expresia din bloc este: > 0, < 0 respectiv = 0.

- **blocul de testare multiplă** cu formatul:

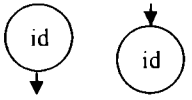
https://biblioteca-digitala.ro / https://unibuc.ro



Aici, **expresie** numită **expresie ordinară**, ia valori într-o mulțime finită, iar $lv_1,\ldots,lv_n$ sunt submulțimi disjuncte două câte două ale acelei mulțimi. Arcul "*" desemnează valorile pe care le poate lua expresia și care nu sunt cuprinse în $\bigcup_{i=1}^{n} lv_i$

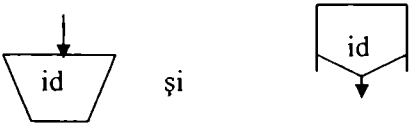
5.Blocurile de conectare nu au o utilitate efectivă în prezentarea prelucrărilor, dar sunt necesare în prezentare pentru aspect, pentru a evita intersectarea arcelor și pentru a desfășura scheme mai ample pe mai multe pagini. Aici întâlnim:

- **Blocuri de conectare în pagină** cu formatul



care evită intersectarea arcelor și folosirea de arce foarte lungi. Se folosesc în pereche, primul fiind sursa conectării, iar al doilea destinația ei, și pentru a identifica o pereche, se folosește în interiorul lor un același **simbol** sau **identificator**.

- **Blocuri de conectare între pagini** cu formatul:



care permit desfășurarea unei scheme pe mai multe pagini. Se folosesc în pereche și o aceeași pereche utilizează același simbol.
O schemă logică se obține înlănțuind blocuri din cele de mai sus prin intermediul arcelor (segmentelor orientate) și respectând următoarele reguli:

https://biblioteca-digitala.ro / https://unibuc.ro

1. Există un singur bloc delimitator de început
2. Există cel puțin un bloc delimitator de sfârșit
3. Extremitatea inițială a fiecărui bloc (intrarea în bloc) trebuie să fie extremitate finală (ieșirea din bloc), a cel puțin unui alt bloc (excepție blocul delimitator de început).
4. Fiecare extremitate terminală a fiecărui bloc trebuie să fie extremitate inițială a unui singur alt bloc (excepție, blocul delimitator de sfârșit).

Exemplu de schemă logică - Sortarea prin inserție
Metoda se aseamănă cu modalitatea în care se sortează un pachet de cărți obișnuite. Operațiile acestei sortări sunt:

- Punem pachetului de cărți pe masă cu fața în jos
- Eliberăm mâna stângă
- Luăm prima carte din pachet și o punem în mâna stângă
- Luăm apoi câte o carte de pe masă si o inserăm în poziția corectă în mâna stângă (poziția corectă a unei cărți se obține comparând cartea luată cu fiecare din cărțile aflate deja în mâna stângă, de la dreapta la stânga).

Fie a(n) un vector cu n≥2 componente reale. Ne propunem să ordonăm crescător elementele acestui vector folosind sortarea prin inserție ca în exemplul următor:
Exemplu: a=(5 2 4 6 1 3), n=6
Parcurgem vectorul de la poziția a 2-a până la poziția n.
Elementul a_2=2 fiind mai mic decât 5 îl vom trece în stânga lui 5ș obținem a=(2 5 4 6 1 3).
Elementul de pe poziția a 3-a, a_3=4 trebuie introdus între 2 și 5ș obținem a=(2 4 5 6 1 3).
Elementul a_4=6 fiind mai mare decât toate elementele din stânga sa, rămâne pe loc. Obținem a=(2 4 5 6 1 3).
Elementul a_5=1 fiind mai mic toate elementele din stânga sa trebuie deplasat la început. Obținem a=(1 2 4 5 6 3).
Elementul a_6=3 tebuie deplasat între 2 și 4. Obținem a=(1 2 3 4 5 6), un vector sortat.
In acest exemplu, elementul de pe poziția i corespunde cărții care trebuie inserată în mâna stângă, care la rândul ei corespunde elementelor vecto-

https://biblioteca-digitala.ro / https://unibuc.ro

rului a(1...i-1). Elementele a(i+1..n) corespund pachetului de cărți aflat pe masă.
Indicele i se deplasează de la stânga la dreapta în interiorul vectorului parcurgând valorile 2...n .
La fiecare iterație elementil a_i se mută într-o locație s, $s \leftarrow a_i$, care corespunde mâinii drepte. Plecând de la poziția i-1, elementele sunt succesiv deplasate câte o poziție spre dreapta până ce este găsită poziția corectă pentru a_i . La acest moment elementul a_i este inserat pe poziția lui corectă (cartea din mâna dreaptă este pusă la poziția corectă în mâna stângă).
Schema logică a acestui algoritm este prezentată in figura 1.2.

1.3 Structuri elementare ce stau la baza programării structurate

În descrierea algoritmilor apar frecvent câteva **structuri de control** numite structuri elementare, structuri fundamentale sau structuri logice reprezentabile prin mai multe blocuri conectate şi caracterizate prin aceea că au o singură intrare şi o singură ieşire.

Bazele realizării şi impunerii unor reguli stricte în realizarea schemelor logice şi a programelor au fost puse după 1970 şi s-au numit **principiile programării structurate**. Aceste reguli au dus la o dezvoltare fără precedent a limbajelor de programare.
Programarea realizată pe baza structurilor fundamentale se numeşte programare structurată şi se caracterizează prin:

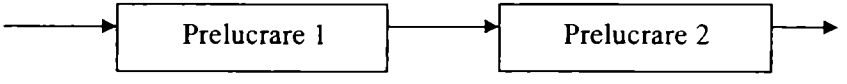
- claritate în redactarea programelor
- depanare uşoară a programelor
- reducerea posibilității de a comite erori
- universalitatea programelor

Structurile de control ce stau la baza programării structurate sunt

1. structura secvențială
2. structuri alternative
3. structuri repetitive

Structura secvențială

Are formatul:



https://biblioteca-digitala.ro / https://unibuc.ro

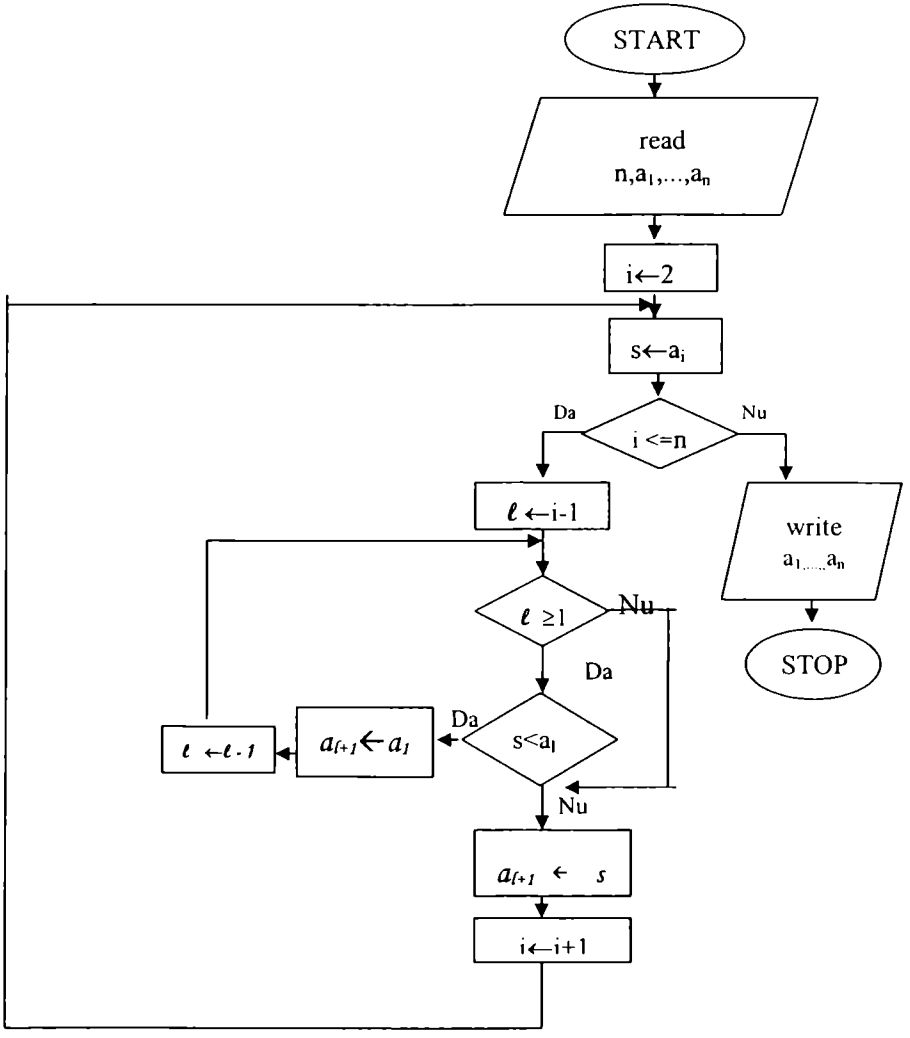
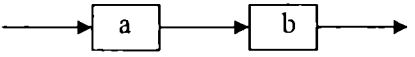


Fig. 1.2

https://biblioteca-digitala.ro / https://unibuc.ro

şi reprezintă o înlănţuire în secvenţă a două prelucrări, a doua urmând să înceapă după terminarea primei prelucrări. Această structură se foloseşte pentru a indica ordinea de desfăşurare a prelucrărilor.

În cele ce urmează, se va nota cu Π (a,b) unde a respectiv b sunt cele două prelucrări:

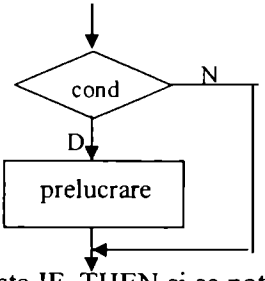


Structuri alternative

Există trei tipuri de structuri alternative care selectează variantele de prelucrare în funcţie de îndeplinirea sau nu a unor condiţii logice. Acestea sunt:

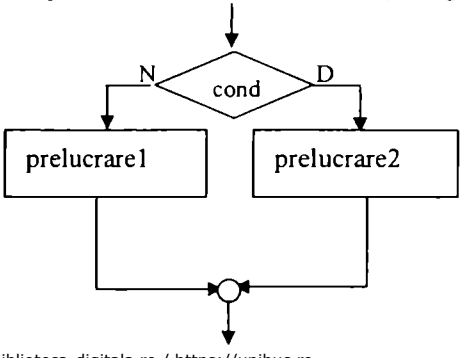
- structura pseudoalternativă
- structura alternativă simplă
- structura alternativă multiplă.

a. ***Structura pseudoalternativă*** are formatul:



Se numeşte IF..THEN şi se notează Λ(C,a) unde C este condiţia din blocul de decizie, iar a este blocul de prelucrare executat atunci când condiţia este indeplinită.

b. ***Structura alternativă simplă*** specifică două procese paralele dintre care se realizează doar acela care corespunde îndeplinirii sau nu a unei condiţii logice. Are formatul:



https://biblioteca-digitala.ro / https://unibuc.ro

Se numeşte IF-THEN-ELSE şi se notează cu $\Delta(C,a,b)$, unde C este condiţia evaluata, iar a şi b sunt cele două prelucrări.

c. ***Structura alternativă multiplă*** are formatul:

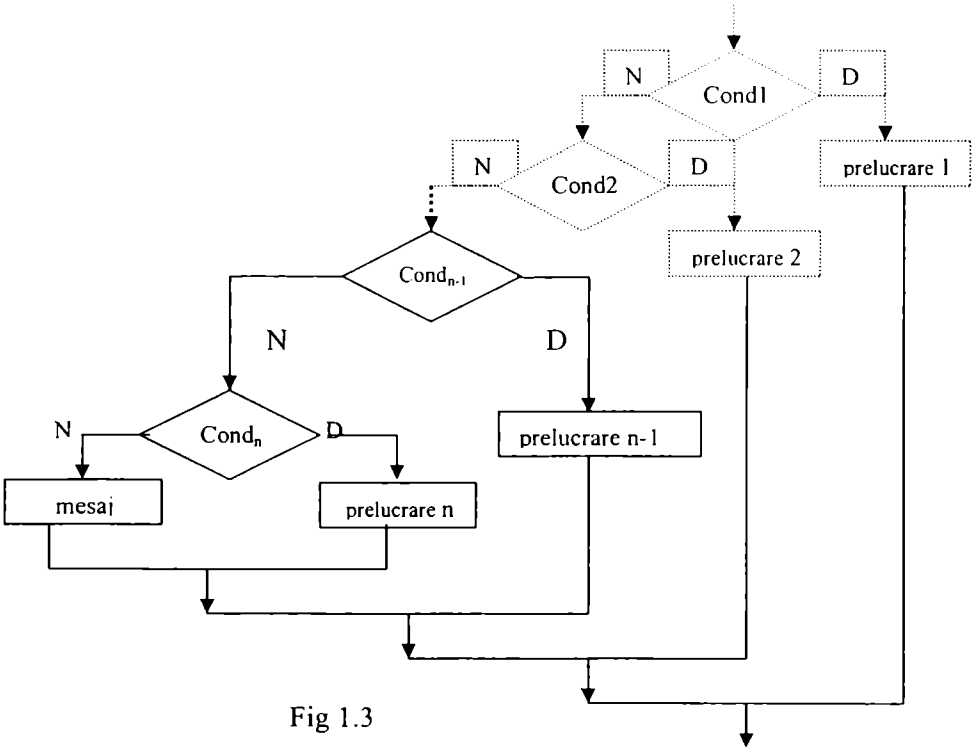


Fig 1.3

Se numeşte CASE şi se notează $\Gamma(C,A)$, unde $C = \{C_1, C_2,...,C_n\}$, $A=\{a_1,...,a_n, \text{mesaj}\}$ iar $C_i=cond_i$, $a_i=prelucrare_i$, i=1,...,n

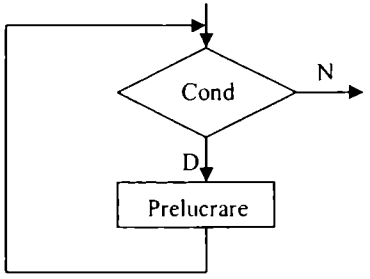
Structuri repetitive

Există trei structuri repetitive care reprezintă modalitatea prin care se exprimă repetarea unui proces de calcul. Acestea sunt:

- Structura repetitivă condiţionată anterior (cu test iniţial)
- Structura repetitivă condiţionată posterior (cu test final)
- Structura repetitivă cu contor (numărător)

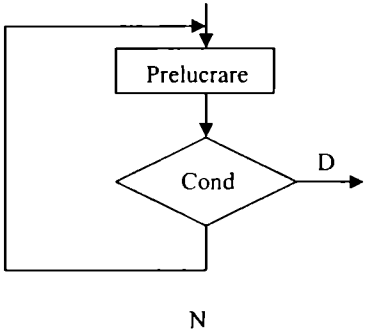
https://biblioteca-digitala.ro / https://unibuc.ro

a. *Structura repetitivă cu test inițial* are formatul:



Se numeşte WHILE-DO şi se notează prin Ω(C,a), unde C=Cond, iar a=Prelucrare. În acest caz, **a** se execută numai când C este îndeplinită. Dacă de la început C este falsă, **a** nu se execută niciodată.

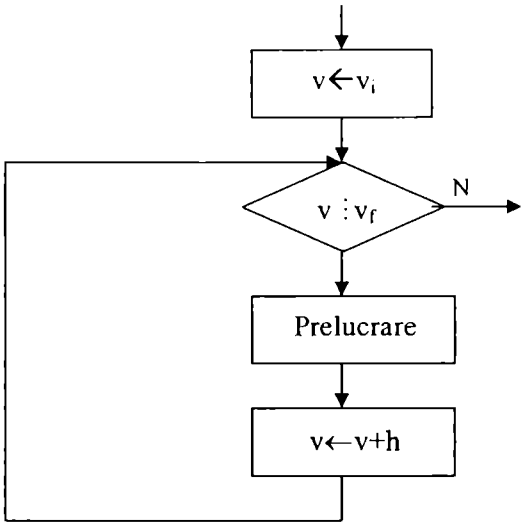
b. *Structura repetitivă cu test final* are formatul:



Se numeşte DO-UNTIL şi se notează Φ(C,a) unde C=Cond şi a=Prelucrare. În acest caz, **a** se execută până ce condiția C este îndeplinită Dacă de la început C este îndeplinită, **a** se execută totuşi odată.
Se observă usor că cele două structuri sunt echivalente.

https://biblioteca-digitala.ro / https://unibuc.ro

c. *Structura repetitivă cu contor* (numarator) este un caz particular al lui WHILE-DO, corespunzător situației în care se cunoaşte apriori numărul de parcurgeri ale structurii repetitive. Are formatul:



şi se numeşte FOR, unde:

v = variabilă de control a ciclării sau contorul
v_i = valoarea inițială
v_f = valoarea finală
h = pas de incrementare, $h \neq 0$

v ia valori între v_i şi v_f cu pasul h.

Aici $\vdots = \begin{cases} \leq, h > 0 \\ \geq, h < 0 \end{cases}$

Dacă $h>0 \Rightarrow v_i \leq v_f$, iar dacă $h<0 \Rightarrow v_i \geq v_f$.

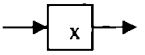
Numărul de parcurgeri ale structurii repetitive este: $1 + \left[\left|\frac{v_f - v_i}{h}\right|\right]$

Observație. Dacă pentru $h > 0$ avem $v_i > v_f$ sau dacă pentru $h < 0$ avem $v_i < v_f$, atunci numărul de parcurgeri este 0.

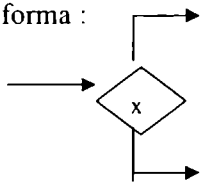
Pentru cazul $h > 0$, structura se mai numeşte FORTO, iar pentru cazul $h < 0$, se mai numeşte FORDOWNTO.

https://biblioteca-digitala.ro / https://unibuc.ro

1.4 Programare structurată [52, 55]

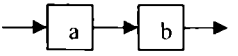
Blocurile unei scheme logice se pot împărți în două categorii mari:
- *blocuri funcționale* de forma: 

ce reprezintă operații elementare asupra elementului x. Aici găsim blocurile de intrare/ieșire și blocurile de prelucrare.
- *blocuri predicative* de forma :

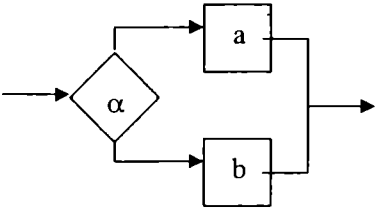


care decid asupra următoarei operații ce trebuie executată în funcție de valoarea predicatului x. Aici găsim blocurile de decizie.

Considerăm structuri de bază ale programării structurate următoarele trei structuri

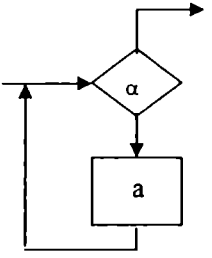
- *secvența,* notată $\pi(a,b)$ și care are forma: 

- *selecția*, notată $\Delta(\alpha,a,b)$ și care are forma:



Ea se numește IF-THEN-ELSE

- *iterația,* notată $\Omega(\alpha,a)$ și care are forma:

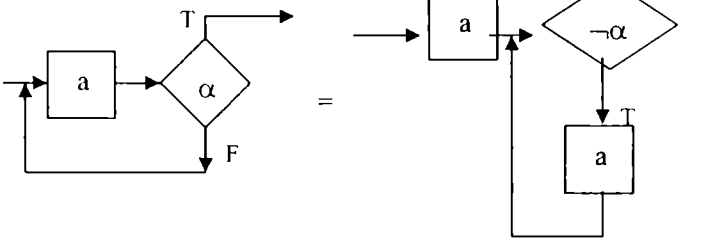


Ea se numește WHILE-DO.

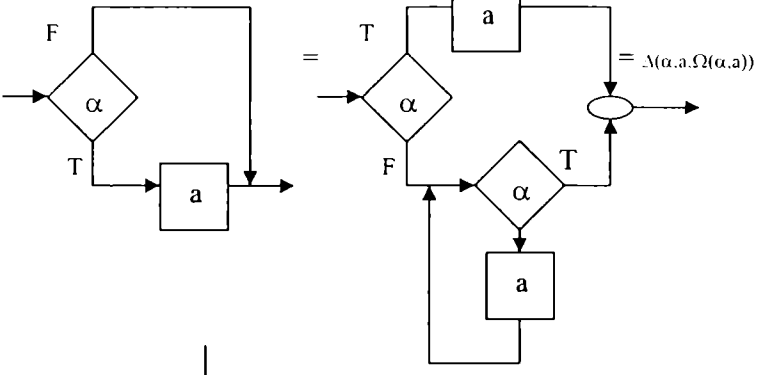
https://biblioteca-digitala.ro / https://unibuc.ro

Observăm că DO-UNTIL şi IF-THEN se pot exprima cu ajutorul celor două blocuri astfel:

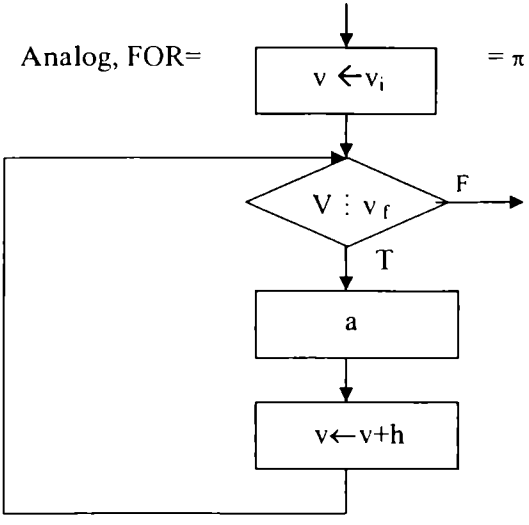
$\Phi(\alpha,a)=\pi(a,\Omega(\neg\alpha,a))$ deoarece:



$\Lambda(\alpha,a)=$



Analog, FOR=



$= \pi\,(v \leftarrow v_i,\ \Omega\,(v \mid v_f,\ \pi\,(a, v \leftarrow v+h)))$

https://biblioteca-digitala.ro / https://unibuc.ro

$\Gamma(C,A) = \Delta(c_1, a_1, \Delta(c_2, a_2, \Delta(c3a3,...., \Delta(c_n, a_n, mesaj))....)$

Definiție: Spunem că schema logică Σ admite **descompunere tare** $\Leftrightarrow \Sigma$ se poate exprima numai în funcție de structurile de bază π, Ω, Δ. , adică $\Sigma=\Sigma(\pi, \Omega, \Delta)$.

Exemple de scheme ce admit descompunere tare sunt DO...UNTIL, IF...THEN, CASE si FOR.

Există şi scheme logice care nu admit descompunere tare, aşa cum ar fi următoarea schema:

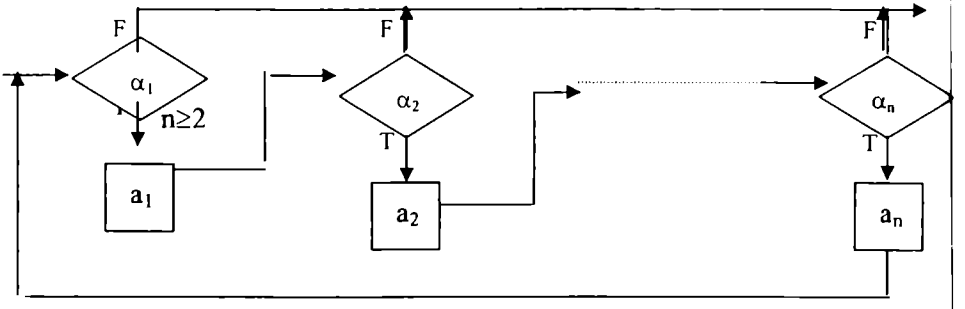


Fig. 1.4

Observăm că pentru n=1, această schemă este $\Omega(\alpha_1, a_1)$ care admite o descompunere tare.

Pentru a exprima şi scheme de tipul celei de mai sus cu ajutorul structurilor π, Ω si Δ, vom mai introduce trei blocuri funcționale T, F, K şi un bloc predicativ ω definite astfel:

- T este un bloc ce transformă elementul x în perechea (t,x), unde t=true, T(x)=(t,x).
- F este un bloc ce transformă elementul x în perechea (f,x) unde f=false, F(x)=(f,x)
- K este un bloc ce alege dintr-o pereche ordonată a doua componentă a sa. K(v,x)=x, v$\in${t,f} .
- ω este dat prin $\omega(v,x)=t \Leftrightarrow v=t$.

Definiție: Spunem că schema logică Σ are **formă normală** (este sub forma normala) $\Leftrightarrow \Sigma$ se poate exprima prin blocurile π, Ω, Δ, T, F, K si ω, adică $\Sigma=\Sigma(\pi,\Omega,\Delta,T,F,K,\omega)$.

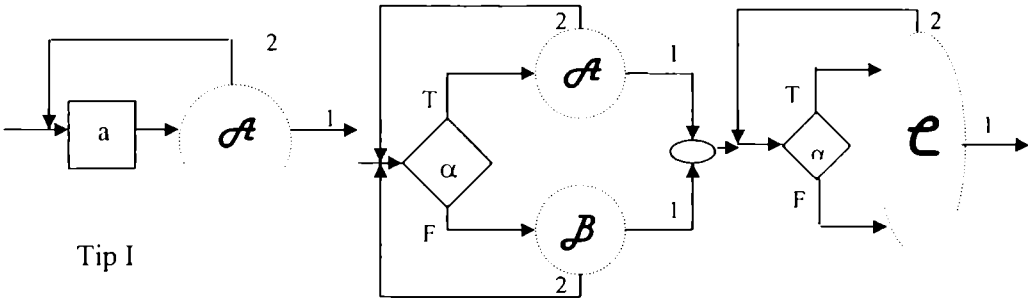
Teorema de structură a lui Böhm-Jacopini

Orice schemă logică se poate aduce la forma normală (se poate normaliza).

https://biblioteca-digitala.ro / https://unibuc.ro

Demonstrație:

Pentru demonstrație vom observa mai întâi că orice schemă logică poate fi de una din următoarele trei tipuri:

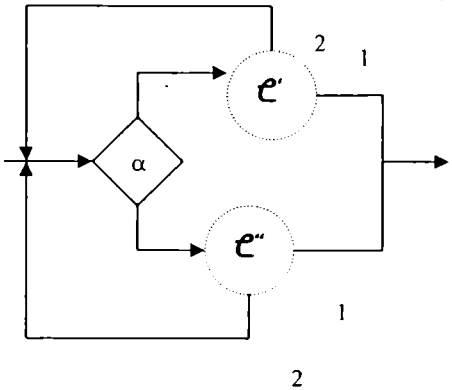


Tip I Tip 2 Tip 3

Fig. 1.5

unde A, B, C sunt părți de schemă logică construite în mod arbitrar. Ramurile 1 și 2 pot să nu fie totdeauna prezente, dar din A, B, sau C trebuie să pornească cel puțin o săgeată.

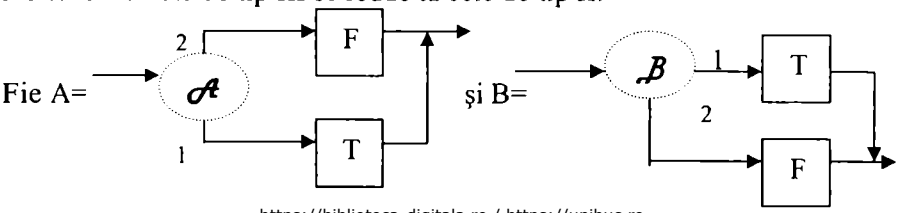
Se mai observă că o schema de tip III poate fi pusă sub forma:



unde C' este o parte a lui C accesibilă de la partea superioară a lui C iar C '' este o parte a lui C acesibilă de la partea inferioară a sa și în plus, C = C ' ∪ C ''.

Fig. 1.6

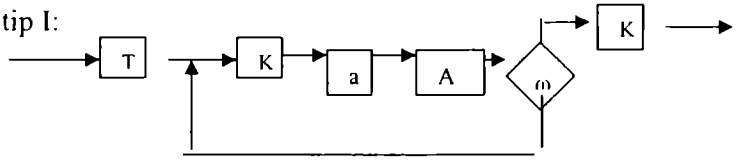
Astfel schemele de tip III se reduc la cele de tip II.



https://biblioteca-digitala.ro / https://unibuc.ro

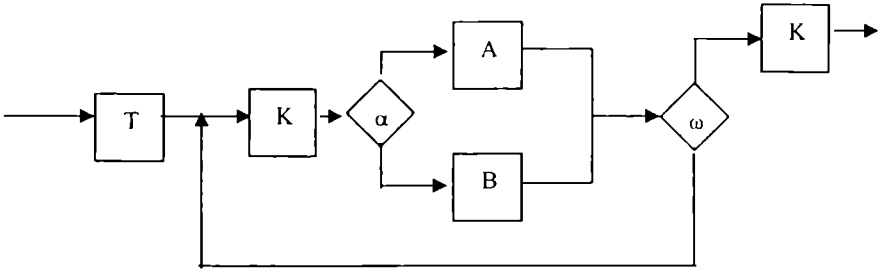
în care una din ramurile 1 sau 2 poate să nu existe.
Cu aceste notații, schemele de tip I şi II se normalizează astfel:

tip I:



$=\pi(T,\pi(\Phi(\omega,\pi(K,\pi(a,A))),K)$.

tip II:



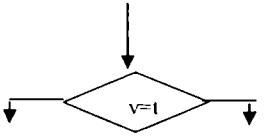
$= \pi(T,\pi(\Phi(\omega,\pi(K,\Delta(\alpha,A,B))),K))$

Presupunând acum, printr-un raționament asemănător inducției că A şi B se pot reprezenta prin structurile π, Ω, Δ, T, F, K şi ω, din normalizările anterioare, rezultă că schemele de tip I, II si III se pot reprezenta prin aceste blocuri, deoarece am văzut mai sus că structura Φ ce intră în aceste normalizări se exprimă prin π şi Ω. Astfel, orice schemă logică se poate normaliza.
În aplicații practice, teorema lui Böhm şi Jacopini a condus la o tehnică de structurare numită **tehnica variabilei booleene**, tehnică ce se aplică în special pentru scheme ce conțin cicluri. Ea constă în:

- Fiecărui ciclu i se asociază o variabilă booleană v care se inițializează prin v←t, în punctul de intrare în ciclu, se transformă ciclul într-o structură repetitivă de tip WHILE-DO cu condiția v=t, iar în cadrul

https://biblioteca-digitala.ro / https://unibuc.ro

structurii repetitive, v se actualizează astfel încât să se poată continua execuția sau să se iasă din ciclu.
În acest caz, blocul T este v←t, blocul F este v←f, iar blocul predicativ ω este



Nu intervine blocul K

Sunt situații în care o singură variabilă booleană permite structurarea mai multor cicluri; de exemplu când ciclurile au același punct de intrare.

Exemplu: Să se normalizeze următoarea schemă:

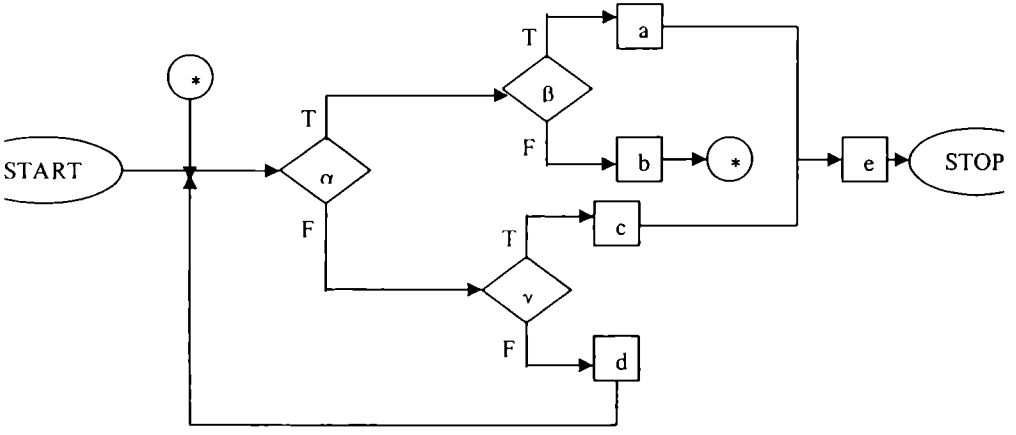


Fig. 1.7

Obținem:

https://biblioteca-digitala.ro / https://unibuc.ro

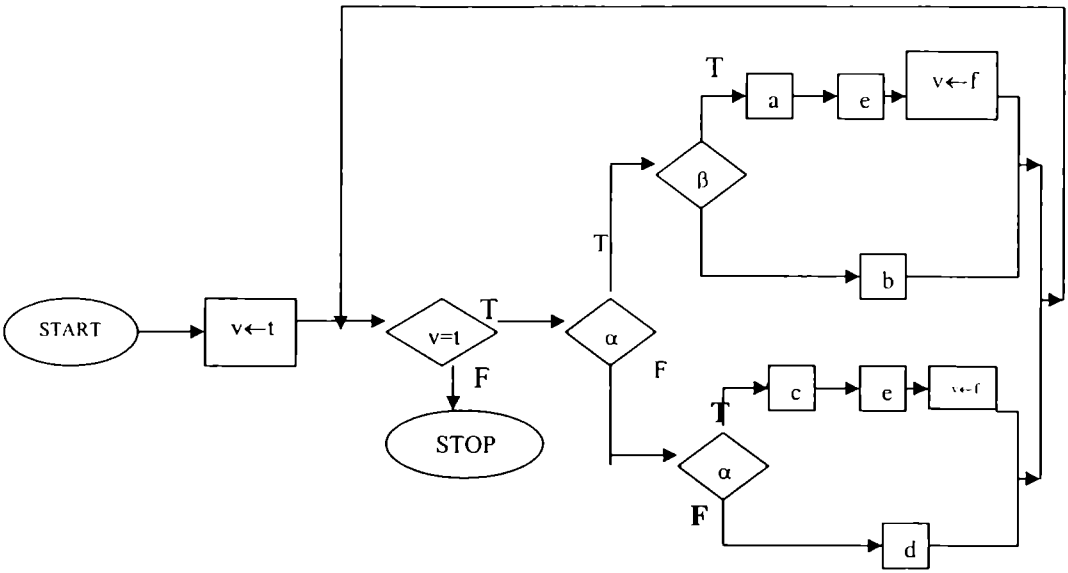


Fig. 1.8

Definiție: Numim ***formulă de structură*** a unei scheme Σ, exprimarea liniară a lui Σ prin structurile de bază π,Ω,Δ.

În exemplul de mai sus avem formula de structură:

$$\Sigma = \pi(v\leftarrow t,\Omega(v=t,\Delta(\alpha,\Delta(\beta,\pi(a,\pi(e,v\leftarrow f)),b),\Delta(\gamma,\pi(c,\pi(e,v\leftarrow f)),d))))$$

1.5 Conceptul NL (Nested Logic) de proiectare şi Programare [55]

NL este un concept prin care o structură de date sau o structura de prelucrare se reprezintă printr-un număr de 3 blocuri fundamentale sub forma unei diagrame arborescente de structură.
Blocurile fundamentale sunt:

- *Secvența*: un grup de instrucțiuni descompus în mod arbitrar în două subgrupuri "a" şi "b" care sunt executate în ordinea scrierii

https://biblioteca-digitala.ro / https://unibuc.ro

lor. Instrucțiunile fiecărui subgrup se execută în ordine secvențială. Acest bloc corespunde lui π(a,b) din programarea structurată.

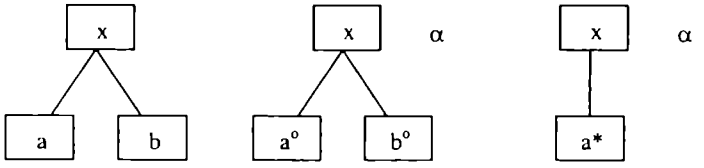
- *Selecția*: un grup de instrucțiuni descompus în funcție de valorile logice ale unei condiții α în două subgrupuri "a" și "b", instrucțiunile fiecărui subgrup executându-se secvențial în ordinea scrierii lor. Subgrupul "a" se execută când α=true iar subgrupul "b" când α=false. Blocul corespunde lui Δ(α,a,b) din programarea structurată.
- *Iterația*: un grup de instrucțiuni "a" ce se execută de un număr de ori în funcție de valorile logice ale unei condiții α. Blocul corespunde lui Ω(α,a) din programarea structurată.

Aceste blocuri au un singur punct de intrare, un singur punct de ieșire și blocul iterativ începe cu testarea condiției de ieșire din ciclu.

Definiție: Se numește ***diagramă de structură NL*** o reprezentare grafică ierarhizată a unei structuri de date sau a unei structuri de prelucrare pe nivele de detaliere sub forma unui arbore binar construit astfel:

- nivelele se parcurg de sus în jos după regula RSD (preordine)
- fiecare nivel este o detaliere a nivelului precedent
- blocurile de pe același nivel se parcurg de la stânga la dreapta

Diagramele de structură ale blocurilor fundamentale sunt:



π(a,b) Δ(α,a,b) Ω(α,a)

unde: "o" indică alegerea unui bloc sau a celuilalt în funcție de valorile condiției α,

"*" indică repetarea de mai multe ori a execuției lui "a" în funcție de valorile condiției α.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu de diagramă de structură NL:

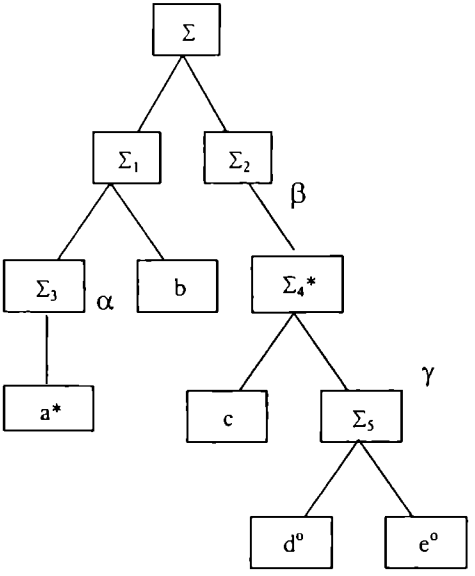


Fig. 1.9

Citirea unei diagrame de structură foloseşte următoarele cuvinte cheie:

- "**constă din**" folosit pentru trecerea de la un nivel la nivelul imediat inferior
- "**urmat de**" pentru trecerea de la un bloc la altul pe acelaşi nivel sau pentru trecerea de la un bloc de pe un nivel la următorul bloc de pe un nivel superior conform parcurgerii RSD a diagramei.
- "**sau**" folosit la selecţie pentru a citi simbolul "o"
- "**mai multe**" folosit la iteraţie pentru a citi simbolul "*".

Cu acestea, diagramele de structură ale blocurilor fundamentale se citesc astfel:

- secvenţa: x constă din "a" urmat de "b"
- selecţia: x constă din sau "a" sau "b"
- iteraţia: x constă din mai multe "a".

Diagrama din exemplul de mai sus se citeşte astfel:

https://biblioteca-digitala.ro / https://unibuc.ro

Σ constă din Σ_1 constă din Σ_3 constă din mai multe "a" urmat de "b" urmat de Σ_2 constă din mai multe Σ_4 constă din "c" urmat de Σ_5 constă din sau "d" sau "e".
Conceptul NL conduce la un limbaj care permite o reprezentare clară şi concisă a unei structuri de date sau de prelucrare, numit **limbaj NL**.

Instrucţiunile de bază ale limbajului NL:

1. ***Instrucţiunea de prelucrare***: cu formatul do x, unde "x" este un bloc de instructiuni.
2. ***Instrucţiunea secvenţială*** cu formatul:
   et seq
   .....
   et end, unde "et" este o etichetă comună a lui seq şi end
3. ***Instrucţiunea selectivă*** cu formatul:
   et if α then
   .....
   et else
   .....
   et end, unde "et" este o etichetă comună a lui if, else şi end.
4. ***Instrucţiunea iterativă*** cu formatul:
   et loop until $\neg \alpha$
   .....
   et end, unde "et" este o etichetă comună a lui loop şi end
5. ***Instrucţiunea CASE***, generalizează selecţia şi are forma:
   et if α_1 then
   .....
   et else if α_2 then
   .....
   et else if α_i then
   .....
   et else if α_{i+1} then
   .....
   et end.

Dacă α_i este îndeplinită se execută blocurile dintre *if* şi *else if* pentru i=1 sau dintre *else if* α_i şi *else if* α_{i+1} pentru i $\geq$2, după care se trece automat la end.

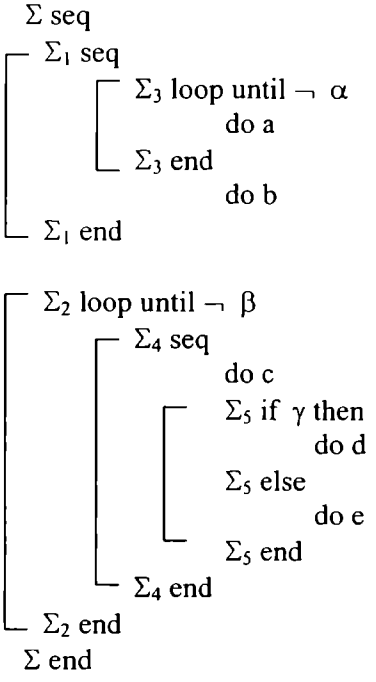
Observaţii

1. Etichetele instrucţiunilor NL sunt numele nodurilor neterminale din diagrama de structură NL.

https://biblioteca-digitala.ro / https://unibuc.ro

2. Instructiunea imperativă **do** foloseşte numele blocurilor terminale ale diagramei de structură NL.

Exemplu: Programul în limbaj NL asociat diagramei de structură din exemplul de mai sus este :



Teorema de structură NL (teorema lui Jackson)
Pentru orice schemă logică Σ, asociată unui proces de prelucrare, există cel puţin un operator T care transformă Σ într-o schemă logică T(Σ) de tip NL echivalentă cu Σ.

Demonstraţia este constructivă. Ea arată cum se construieşte un operator T care să satisfacă cerinţele teoremei.
T se obţine conform următorului algoritm:

https://biblioteca-digitala.ro / https://unibuc.ro

1. Se numerotează toate blocurile lui Σ, excepție START, începând cu 1 pentru blocul de intrare (cel imediat de după **start**), într-o ordine arbitrară și terminând cu **stop**, căruia i se asociază cifra "0".

2. Se definește o variabilă I numită **variabilă de stare** care ia valorile 0,1,...,N (N=numărul de blocuri fără **start** și **stop** ale lui Σ) și care se inițializează cu 1 în blocul de intrare al schemei NL.
3. Se definește un bloc iterativ cu condiția I≠0.

4. Se definește un bloc CASE cu N ramuri, câte o ramură pentru fiecare valoare diferită de 0 a lui I.

5. Fiecare ramură din CASE conține blocul corespunzător din Σ urmat de un bloc de actualizare a lui I care să permită execuția blocului următor din Σ.

Astfel, diagrama T(Σ) este de tip NL și în plus realizează aceleași prelucrări ca și Σ, adică este echivalentă cu ea.

Exemplu:

Fie Σ:

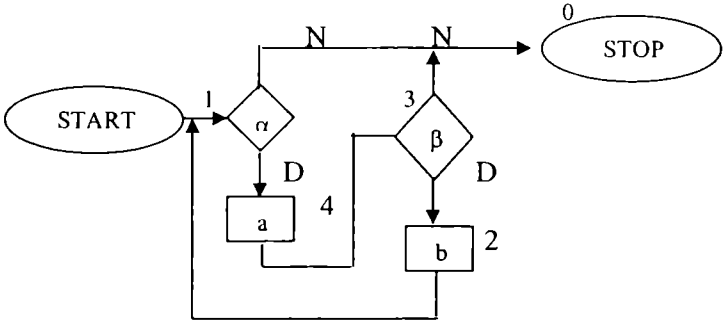


Fig. 1.10

Atunci T(Σ):

https://biblioteca-digitala.ro / https://unibuc.ro

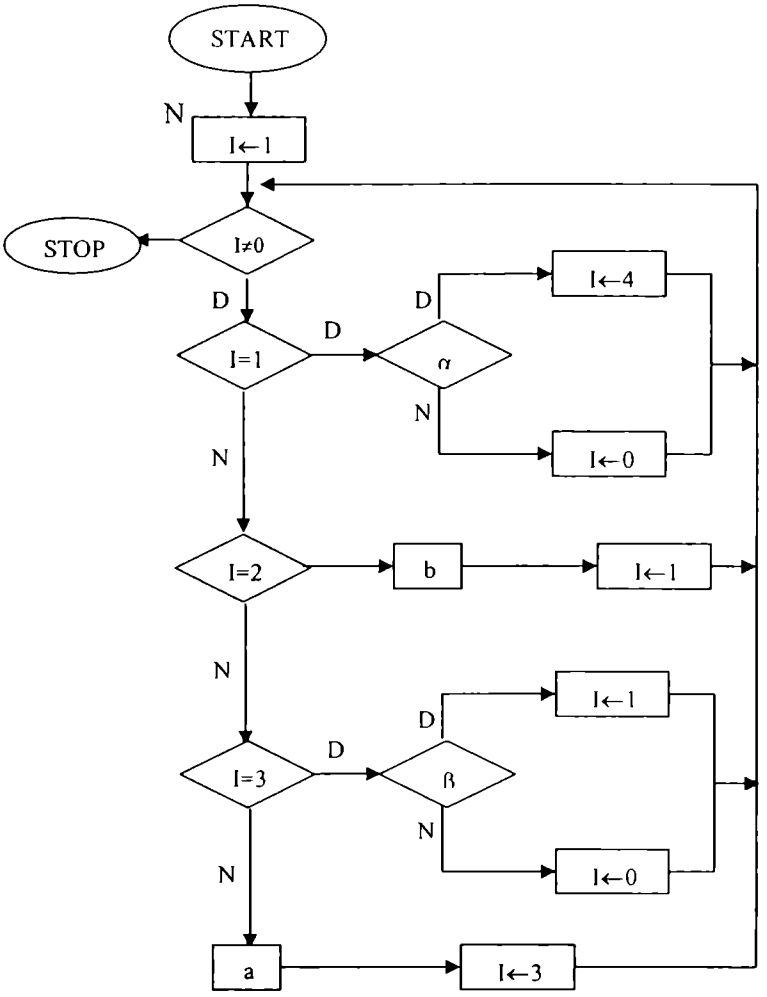


Fig. 1.11

Observație: În practică, structurarea unei scheme logice se face prin metodele următoare:

1. **Tehnica dedublării blocurilor,** care se folosește când schema nu conține cicluri. În acest caz, schema se structurează numai

https://biblioteca-digitala.ro / https://unibuc.ro

prin π şi Δ repetând la nevoie unele blocuri pe diferite ramuri ale schemei.

2. **Tehnica variabile booleene,** care corespunde teoremei lui Böhm şi Jacopini, tehnică descrisă anterior.
3. **Tehnica variabilei de stare,** care corespunde teoremei de structură NL a lui Jackson, tehnică pentru care se aplică algoritmul din demonstraţia teoremei.

Exemplu complet:

Pentru schema logica alăturată se cere:

1. structurare (prin dedublare)
2. formula de structură
3. diagrama de structură NL
4. programul în limbajul NL.

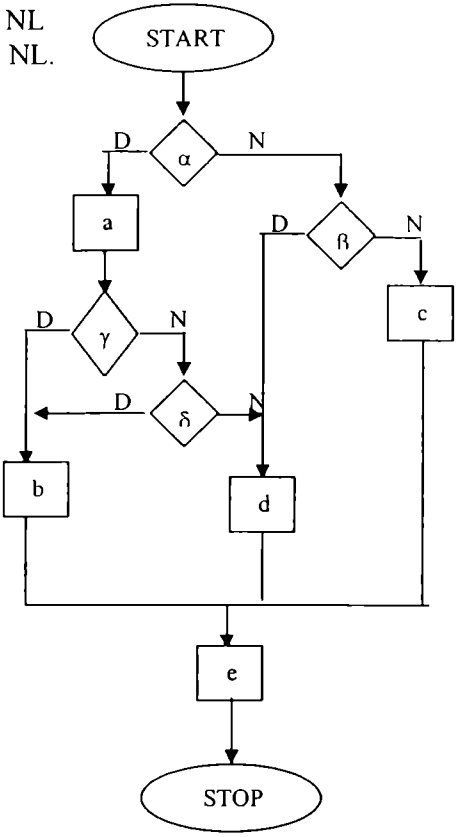


Fig. 1.12

https://biblioteca-digitala.ro / https://unibuc.ro

Structurare:

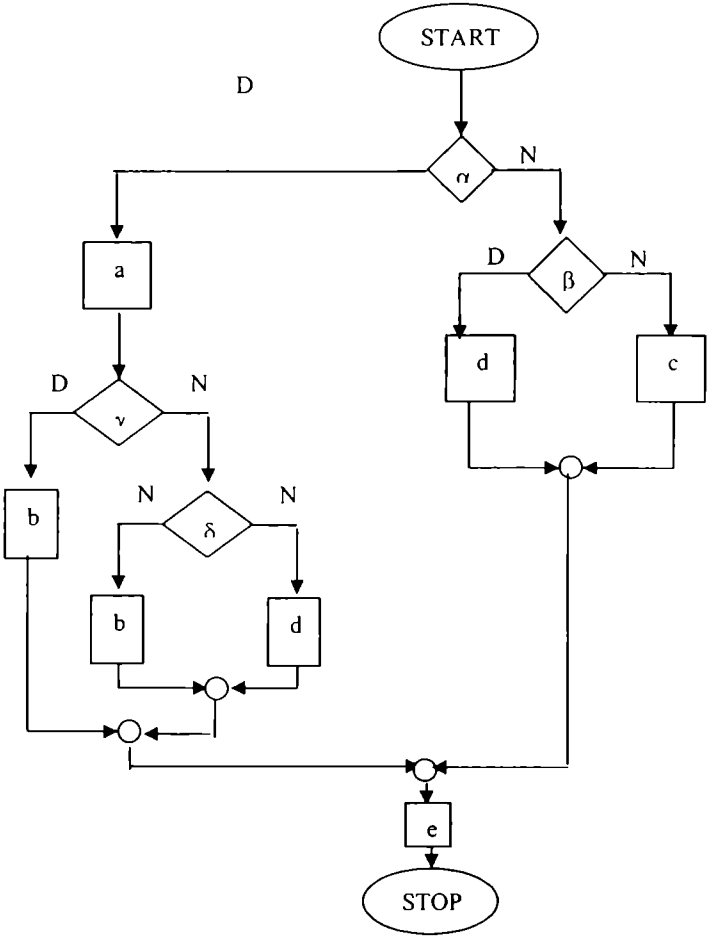


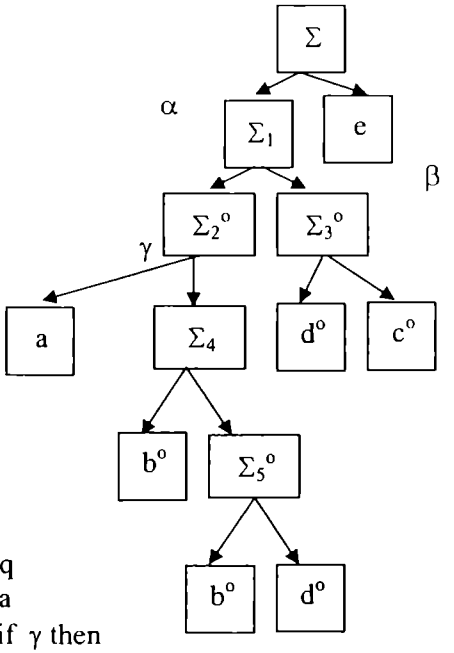
Fig. 1.13

Formula de structură:

$$\Sigma=\pi(\ \Delta(\ \alpha;\ \pi\ (a,\ \Delta(\ \gamma,\ b,\ \Delta(\delta,\ b,\ d))),\ \Delta(\beta,\ d,\ c)),\ e)$$

https://biblioteca-digitala.ro / https://unibuc.ro

Diagrama NL:



Programul în limbaj NL

```
Σ seq
    Σ1 if α
        Σ2 seq
            do a
            Σ4 if γ then
                do b
            Σ4 else
                Σ5 if δ then
                    do b
                Σ5 else
                    do d
                Σ5 end
            Σ4 end
        Σ2 end
    Σ1 else
        Σ3 if β then
                do d
        Σ3 else
                do c
        Σ3 end
    Σ1 end
        do e
Σ end
```

Fig. 1.14

https://biblioteca-digitala.ro / https://unibuc.ro

Observație: Etichetele programului NL sunt numele nodurilor interne ale diagramei de structură (nodurile neterminale).

1.6 Reprezentarea algoritmilor în pseudocod [37, 9, 44]

Pseudocodul este un mod de reprezentare a algoritmilor mult mai apropiat atât de limbajul vorbirii curente, de limbajul matematic uzual cât și de limbajele de programare moderne și care în plus permite folosirea unor instrucțiuni care conțin condensat structurile de control frecvent folosite în programare.

Un program în pseudocod este o succesiune de una sau mai multe proceduri, una fiind procedură principală, succesiune ce se execută secvențial începând cu prima din secvență.
O procedură are formatul:

```
|—— procedure nume [ (listă de parametri) ]
|         parte declarativă
|         S
|—— end [nume]          ,        unde:
```

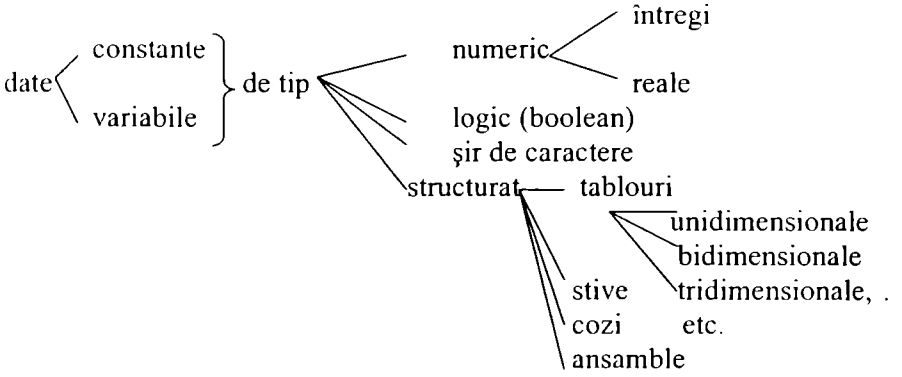
- *nume*, identifică procedura
- [ ] indică opționalitatea conținutului său
- *listă de parametri*, este un șir de variabile separate prin virgule. Ea permite transmiterea datelor între proceduri
- *partea declarativă*, este o secvență de instrucțiuni ce descriu tipul variabilelor folosite de procedura
- *S* este o secvență de instrucțiuni efective care efectuează operațiile elementare ale algoritmului și care se execută începând cu prima instrucțiune din S și apoi în ordine secvențială celelalte până la întâlnirea instrucțiunilor: *stop, return, exit, cycle, go to, call,* care modifică ordinea secvențială.

Apelul unei proceduri se face prin instrucțiunea *call*, folosind sintaxa:
call nume [(listă de argumente)]
Este necesară o bijecție între **lista de parametri** și **lista de argumente**, corespondență care să conserve: numărul, ordinea, tipul și dimensiunile.
Execuția unei proceduri se încheie cu instrucțiunile *stop* și *return*.
Instrucțiunea *return* determină revenirea la procedura apelantă și anume la instrucțiunea ce urmează imediat *call*.

https://biblioteca-digitala.ro / https://unibuc.ro

Elementele de bază ale pseudocodului

Acestea sunt:



cuvinte cheie

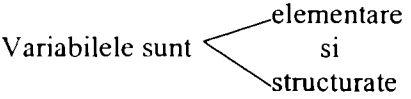


Constantele sunt elemente de dată ce nu îşi modifică valoarea în timpul execuției programelor.

Constantele

Numerice, sunt numere întregi şi reale
logice, sunt 0 pentru *false* si 1 pentru *true*
şiruri de caractere, sunt succesiuni de caractere cuprinse între apostrofuri

Variabilele sunt elemente de data ale căror valoari se modifică în timpul execuției programelor; se pot referi prin *nume alfanumeric*.



Operațiile cu date:

- pentru date întregi şi reale operațiile permise sunt cele aritmetice uzuale: *, -, +, /, **(^)

https://biblioteca-digitala.ro / https://unibuc.ro

- pentru date logice operațiile sunt: disjuncția ($\vee$)
  conjuncția ($\wedge$)
  negația ($\neg$)
- pentru șiruri de caractere operațiile sunt: concaternarea și comparația
- pentru stive, cozi, ansamble operațiile permise sunt:
  - extragerea unui element: x<=A
  - introducerea unui element: x => A
  - verificarea dacă o structură este vidă A=∅, (A = structura)

Cuvintele cheie sunt cuvinte rezervate din limba engleză cu semnificații predefinite prin care se identifică instrucțiunile. Pentru inceput se recomanda sublinierea lor pentru a le distinge de alte elemente ale pseudocodului.

Instrucțiunile se scriu liber, putând scrie una pe mai multe rânduri sau mai multe pe un rând, caz în care separatorul este ";". Rândurile se incheie cu spații sau cu comentarii de forma: /*...*/ urmate de spații.

Instrucțiunile declarative descriu tipul variabilelor folosite și sunt: *integer, real, bool, char, array, stack, queue si heap.*

Acestea sunt urmate de o listă de variabile separate prin ",".

Exemplu: integer a,b,c; real d(m,n); stack S

Instrucțiunile efective descriu operațiile elementare ale algoritmului și sunt: *de atribuire*($\leftarrow$), *read/write, return, stop, exit, cycle, goto, call, while, if-then-else, for, do* si *case.*

Acestea au urmatoarele formate:

1. **Instrucțiunea de atribuire**: $v \leftarrow e$, unde v este o variabilă de tip: integer, real, bool, char, iar "e" este o expresie de același tip cu v.

Se admit și atribuiri de forma $(v_1, v_2, ..., v_n) \leftarrow (e_1, e_2, ..., e_n)$ cu sensul ca pentru $\forall$ i=1,...,n , $v_i \leftarrow e_i$.

2. **Instrucțiunea de citire:** *read* $a_1, a_2, ..., a_n$, unde pentru $\forall$ i = 1,...,n, a_i este o variabilă în a cărei locație de memorie se introduce data de ordin i.

3. **Instrucțiunea de scriere:** *write* $x_1, ..., x_p$ unde pentru $\forall$ i =1,..., p , x_i este constantă, variabilă, șir de caractere sau expresie a cărei valoare se afișează.

https://biblioteca-digitala.ro / https://unibuc.ro

4. **Instrucțiunea return** determină transfer în procedura apelantă la instrucțiunea imediat următoare celei de apel (call).

5. **Instrucțiunea stop** determină oprirea execuției procedurii curente și deci a întregii secvențe de proceduri

6. **Instrucțiunea exit** determină trecerea la prima instrucțiune de după cel mai interior ciclu do, while, for care o conține , ieșindu-se astfel din ciclu.

7. **Instrucțiunea cycle** determină trecerea la ultima instrucțiune a celui mai interior ciclu care o conține (transfer la *repeat* sau *until*), permițând astfel reluarea celui mai interior ciclu ce o conține.

8. **Instrucțiunea goto** are formatul ***goto*** et, unde et este eticheta unei instrucțiuni din aceeași procedură. Eticheta este un nume asociat instrucțiunii, precede instrucțiunea și este separată de aceasta prin ":".
Instrucțiunea *goto* realizeaza un salt necondiționat la instrucțiunea precedată de eticheta "et".
Se recomandă evitarea folosirii acestei instrucțiuni deoarece îngreuiază înțelegerea algoritmului și in plus contravine principiilor programării structurate.

9. **Instrucțiunea de ramificare if-then-else:** are formatul

```
if p then S1
     [else S2]
endif
```

Dacă pentru p=false nu se indică nici o acțiune, "else S_2" nu se va folosi. Astfel formatul conține și structura pseudoalternativă *if-then*.

10. **Instrucțiunea de ramificare case** are formatul

```
case   p1: S1                    case   p1: S1
       p2: S2                           .....
       .....          sau               pn: Sn
       pn: Sn                           else Sn+1
endcase                          endcase
```

https://biblioteca-digitala.ro / https://unibuc.ro

în care $\forall$ i, j $\neq 1,\ldots,n$, $p_i \wedge p_j = 0$ (nu există două predicate simultan adevărate adica cel mult un predicat este adevărat)

Pentru primul format avem şi condiţia: $\bigvee_{i=1}^{n} p_i = 1$, adică nu toate predicatele sunt false $\Leftrightarrow$ cel puţin un predicat adevărat.

Combinând aceste două condiţii, pentru formatul 1 rezultă că dintre $p_1, \ldots, p_n$ există un singur predicat p_i adevărat şi toate celelalte sunt false. Se va executa secvenţa de instrucţiuni S_i corespunzătoare lui p_i=1 şi apoi se va trece la *endcase*.

11. **Instrucţiunea repetitivă while** are formatul:

```
|— while p
|    S
|_ repeat ,
```

unde p este un predicat şi S este o secvenţă de instrucţiuni.

12. **Instrucţiunea repetitivă do** are două formate:

```
|‾‾ do                          |‾‾ do
|    S                         |    S
|__ until p        sau         |__ repeat
```

În al doilea format, pentru a nu avea ciclare infinită, S trebuie să conţină *stop, return* sau *exit.*

13. **Instrucţiunea repetitivă for** are formatul:

```
|— for v=e1,e2 [,e3]
|   S
|_ repeat                , unde:
```

v=variabilă de control numită contor sau variabilă de ciclare
$\forall i = \overline{1,3}$, e_i este o expresie aritmetică şi $e_3 \neq 0$
Dacă e_2=1, ea poate lipsi din formatul instrucţiunii.

14. **Instrucţiunea call** a fost descrisă la începutul paragrafului.
Prin secvenţă de instrucţiuni înţelegem:

https://biblioteca-digitala.ro / https://unibuc.ro

1. Instrucțiunile *(←), read, write, stop, return, exit, cycle, call* sunt secvente de instructiuni..
2. Dacă S_1, ... , S_n, S_{n+1} sunt secvențe de instrucțiuni, atunci instrucțiunile *while, if-then-else, for, do, case* sunt secvențe de instrucțiuni.
3. Dacă S_1 si S_2 sunt secvențe de instrucțiuni, atunci S_1; S_2 este de asemenea secvență de instrucțiuni.
4. Orice secvență de instrucțiuni se obține pornind cu instrucțiunile de la 1 şi aplicând 2 şi 3 de un număr finit de ori.

Exemple de algoritmi reprezentați în pseudocod:
Pentru a(n), n≥1 vector cu n componente reale, să se determine

$$m = \min_{i=\overline{1,n}}(a_i) \quad , \quad M = \max_{i=\overline{1,n}}(a_i)$$

Pseudocodul asociat este:

```
procedure minmax;
integer n,i; real a(n), m, M;
read n, a(n)
m←a1; M←a1
for i=2,n
    if ai<m then m←ai
    endif
    if ai>M then M←ai
    endif
repeat
write m,M
stop
end.
```

1. Fie $f : \prod_{i=1}^{k} \{1,2,...,n_i\} \to \{1,2,...,\prod_{i=1}^{k} n_i\}$ dată prin:

$f(i_1,...,i_k) = i_1+(i_2-1)\cdot n_1+(i_3-1)\cdot n_1\cdot n_2+...+(i_k-1)\cdot n_1\cdot...\cdot n_{k-1}$.

Aceasta asociază elementului de pe poziția $(i_1,i_2, ... , i_k)$ din tabloul k-dimensional $a(n_1,...,n_k)$ adresa de memorie unde acesta va fi înregistrat.

Să se calculeze valoarea lui f în punctul $(i_1, ... ,i_k)$ dat.

https://biblioteca-digitala.ro / https://unibuc.ro

Evident, $f(i_1, \ldots, i_k) = i_1 + n_1 \cdot [i_2 - 1 + n_2 \cdot (i_3 - 1 + n_3[i_{n-1} + \ldots + n_{k-2} \cdot (i_{k-1} - 1 + n_{k-1}(i_k - 1)]]\ldots]$

Notam $f = i_1 + n_1 \cdot [i_2 - 1 + n_2 \cdot (i_3 - 1 + n_3[i_{n-1} + \ldots + n_{k-2} \cdot (i_{k-1} - 1 + n_{k-1}(i_k - 1)]]\ldots]$

$\Rightarrow f - 1 = i_1 - 1 + n_1[i_2 - 1 + n_2[i_3 - 1 \cdot \ldots \cdot (i_{k-2} - 1 + n_{k-2}(i_{k-1} - 1 + n_{k-1}(i_k - 1)]\ldots]$

folosim metoda mersului invers.

Algoritmul este:

$f \leftarrow i_k - 1$

pentru j=k-1, 1,-1, luam $f \leftarrow f * n_j + i_j - 1$

$f \leftarrow f + 1$

iar pseudocodul este:

```
procedure funcție
integer k,n(k), i(k), j, f
read k, n(k), i(k)
f←ik-1
for j=k-1,1,-1
  f←f*nj+ij-1
repeat
f←f+1
write f
stop
end
```

3.Sortarea prin inserție are pseudocodul:

```
procedure insertie
integer n,i,a(n)
read n,a(n)
for i=2,n
s←ai
  for k=i-1,1,-1
        if s<ak then ak+1←ak
        endif
  repeat
ak+1←s;
repeat
write a(n)
stop
end
```

https://biblioteca-digitala.ro / https://unibuc.ro

1.7 Reprezentarea algoritmilor prin tabele de decizie [5]

Orice problemă sau proces de prelucrare P este caracterizată de o mulțime de condiții în care se execută o mulțime de acțiuni. Condițiile și acțiunile ce caracterizează problema P, precum și interacțiunile dintre acestea pot fi descrise printr-o tabelă de decizie asociată problemei P.

Definiție: Se numește ***tabelă de decizie*** asociată unei probleme P un tablou bidimensional cu dublă intrare T_p care are forma din figura alăturată:

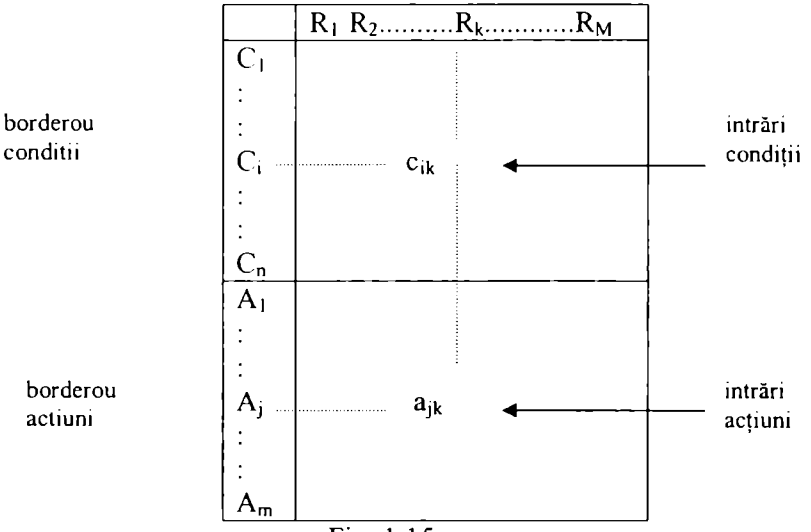


Fig. 1.15

unde:

C_i, i=1,...,n sunt condițiile sau situațiile semnificative ale lui P

A_j, j=1,...,m sunt acțiunile posibile ale problemei P

R_k, k=1,...,M sunt regulile de decizie care precizează pentru fiecare sistem de valori ale condițiilor C_1, C_2, ..., C_n, acțiunile care urmează să fie executate.

Notăm această tabelă de decizie prin $T_P(n,m,M)$.

Observăm că dacă v_i este numărul de valori ale condiției C_i, atunci numărul maxim de reguli de decizie este:

https://biblioteca-digitala.ro / https://unibuc.ro

$$N = \prod_{i=1}^{n} v_i$$

Astfel, în T_P (n,m,M) avem $M \leq N$.

Atunci când M=1 se obține o tabelă de decizie degenerată. Aceasta se întâmplă când în tabelă lipseşte borderoul condițiilor şi porțiunea de intrări condiții. În acest caz, tabela este alcătuită dintr-o secvență de instrucțiuni care constituie acțiunile problemei. În cele ce urmează vom lucra numai cu tabele de decizie nedegenerate.

Atunci când M=N spunem că avem o tabelă de decizie cu **intrări complete**.

O tabelă de decizie se numeşte **completă** dacă pentru fiecare sistem de valori ale condițiilor sunt precizate acțiunile ce se intreprind.

Evident, o tabelă cu intrări complete este completă. O tabelă ce nu este cu intrări complete în general nu este completă, dar se poate transforma într-o tabelă completă dacă mulțimea regulilor ce lipsesc din tabelă este înlocuită cu regula ELSE, care se introduce în noua tabelă şi căreia i se asociază o mulțime de acțiuni sau prin lipsă i se asociază acțiunea EROARE.

Pentru construirea rapidă a unei tabele de decizie cu intrări complete vom folosi următorul algoritm:

Algoritm (de completare a porțiunii de intrări condiții a unei table T_P (n,m,N)).

1. $i \leftarrow 1$, $N = v_1 v_2 \ldots v_n$, $N_0 \leftarrow N$
2. $N_i \leftarrow N_{i-1}/v_i$
3. completăm pe linia condiției C_i, sub fiecare grup constant de valori ale lui C_{i-1}, de N_i ori fiecare din cele v_i valori ale lui C_i.
4. dacă $i<n$, luăm $i \leftarrow i+1$ şi GOTO 2, altfel STOP.

Observație: Convenim ca C_0 să fie o condiție care nu face parte din tabelă şi care are toate valorile blank.

Exemplul 1 de asociere la o problemă P a unei tabele de decizie

P: Pentru $a,b \in R$, să se rezolve ecuația $a \cdot x = b$.

Soluție: Din analiza problemei se observă factorii care caracterizează soluția problemei. Aceştia sunt:

Condițiile: C_1: a=0
C_2: b=0

Acțiunile: A_1: write "$\varnothing$"

https://biblioteca-digitala.ro / https://unibuc.ro

A_2: write "R"
A_3: write b/a

Condițiile C_1 și C_2 sunt predicate booleene cu doar două valori posibile: D pentru condiție îndeplinită și N pentru condiție neîndeplinită.

Astfel, numărul regulilor tabelei de decizie este: N=4.

Vom lua ca intrări ale acțiunilor valoarea "*" când acțiunea se execută și blank în caz contrar.

Tabela de decizie asociată lui P, completată în porțiunea de intrări condiții după algoritmul de mai sus, este următoarea:

	R_1	R_2	R_3	R_4
C_1	D	D	N	N
C_2	D	N	D	N
A_1		*		
A_2	*			
A_3			*	*

Aceasta se poate reprezenta și sub forma următoare:

	R_1	R_2	R_3	R_4
C_1	D	D	N	N
C_2	D	N	D	N
A	A_2	A_1	A_3	A_3

Pe această formă observăm că atunci când C_1=N, indiferent cum este C_2, se execută acțiunea A_3. Aceasta indică faptul că pentru C_1=N, C_2

https://biblioteca-digitala.ro / https://unibuc.ro

este neutră la executarea acțiunii A_3. Vom indica aceasta introducând valoarea "-" (indiferent) pentru condiția C_2 atunci când C_1=N. Tabela va avea acum forma restrânsă următoare

:

	R_1	R_2	R_{34}
C_1	D	D	N
C_2	D	N	-
A	A_2	A_1	A_3

Observație: orice tabelă se poate reprezenta sub această ultimă formă dacă pentru fiecare $j \in \{1,...,M\}$ definim noțiunea de *program de acțiuni asociat regulii* R_j prin:

$PA_j=\{A_k, k \in \{1,...,m\} | a_{kj}=*\}$

Cu aceasta, fiecărei reguli de decizie i se asociază un program de acțiuni unic.

În cazul nostru, $PA_1=\{A_2\}$, $PA_2=\{A_1\}$, $PA_3=\{A_3\}$

Exemplul 2: Presupunem că o societate economică are nevoie de o anumită cantitate dintr-un material procurabil în două calități, dintr-un depozit D. Dacă D nu dispune de calitatea solicitată în cantitatea cerută, societatea este întrebată dacă nu dorește livrarea cantității dorite dar cu material din cealaltă calitate, atunci când aceasta există în D în cantitatea cerută. Dacă nu se acceptă schimbarea calității, sau D nu dispune de cantitatea dorită nici din cealaltă calitate, atunci se face o comandă de reaprovizionare cu material din calitatea solicitată. Dacă se acceptă schimbarea calității va fi livrată cantitatea cerută de calitatea schimbată.

Dorim să asociem acestui proces de prelucrare o tabelă de decizie.

Soluție: Examinând cu atenție enunțul problemei descoperim factorii care influențează luarea deciziei și anume:

- calitatea de material solicitată
- disponibilitatea efectivă a materialului de calitatea dorită în cantitatea necesară
- acceptarea schimbării calității solicitate inițial.

Acțiunile posibile ale procesului sunt:

https://biblioteca-digitala.ro / https://unibuc.ro

- eliberarea materialelor în cantitatea şi calitatea solicitată sau acceptată, separat pe cele două calităţi.
- emiterea unei comenzi de aprovizionare pentru calitatea şi cantitatea solicitată, separat pe cele două calităţi

De aici rezultă următoarele condiţii şi acţiuni ale procesului de prelucrare:

C_1 – se solicită materialul X de calitatea I-a
C_2 – D dispune de materialul solicitat în cantitatea Q
C_3 – D dispune de materialul X de calitatea a II-a în cantitatea Q
C_4 – societatea acceptă schimbarea calităţii.

Acţiuni:

A_1 – eliberează X de calitatea I-a în cantitatea Q
A_2 – eliberează X de calitatea a II-a în cantitatea Q
A_3 – emite comanda de aprovizionare cu X de calitatea I-a
A_4 – emite comanda de aprovizionare cu X de calitatea a II-a

Condiţiile C_1, C_2, C_3, C_4 au câte două valori posibile: D pentru DA şi N pentru NU. Astfel, numărul total de reguli ale tabelei asociate este $N = \prod_{i=1}^{4} 2 = 16$.

Şi aici vom lua ca intrări ale acţiunilor valoarea "*" pentru acţiune executată şi blank în caz contrar.

Tabela de decizie asociată acestui proces, completată în porţiunea de intrări condiţii, conform algoritmului de mai sus este următoarea:

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}
C_1	D	D	D	D	D	D	D	D	N	N	N	N	N	N	N	N
C_2	D	D	D	D	N	N	N	N	D	D	D	D	N	N	N	N
C_3	D	D	N	N	D	D	N	N	D	D	N	N	D	D	N	N
C_4	D	N	D	N	D	N	D	N	D	N	D	N	D	N	D	N
A_1	*	*	*	*							*					
A_2					*				*	*			*	*		
A_3						*	*	*								
A_4												*			*	*

https://biblioteca-digitala.ro / https://unibuc.ro

Observând pe această tabelă că există reguli cu acțiuni identice, ea se poate restrânge sub următoarea formă unde: $PA_1=\{A_1\}$, $PA_2=\{A_2\}$, $PA_3=\{A_3\}$, $PA_4=\{A_4\}$.

	R_1*	R_2*	R_3*	R_4*	R_5*	R_6*	R_7*	R_8*
C_1	D	D	D	D	N	N	N	N
C_2	D	N	N	N	-	D	D	N
C_3	-	D	D	N	D	N	N	N
C_4	-	D	N	-	-	D	N	-
Programe de acțiuni	PA_1	PA_2	PA_3	PA_3	PA_2	PA_1	PA_4	PA_4

Pornind de la tabele de decizie în forma redusă se pot scrie direct programe în limbaje de programare moderne care să rezolve problema căreia i-a fost asociată tabela de decizie. Ordini diferite de testare a condițiilor unei tabele de decizie conduc la programe diferite. Se pune problema determinării acelei ordini de testare care să conducă la programe care să minimizeze fie memoria ocupată de codul programului, fie timpul necesar execuției acestuia, fie ambele.

Au fost imaginați o serie de algoritmi de translatare care să obțină un program optim atât în ceea ce privește memoria acupată cât și în ce privește timpul de calcul necesar. Acești algoritmi au fost codificați și astfel există câteva produse software care translatează optim tabele de decizie în programe scrise în COBOL, C, etc. așa cum ar fi TABDEC, TABDECO, etc. Aceste produse au la bază o serie de algorimi de translatare care se împart în două mari categorii:

- algorimi de translatare bazați pe analiza sintactică a porțiunii de intrări condiții printre care: algoritmii 1 și 2 ai lui POLLAK, algoritmul lui PRESS, algoritmii 1 și 2 ai lui SHWAYDER, algoritmul lui CANAPATY-RAJARAMAN, etc.
- algoritmi de translatare bazați pe anumite tehnici de mascare, printre care: algoritmul lui KIRK, algoritmul modificat al lui KIRK, algoritmii 1 și 2 ai lui MUTHUKRISHNAN și RAJARAMAN, etc.

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL II Structuri elementare de date, modularizare si codificarea informatiilor

2.1 Structuri statice; tablouri [8, 52, 90]

Memoria internă a unui calculator are o structură liniară şi astfel este ocupată de variabilele unui program succesiv locaţie după locaţie.

Variabilele simple ocupă câte o locaţie de memorie, iar elementele unui vector ocupă în ordine locaţii succesive de memorie.

Pentru un tablou bidimensional, $T_{m,n}$ cu m linii şi n coloane, există două metode de reprezentare în memorie:

- T se memorează succesiv linie după linie
- T se memorează succesiv coloană după coloană.

Pentru primul caz, elementele lui T se memorează în ordinea:

$t_{11}\ t_{12}\ \ldots\ t_{1n}\ t_{21}\ t_{22}\ \ldots t_{2n} \ldots t_{m-1\ 1}\ t_{m-1\ 2}\ \ldots t_{m-1\ n}\ t_{m1}\ t_{m2}\ \ldots\ t_{mn}$, ocupând respectiv locaţiile:

$\underbrace{1\ 2\ \ldots n}\ \underbrace{n+1\ \ldots\ 2n}\ \ldots\ \underbrace{(m-1)n+1\ \ldots\ (n-1)n+m}$

linie 1 linie 2 linie m

Dacă definim $g:\{1,2,\ldots,m\}x\{1,2,\ldots,n\}\rightarrow\{1,2,\ldots,mn\}$ prin:

$g(i,j) = (i-1)n+j$, $i=1,m$, $j=1,n$, obţinem o funcţie cu care realizăm memorarea elementelor lui T pe linii.

Pentru a doua metodă (memorarea pe coloane) elementele din T se vor depune în memorie în ordinea:

$t_{11}\ t_{21}\ \ldots t_{m1}\ t_{12}\ \ t_{22}\ \ t_{m2}\ \ldots\ t_{1n}\ \ t_{2n}\ldots t_{mn}$, ocupând respectiv locaţiile

$\underbrace{1\ 2\ 3\ \ldots m}\ \underbrace{m+1\ \ldots\ 2m}\ ,,,,,,\ \underbrace{(n-1)m+1\ \ (n-1)m+2\ \ldots\ (m-1)n+n}$

coloană 1 coloană 2 coloana n

Dacă definim:

$f:\{1,2,\ldots m\}x\{1,2,\ldots,n\}\rightarrow\{1,2,\ldots,mn\}$ prin $f(i,j)=(j-1)m+i$, $j=1,n$, $i=1,m$, obţinem o funcţie ce realizează memorarea elementelor lui T pe coloane.

Teorema: Funcţiile f şi g definite mai sus sunt bijective .

Demonstraţie. Rezultă din definiţiile celor două funcţii.

https://biblioteca-digitala.ro / https://unibuc.ro

Exercițiu. Sa se determine inversele lui f și g.

Definiție: Fie (A_i, ρ_i) o mulțime total ordonată pentru $i\in\{1,...,n\}$ (adică ρ_i este o relatie reflexivă, antisimetrică, tranzitivă și totală) și fie $B=\prod_{i=1}^{n} A_i$. Definim pe B două relații binare ρ și ρ' prin $\rho,\rho'\subset BxB$ și dacă $\alpha=(a_1...a_n)$ și $\beta=(b_1...b_n)\in B$, atunci:

$\alpha\rho\beta\Leftrightarrow$ 1.$\alpha=\beta$ sau
2. $a_1\neq b_1$ dar $a_1\rho_1 b_1$ sau
3. $\exists k\in\{2,...,n\}$ a.î. $a_i=b_i$, $\forall i\in\{1,...,k-1\}$ și $a_k\neq b_k$ dar $a_k\rho_k b_k$

$\rho'\subset BxB$ și dacă $\alpha=(a_1...a_n)$ și $\beta=(b_1...b_n)\in B$, atunci:

$\alpha\rho'\beta\Leftrightarrow$ 1. $\alpha=\beta$ sau
2. $a_n\neq b_n$ dar $a_n\rho_n b_n$ sau
3. $\exists k\in\{1,...,n-1\}$ a.î.$a_k\neq b_k$ dar $a_k\rho_k b_k$ și $a_i=b_i$, $\forall i\in\{k+1,...,n\}$

Teorema. Relațiile ρ și ρ' definite mai sus sunt relații de ordine pe B.

Demonstrație: Se arată că ρ și ρ' sunt reflexive, antisimetrice și tranzitive folosind definițiile lor și faptul că pentru $\forall i\in\{1,...,n\}$, ρ_i este relație **de ordine**.

Relația ρ definită mai sus se numește ***relație de ordine lexicografică*** indusă de relațiile ρ_i, $i\in\{1,...,n\}$.

Relația ρ' se numește ***relație de ordine invers lexicografică*** indusă de relațiile ρ_i, $i\in\{1,...,n\}$.

Observație. Dacă pentru $\forall i\in\{1,...,n\}$, $A_i=N$, atunci ρ_i="$\leq$" de pe N.

Astfel, în memorarea pe linii elementele:

11, 12,...1n, 21, 22,..., 2n, 31, 32,..., 3n,...m1, m2, m3,..., mn,

$\leq$(cf. 3) $1\leq 2$ (cf.2)

sunt în ordine lexicografică

În memorarea pe coloane elementele:

11, 21,..., m1, 12, 22, ..., m2,..., 1n, 2n,..., mn,

$\leq$ (cf.3) $1\leq$ (cf.2)

sunt în ordine invers lexicografică.

https://biblioteca-digitala.ro / https://unibuc.ro

În concluzie, memorarea unui tablou pe linii corespunde ordinii lexicografice, iar memorarea pe coloane corespunde ordinii invers lexicografice, definite pe $\{1,2,\ldots,m\}x\{1,2,..,n\}$.

Fie acum T_{n_1, n_3} un tablou bidimensional cu n_1 linii şi n_2 coloane.

Aşezăm elementele lui $\{1, 2,\ldots, n_1\}x\{1, 2,\ldots, n_2\}$ conform celor două ordini:

ordinea lexicografică **ordinea invers lexi cografică**

i_1, i_2	g
n_2 ori { 1 1; 1 2; 1 3; . .; 1 n_2 }	
n_2 ori { 2 1; 2 2; 2 3; . .; 2 n_2 }	$\Rightarrow g(i_1,i_2) = (i_1 - 1)n_2 + i_2$
: :	
n_2 ori { n_1 1; n_1 2; n_1 3; . .; n_1 n_2 }	

i_1, i_2	f
n_1 ori { 1 1; 2 1; 3 1; . .; n_1 1 }	
n_1 ori { 1 2; 2 2; 3 2; . .; n_1 2 }	$\Rightarrow f(i_1,i_2) = i_1 + (i_2-1)n_1$
: :	
n_1 ori { 1 n_2; 2 n_2; 3 n_2; . .; n_1 n_2 }	

unde $i_1=\{1, 2, \ldots, n_1\}$, $i_2=\{1, 2,\ldots, n_2\}$

Pentru T_{n_1, n_2, n_3} tablou tridimensional aşezăm elementele lui:

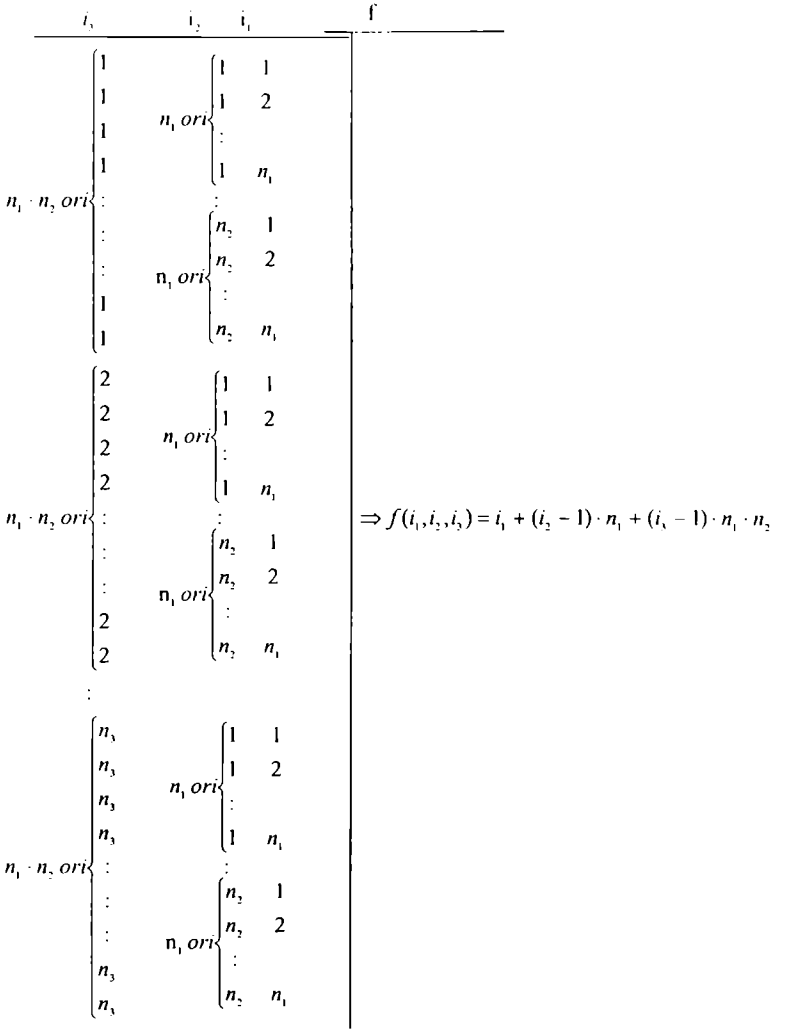
$\{1,2,\ldots,n_1\}x\ \{1,2,\ldots,n_2\}x\ \{1,2,\ldots,n_3\}$ conform celor două ordini.

https://biblioteca-digitala.ro / https://unibuc.ro

i_1	i_2	i_3	g
1	n_3 ori	1 1 … 1 n_3	
$n_2 \cdot n_3$ ori			
2	n_3 ori	1 1 … 1 n_3	$g(ig_1, i_2, i_3) = (i_1 - 1) \cdot n_2 \cdot n_3 +$
$n_2 \cdot n_3$ ori			$+(i_2\text{-}1) \bullet n_3 + i_3$
n_1	n_3 ori	1 1 … 1 n_3	
$n_2 \cdot n_3$ ori			

unde $i_1 = \overline{1, n_1}$, $i_2 = \overline{1, n_2}$, $i_3 = \overline{1, n_3}$

https://biblioteca-digitala.ro / https://unibuc.ro



unde $i_1=\{1,2,\ldots,n_1\}$, $i_2=\{1,2,\ldots,n_2\}$, $i_3=\{1,2,\ldots,n_3\}$

În general, pentru tablourile k-dimensiune $T_{n_1,n_2,\ldots n_k}$ observăm un algoritm de aşezare a elementelor produsului:

$\prod_{i=1}^{k} \{1, 2, \ldots, n_i\}$ în cele două ordini.

https://biblioteca-digitala.ro / https://unibuc.ro

Pentru ordinea lexicografică, algoritmul de ordonare este:

1. Punem pe o linie indicii $i_1\ i_2\ \ldots i_l \ldots\ldots i_k$, unde $i_1=\{1,2,..,n_1\},\ldots,$ $i_k=\{1,2,..,n_k\}$.
2. Sub indicele i_1 punem fiecare din valorile sale de câte $n_2n_3\ldots n_k$ ori.
3. Pentru fiecare $l = \{2,3,\ldots,k-1\}$, sub indicele i_l punem, pentru fiecare grup constant de valori ale lui i_{l-1} , fiecare din valorile sale de $n_{l+1}n_{l+2}\ldots n_k$ ori.
4. Sub indicele i_k punem pentru fiecare grup constant de valori ale lui i_{k-1} fiecare din valorile sale $1,2,\ldots,n_k$ o singură dată.

Cu acest algoritm găsim funcția:

$$g:\prod_{i=1}^{k}\{1,2,...,n_i\} \rightarrow \{1,2,...,\prod_{i=1}^{k} n_i\} \text{ dată prin:}$$

$$g(i_1,i_2,\ldots,i_k) = (i_1\text{-}1)\ n_2n_3\ldots n_k+(i_2\text{-}1)\ n_3n_4\ldots n_k+\ldots.+i_{k-1}\text{-}1)n_k+i_k\ .$$

Pentru ordinea invers lexicografică, algoritmul de ordonare este:

1. Punem pe o linie indicii $i_1\ i_2\ \ldots i_l \ldots\ldots i_k$
2. Sub indicele i_k punem fiecare din valorile sale de câte $n_1n_2\ldots n_{k-1}$ ori.
3. Pentru fiecare $l = \{2,3,\ldots,k-1\}$, sub indicele i_l punem, pentru fiecare grup constant de valori ale lui i_{l+1} , fiecare din valorile sale de $n_1n_2\ldots n_{l-1}$ ori.
4. Sub indicele i_1 punem pentru fiecare grup constant de valori ale lui i_2 fiecare din valorile sale $1,2,\ldots,n_1$ o singură dată.

Cu acest algoritm găsim funcția:

$$f:\prod_{i=1}^{k}\{1,2,...,n_i\} \rightarrow \{1,2,...,\prod_{i=1}^{k} n_i\} \text{ dată prin:}$$

$$f(i_1,i_2,\ldots,i_k) = i_1+(i_2\text{-}1)n_1+(i_3\text{-}1)n_1n_2+\ldots.+(i_k\text{-}1)n_1n_2\ldots n_{k-1}.$$

Observații:

a) cu funcția g memorăm tabloul k-dimensional $T_{n_1,n_2,..,n_k}$ conform relației de ordine lexicografice . Ea asociază elementului de pe poziția $(i_1,\ldots,i_k)$ din T, numărul locației de memorie în care va fi depus.

https://biblioteca-digitala.ro / https://unibuc.ro

b) cu funcția f memorăm tabloul $T_{n_1,n_2,\ldots,n_k}$ conform relației de ordine invers lexicografică. Ea asociază elementului de pe poziția $(i_1,i_2,\ldots,i_k)$ din T numărul locației de memorie în care va fi memorat.

Teorema . Funcțiile f și g definite mai sus sunt bijective.
Demonstrația rezultă din definiția funcțiilor.

Exercițiu : Sa se determine f^{-1} și g^{-1}.

Calculul valorilor lui f și f^{-1} este realizat de procedurile **directă** respectiv **inversă** conform pseudocodurilor următoare:

```
procedure directă
integer k,j,n(k),i(k),f
read k,n(k),i(k)
f←i_k-1
for j=k-1,1,-1
        f←f*n_j+i_j-1
repeat
f←f+1
write f
stop
end
```

```
procedure inversa
integer k,j,n(k),i(k),f
read k,n(k),f
x←f-1
for j=1,k-1
        y←[x/n_j]
        i_j←x-y*n_j+1
        x←y
repeat
i_k←x+1
write i(k)
stop
end
```

Exercițiu. Să se scrie câte o procedură pentru calculul funcțiilor g și g^{-1}, corespunzătoare ordinii lexicografice.

2.2 Structuri dinamice [37, 18, 38, 90]

Mulțimea, ca noțiune fundamentală este la fel de importantă în informatică precum și în matematică. Spre deosebire de matematică unde mulțimile sunt statice (nemodificabile), în informatică, mulțimile manipulate de algoritmi, pot crește, descrește sau se pot modifica în timp. Astfel de mulțimi se numesc ***dinamice***. Operațiile specifice mulțimilor dinamice sunt inserare de elemente, ștergere de elemente, testarea apartenței unui element la o mulțime, etc. Astfel, implementarea unei mulțimi depinde de operațiile pe care trebuie să le execute.

https://biblioteca-digitala.ro / https://unibuc.ro

Într-o implementare tipică a unei mulțimi dinamice, fiecare element este reprezentat de un obiect ale carui câmpuri pot fi examinate și manipulate dacă avem o referință (pointer) la acel element.

Unele categorii de mulțimi dinamice presupun că un câmp al obiectului este un câmp cheie pentru identificare. Dacă toate cheile sunt diferite, mulțimea dinamică poate fi privită ca o mulțime de valori ale cheilor.

Unele mulțimi dinamice presupun că valorile cheilor sunt alese dintr-o mulțime total ordonată cum ar fi R sau mulțimea tuturor cuvintelor aflate în ordine lexicografică. O ordine totală ne permite să definim cel mai mic element din mulțime, să vorbim despre următorul element mai mare decât un element dat în mulțime, etc.

2.2.1 Operații pe mulțimi dinamice [38, 44, 69, 90]

Acestea sunt de două categorii:

- de tip interogare, care returnează informații despre mulțime
- de tip modificare, care actualizează mulțimea.

Operațiile uzuale sunt:

1. **caută (S,k)** este o interogare în care pentru o mulțime S și o valoare k, returnează un pointer x la un element din S astfel încât cheie[x]=k sau NIL dacă nu există un astfel de element în S.

2. **inserează (S,x),** operație de modificare, ce adaugă la mulțimea S elementul referit de x.

3. **șterge (S,x),** operație de modificare ce pentru un pointer x la un element din mulțimea S, șterge elementul referit.

4. **minim(S),** o interogare pe mulțimea total ordonată S care returnează elementul din S cu cea mai mică valoare a cheii.

5. **maxim(S),** o interogare pe mulțimea total ordonată S care returnează elementul din S cu cea mai mare valoare a cheii.

6. **succesor(S,x),** o interogare ce pentru un element x a cărui cheie este din mulțimea total ordonată S , returnează elementul următor, mai mare decât x din S sau NIL dacă x este elementul maxim.

7. **predecesor(S,x)** o interogare ce pentru un element x a cărui cheie este din mulțimea total ordonată S , returnează elementul anterior, mai mic decât x din S sau NIL dacă x este elementul minim.

https://biblioteca-digitala.ro / https://unibuc.ro

În continuare vom examina reprezentarea mulțimilor dinamice ca structuri de date simple care folosesc referințe printre care structurile elemetare: liste, stive, cozi.

2.2.2 Liste înlănțuite [37, 69, 90]

O listă înlănțuită este o structură de date, în care obiectele sunt aranjate într-o anumită ordine.

Spre deosebire de tablou, în care ordinea este determinată de indicii tabloului, ordinea într-o listă înlănțuită este determinată de un pointer conținut în fiecare obiect. Listele înlănțuite asigură o reprezentare mai simplă și mai flexibilă pentru mulțimi dinamice suportând toate operațiile descrise în paragraful precedent.

O listă înlănțuită poate fi: simplu sau dublu înlănțuită, sortată sau nesortată, circulară sau liniară.

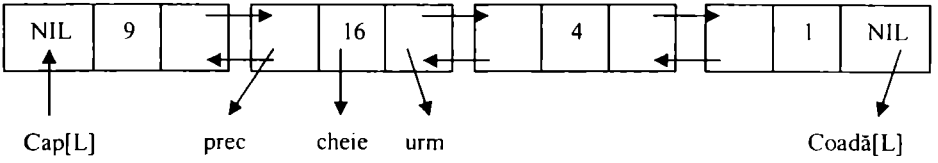
Capetele unei liste se numesc *bază* sau *cap* (primul element al listei) și respectiv *coadă* sau *vârf* (ultimul element al listei).

Într-o listă simplu înlănțuită, fiecare element al listei este alcătuit dintr-un câmp *cheie* (valoarea elementului) urmat de un câmp pointer sau referință numit *urm* (următor).

Pentru un element x din listă, urm[x] indică spre succesorul său din lista înlănțuită. Dacă urm[x]=NIL atunci elementul x nu are succesor și deci este vârful sau coada listei.

Atributul cap[L] referă primul element al listei. Dacă, cap[L]=NIL atunci lista este vidă.

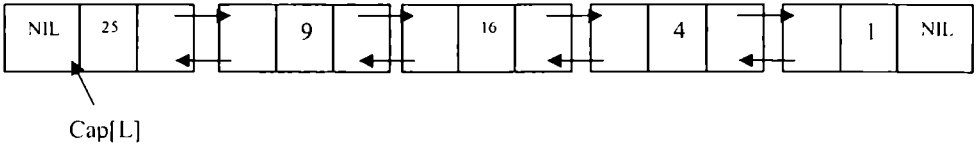
Într-o listă dublu înlănțuită L, fiecare element este un obiect alcătuit dintr-un câmp cheie (valoarea elementului), încadrat de două câmpuri pointer: *prec* (precedent) și *urm* (următor). Pentru elementul x din L, prec[x] indică spre predecesorul său. Dacă prec[x]=NIL înseamnă că x nu are nici un predecesor și deci este baza sau capul listei. Câmpul pointer *urm* are aceeași semnificație ca la o listă simplu înlănțuită. Un exemplu de listă dublu înlănțuită este cel din figura de mai jos:



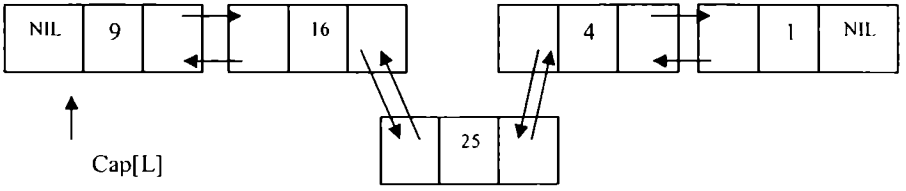
https://biblioteca-digitala.ro / https://unibuc.ro

Se observă că alocarea înlănțuită consumă mai multă memorie decât alocarea secvențială sub formă de vector dar elimină necesitatea deplasării de elemente pentru operațiile de introducere și scoatere de elemente în și din listă.

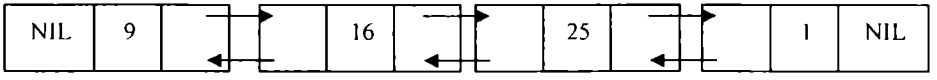
După operația listă_inserează[L,x] unde cheie[x]=25, capul listei înlănțuite este un nou obiect având cheia 25. Acest nou obiect referă vechiul cap, cel ce are cheia 9. Lista de mai sus are acum următoarea formă:



Dacă dorim inserarea aceluiași element între obiectele cu cheile 4 și 16 procedăm ca mai jos:



Rezultatul apelului listă_șterge(L,x), unde cheie[x]=4 și L este ultima listă, va fi cel dat mai jos:



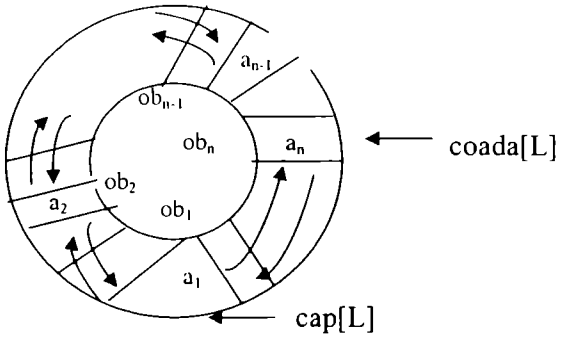
Lista dublu înlănțuită permite refacerea cu ușurință a drumului invers de la cap la bază.

Dacă o listă este sortată, ordinea din listă corespunde ordinii cheilor memorate în elementele listei; elementul minim este capul listei, iar elementul maxim este coada listei.

Dacă lista este nesortată elementele sale pot apărea în orice ordine.

Într-o listă circulară referința *prec* a capului listei referă coada listei, iar referința *urm* a cozii listei referă capul său. Astfel, lista poate fi privită ca un inel de elemente și arată sub forma următoare

https://biblioteca-digitala.ro / https://unibuc.ro



Aici L este o listă circulară, dublu înlănțuită cu n elemente.

În cele ce urmează vom lucra numai cu liste dublu înlănțuite nesortate.

Căutarea într-o listă înlănțuită

Procedura Listă_caută(L,k) găsește primul element ce are cheia k din lista L printr-o căutare liniară simplă, returnând pointerul la acel element. Dacă în listă nu apare nici un element cu cheia k, procedura returnează NIL.

```
Listă_caută(L,k)
x←cap[L]                    (referă primul element al listei)
while (x≠NIL and cheie[x]≠k) do
x←urm[x]
return x
```

Exemplu Pentru lista dublu înlănțuită de la sfârșit, apelul Listă_caută (L,25), returnează un pointer spre al treilea element iar apelul Lista_cauta (L,4) returnează NIL.

Complexitatea temporală a procedurii Listă_cauta pentru o listă cu n elemente este în cazul cel mai nefavorabil O(n), deoarece trebuie să caute în toată lista)a se vedea capitolul IV).

Inserarea într-o listă înlănțuită

Fiind dat un element x al cărui câmp cheie a fost inițializat , procedura Listă_inserează îl plasează pe x în fața listei.

Listă_inserează(L,x)

https://biblioteca-digitala.ro / https://unibuc.ro

urm[x]←cap[L] (următorul element al lui x este cel care a fost primul în L)
if cap[L]≠NIL then
prec[cap[L]]←x (precedentul elementului care a fost primul în L este acum x)
cap[L]←x (primul element al listei este x)
prec[x]←NIL

Complexitatea temporală a acestei proceduri, pentru o listă cu n elemente, este evident O(1).

Ştergerea dintr-o listă

Procedura Listă_şterge elimină un element x dintr-o listă L. Pentru aceasta trebuie transmis ca argument un pointer spre x şi x va fi scos din listă prin actualizarea pointerului. Dacă dorim ştergerea unui element cu o cheie dată, mai întâi trebuie să apelăm procedura Listă_caută pentru a afla pointerul spre acel element.

Listă_şterge (L,x)
if prec[x]≠NIL then
urm[prec[x]]←urm[x]
else
cap[L]←urm[x]
if urm[x]≠NIL then
prec[urm[x]]←prec[x]

Complexitatea temporală a acestei proceduri este O(1). Dacă dorim ştergerea unui element având o cheie dată, timpul necesar pentru cazul cel mai defavorabil este O(n), deoarece va trebui mai întâi să apelăm procedura Listă_caută.

Santinele

Codul pentru Listă_şterge ar fi mai simplu dacă am ignora condiţiile extreme de la capul şi coada listei.

Obţinem:

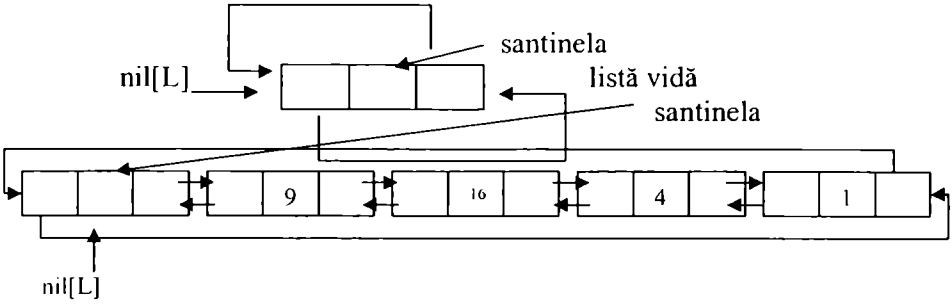
Listă_şterge (L,x)
urm[prec[x]]←urm[x]
prec[urm[x]]←prec[x]

O *santinelă* este un obiect fictiv care permite simplificarea condiţiilor de la extreme.

https://biblioteca-digitala.ro / https://unibuc.ro

De exemplu, să presupunem că în lista L avem la dispoziție un obiect nil[L], care reprezintă NIL, dar care are toate câmpurile unui element al listei. De câte ori avem o referință la NIL, în cadrul listei, o vom înlocui cu o referință la santinela nil[L]. Santinela nil[L] plasată între capul și coada unei liste dublu înlănțuite, transformă această listă într-una circulară; câmpul urm[nil[L]] indică spre capul listei, iar prec[nil[L]] indică spre coada listei. Analog, atât câmpul *urm* al cozii, cât și campul *prec* al capului indică spre nil[L]. Deoarece, urm[nil[L]] indică spre cap[L], putem elimina atributul cap[L], înlocuind referințele către el cu referința către urm[nil[L]]. O listă vidă va conține doar santinela, deoarece atât urm[nil[L]] cât și prec[nil[L]] pot fi setate pe nil[L].

În figura următoare avem o listă dublu înlănțuită ce folosește o santinelă nil[L] pusă între cap și coadă, devenind astfel listă circulară.



Codul pentru Listă_Caută rămâne același dar cu referințele spre NIL și cap[L] schimbate cum s-a precizat mai sus.

```
Listă_caută(L,k)
x←urm[nil[L]]
while (x≠nil[L] and cheie[x]≠k) do
                    x←urm[x]
return x
```

Listă_șterge (L,x) are formatul cu doua linii prezentat mai sus.

Pentru inserarea unui element x folosim acum următoarea procedură:

```
Listă_inserează(L,x)
urm[x]←urm[nil[L]]
prec[urm[nil[L]]]←x
urm[nil[L]]←x
```

https://biblioteca-digitala.ro / https://unibuc.ro

prec[x]← nil[L]

Folosirea santinelelor aduce un plus de claritate în codul procedurilor fără să reducă prea mult din complexitatea temporală.

De exemplu, codul pentru lista înlănțuită din procedurile Listă_inserează și Listă_șterge este simplificat prin folosirea santinelelor dar timpul câștigat este O(1).

2.2.3 Stive și cozi [37, 18, 17, 90]

Stivele și cozile sunt mulțimi dinamice în care elementele ce se elimină din mulțime prin operația **șterge** este prespecificat.

Într-o **stivă** elementul șters din mulțime este elementul cel mai recent inserat. Stiva implementează principiul LIFO (last-in, first-out).

Analog, într-o **coadă**, elementul șters este totdeauna cel care a stat în mulțime cel mai mult (primul introdus). Coada implementează principiul FIFO (first-in, first-out).

Stivele și cozile se pot implementa prin mai multe modalități eficiente printre care: alocarea secvențială, alocarea înlănțuită, etc. În cele ce urmează, vom arăta cum putem folosi tablourile simple pentru a implementa fiecare din aceste structuri.

2.2.3.1 Stive [90]

Stiva este o structură de date la care introducerea și eliminarea elementelor se face prin vârful stivei. Stiva fără elemente se numește stivă **vidă**.

Memorarea unei stive se face prin alocare secvențială, alocare înlănțuită, etc. Indiferent de metodă, memorarea unei stive presupune alocarea unui spațiu limitat, care dacă este ocupat în întregime, spunem că **stiva** este **plină**.

O stivă cu n elemente poate fi implementată printr-un tablou S[1...n]. Tabloul are un atribut varf[S] care este indicele celui mai recent element introdus. Stiva constă din elementele S[1..vârf[S]], unde S[1] este elementul de la baza stivei, iar S[vârf[S]] este elementul din vârful stivei.

Atunci când varf[S]=0, stiva nu conține nici un element și deci este vidă. Se poate testa dacă stiva este vidă prin interogarea **stivă_vidă**.

Dacă se încearcă extragerea (se apelează operația scoate_din_stivă) unui element dintr-o stivă vidă spunem că are loc o *depășire inferioară*,

https://biblioteca-digitala.ro / https://unibuc.ro

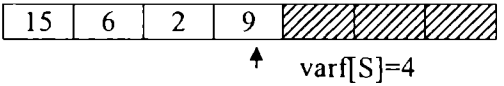
care în mod normal este eroare. Dacă vârf[S] depăşeşte pe n are loc o *depăşire superioară*. Operaţiile cu stive sunt: operaţia **Inserează** asupra stivelor numită pune_în_stivă, şi operaţia **Şterge** numită şi scoate_din_stivă.

```
Stivă_vidă(S)
if varf[S]=0 then
                    return Adevarat
          else
                    return fals

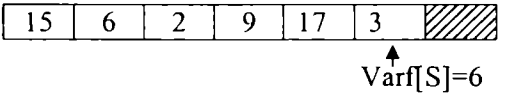
Pune_în_stivă(S,x)
varf[S]←varf[S]+1  (nu ne-am pus problema depăşirii superioare)
S[varf[S]←x

Scoate_din_stivă(S)
if stivă_vidă(S) then
                    return eroare "depăşire inferioară"
               else
                    varf[S]←varf[S]-1
return S[varf[S]+1]
```

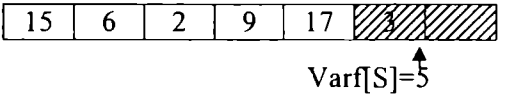
Exemplu: Fie S o stivă cu 4 elemente ca în desenul de mai jos. Elementul din vârful stivei este 9.



După apelurile Pune_în_stivă(S,17) şi apoi Pune_în_stivă (S,3), stiva va avea forma:



După apelul procedurii Scoate_din_stivă(S), obţinem ca rezultat elementul 3 (cel mai recent introdus), care după cum se vede mai jos, deşi apare în tabel, el nu mai este în stivă, elementul din vârful stivei fiind 17



https://biblioteca-digitala.ro / https://unibuc.ro

2.2.3.2 Cozi [87, 53, 90]

Coada este o listă la care introducerea elementelor se face pe la baza listei, iar scoaterea lor se face pe la vârful listei. Numim Pune_în_coadă operația **Inserează** aplicată asupra unei cozi, iar operația Șterge o vom numi scoate_din_coadă. Asemănător operației Scoate_din_stivă aplicată unei stive, scoate_din_coadă nu necesită nici un argument.

Principiul FIFO, propriu cozii, impune ca aceasta să opereze asemănător ca un rând de oameni ce așteaptă la un ghișeu. Coada are un cap (vârf) și o coadă (bază). Când un element este pus în coadă, el ocupă locul de la sfârșitul cozii, ca și un nou venit ce își ia locul la coada rândului de la ghișeu.

Elementul scos din coadă este totdeauna cel din capul cozii, asemănător persoanei din capul rândului care a așteptat cel mai mult.

Figura de mai jos ilustrează o modalitate de a implementa o coadă cu cel mult n-1 elemente, folosind un tablou Q[1,...,n]. Coada are un atribut cap[Q] care conține indicele capului ei. Atributul coadă[Q] conține indicele noii locații în care se va insera în coadă elementul nou venit. Elementele din coadă se află în locațiile:

cap[Q], cap[Q]+1,....,coadă[Q]-1, iar indicii se parcurg circular, în sensul că locația 1 urmează imediat după locația n.

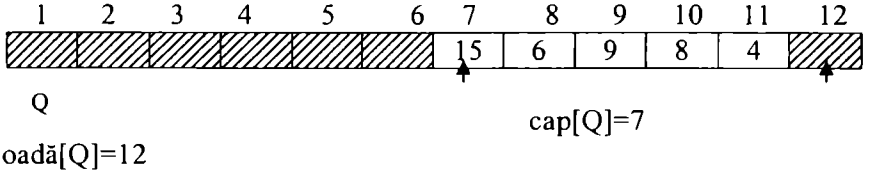
Când cap[Q]=coadă[Q], coada este vidă.

Inițial, avem cap[Q]=coadă[Q]=1.

Încercarea de a scoate un element dintr-o coadă vidă cauzează depășirea inferioară în coadă.

Când coadă[Q] = n+1, coada este plină și o încercare de a pune în coadă cauzează o depășire superioară în coadă.

Exemplu. Fie coada din figura următoare în care n=12.



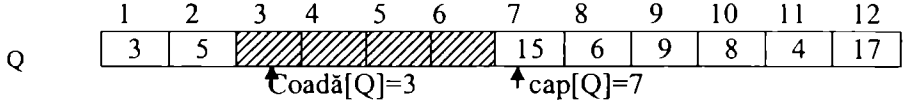
https://biblioteca-digitala.ro / https://unibuc.ro

În acest exemplu implementăm o coadă folosind un tablou Q[1,...,12]. Elementele cozii apar doar în pozițiile nehașurate (locațiile Q[7..11]). Operațiile Pune_în_coadă și Scoate_din_coadă, fără verificarea erorilor de depășire, au codurile următoare:

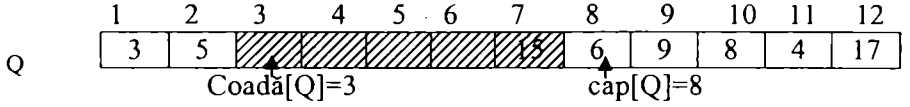
```
Pune_în_coadă(Q,x)
Q[coadă[Q]]←x
if coadă[Q]=lung[Q] then
                    coadă[Q]←1
                    else
                    coadă[Q]←coadă[Q]+1

Scoate_din_coadă[Q]
x←Q[cap[Q]]
if cap[Q]=lung[Q] then
                    cap[Q]←1
                    else
                    cap[Q]←cap[Q]+1
                    return x
```

Configurația cozii de mai sus după apelurile: Pune_în_coadă(Q,17), Pune_în_coadă(Q,3) și Pune_în_coadă(Q,5) este:



Configurația acestei cozi după apelul Scoate_din_coadă(Q) ce returnează valoarea cheii 15, este următoarea:



Deși în această configurație cheia 15 apare în tabel, ea nu mai face parte din coadă. Noul cap este acum cel ce are cheia 6.

https://biblioteca-digitala.ro / https://unibuc.ro

Observație. În timp ce stiva permite inserarea și ștergerea de elemente doar de la un singur capăt, coada permite inserarea la un capăt și ștergerea la celălalt capăt. Această modalitate de introducere și scoatere de elemente în și din coadă fac ca aceasta să migreze spre dreapta, ajungând la situația absurdă când deși coada este vidă, (cap[Q]=coadă[Q]=n+1) să nu putem introduce elemente în ea. Pentru a elimina acest neajuns, se consideră într-o coadă că după locația de ordin n urmează prima locație. În acest caz, relația cap[Q]=coadă[Q] este verificată atât pentru o coadă vidă, cât și pentru o coadă plină. Pentru a distinge cele două cazuri, folosim efectiv n-1 poziții în vectorul Q[1..n] și astfel coada vidă se identifică prin cap[Q]=coadă[Q] iar coada plina prin cap[Q]=coadă[Q]+1 (mod n).
Există și **coadă completă** care permite inserări și ștergeri de elemente la ambele sale capete.

2.3 Descompunerea problemelor în subprobleme – modularizare [8, 9, 55, 90]

În multe cazuri este util ca algoritmul de rezolvare a unei probleme să fie descompus în mai multe părți, iar operațiile elementare necesare fiecărei părți să fie programate separat. La nivelul unui limbaj de programare, acest lucru se realizează prin folosirea de subprograme.

Limbajele algoritmice se deosebesc de limbajele de programare printr-o sinteză mai puțin strictă, prin utilizarea textelor înlocuitoare, care contin o acțiune ce urmează a fi descrisă în limbaj algoritmic, printr-o mare libertate de scriere a instrucțiunilor limbajului și prin acceptarea unor abateri de la regulile de scriere a instrucțiunilor cât timp semnificațiile variabilelor și instrucțiunilor rămân clare.

Textele înlocuitoare se scriu între paranteze drepte, ca in exemplul următor:

i←[cel mai mare număr prim mai mic decât k]

Până la acest moment prin program s-a înțeles fie o secvență de instrucțiuni ca în schemele logice sau în descrierea algoritmilor în limbaj natural, fie o secvență de proceduri ca în limbajele algoritmice, cum ar fi pseudocodul.

În limbajele de programare, un program este format dintr-o procedură principală sau program principal urmată eventual de unul sau mai

https://biblioteca-digitala.ro / https://unibuc.ro

multe subprograme care au aceeaşi structură ca şi procedurile din pseudocod.

Programul principal are forma generală ca a unei proceduri din pseudocod cu singura deosebire că începe cu cuvântul rezervat **program** (în Pascal) sau **main** (în C, C++) în loc de **procedure** din pseudocod.

Necesitatea subprogramelor rezultă şi din aceea că de multe ori algoritmul prevede executarea unor aceleaşi calcule pentru seturi diferite de date. Subprogramele permit scrierea acestor calcule o singură dată mărind astfel claritatea programelor.

Exemplu.

Pentru variabila indexată $a=(a_1, a_2, ..., a_k, a_{k+1}, a_n)$ şi $k=1,...,n-1$ date, să se scrie un subprogram care să transforme această variabilă încât să aibă forma $a=(a_{k+1},a_{k+2},...,a_n,a_1,...,a_k)$.

Soluţie.

O metodă simplă şi eficientă de rezolvare necesită următoarele operaţii:

1. Se inversează ordinea elementelor din $(a_1, a_2,...,a_k)$
2. Se inversează ordinea elementelor din $(a_{k+1},...,a_n)$
3. Se inversează ordinea elementelor din a.

Observăm că operaţia de inversare a ordinii elementelor unui vector se repetă de 3 ori în acest program. Este deci necesar un subprogram care să realizeze această operaţie.

Există limbaje de programare în care subprogramele sunt de două tipuri: proceduri şi funcţii, cum ar fi limbajul Pascal, şi există limbaje în care procedurile se numesc funcţii, cum ar fi C, C++, TC, BC etc.

Pentru primul caz, o procedură este formată din instrucţiunea **procedure**, urmată de declarări de variabile, apoi de o secvenţă de instrucţiuni efective şi încheiată prin instructţiunea **end**, care delimitează sfârşitul procedurii.

Instrucţiunea procedurală are una din formele:

procedure nume
procedure nume (listă_parametri)

în care elementele au semnificaţia din pseudocod. Parametrii servesc la transmiterea de informaţii între diferite unităţi de program (program principal şi subprograme).

Apelarea unei proceduri se face prin instrucţiunea de apel cu una din formele:

nume
nume (listă_argumente)

https://biblioteca-digitala.ro / https://unibuc.ro

unde *nume* este identificatorul procedurii apelate, iar argumentele sunt expresii despărțite prin virgulă.

Un program se execută începând cu executarea instrucțiunilor programului principal. Când se ajunge la o instrucțiune de apel a unei proceduri se întreprind următoarele acțiuni:

- se stabileşte corespondența între argumente şi parametri
- se execută instrucțiunile procedurii apelate până se ajunge la instrucțiunea **return** al cărei efect este întoarcerea la unitatea de program în care a avut loc apelul şi anume la instrucțiunea imediat următoare celei de apel. Un subprogram poate apela la rândul lui un alt subprogram. Executarea instrucțiunii **stop** într-un subprogram termină executarea întregului program.

Un apel de procedură este corect dacă între argumente şi parametri există o corespondență biunivocă atât ca număr, cât şi ca tip şi ordine. În acest caz, legătura între argumente şi parametri se poate face prin metoda **apelului prin referință** , în care se transmit adresele de memorie ale argumentelor şi nu valorile lor.

Schimbul de informații între diferite unități de program se face prin corespondența argumente↔parametri ce are loc la apelul subprogramelor.

Astfel, dacă unitatea U apelează unitatea S şi v este o variabilă comună a celor două unități de program dar nu apare în lista parametrilor lui S, atunci cele două nume v din U şi S se consideră ca fiind două variabile distincte, ocupând locații de memorie diferite.

Anumite proceduri sunt deja cunoscute de limbajul folosit, purtând denumirea de **proceduri predefinite**. Aceste proceduri nu mai trebuie declarate. Exemple ar fi operațiile de citire-scriere care pot fi privite ca fiind apeluri de proceduri predefinite.

Exemplu. Reluăm exemplul anterior şi scriem un program principal care citeşte k,n şi a(n), apelează o procedură pentru inversarea ordinii elementelor unui vector, numită **invers** şi apoi afişează vectorul modificat. Procedura invers are antetul *procedure invers(a,n,p,m)* şi ea inversează ordinea elemetelor vectorului a(n) de pe pozițiile p,p+1,...,m.

Soluție.

```
program vector;
integer n,k; real a(n);
read n,k,a(n)
invers(a,n,1,k); invers(a,n,k+1,n); invers(a,n,1,n)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
write a(n)
stop
end
procedure invers(a,n,p,m)
integer i,j
i←p; j←m
while i<j
  a_i↔a_j; i←i+1; j←j-1
  repeat
  return
  end
```

Observație. Subprogramele de tip procedură se folosesc atunci când avem de obținut ca rezultat mai multe valori, așa cum ar fi cazul în care avem de modificat valorile memorate într-un vector.

O **funcție** este formată din instrucțiunea **function** urmată de declarări de variabile, apoi de secvențe de instrucțiuni efective și încheiată cu **end** care delimitează sfârșitul funcției. Instrucțiunea **function** are una din formele:

function nume
function nume(listă_parametri)

unde elementele au aceeași semnificație ca în pseudocod.
Funcțiile diferă de proceduri prin următoarele:

- numele funcției servește nu numai la identificarea subprogramului de tip funcție, ci și la memorarea unei valori de tip integer, real, char sau bool.
- în corpul funcției, variabila **nume** trebuie să primească cel puțin o valoare, fie printr-o citire, fie printr-o atribuire.
- apelul funcției se face prin apariția lui **nume**, urmat eventual de o listă de argumente cuprinse între paranteze rotunde, în cadrul unei expresii. La încheierea executării funcției, valoarea curentă a numelui funcției va fi folosită la evaluarea expresiei respective.

Anumite funcții sunt deja cunoscute de limbajul folosit și se numesc **funcții predefinite**. Acestea nu mai trebuie declarate. Exemplu, calculul valorii absolute a unui număr se execută la apelul funcției denumită **abs**.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu: Fie a(n), b(n) vectori cu n≥1 componente reale. Să se determine produsul scalar a·b.

Soluție: Vom scrie un program format dintr-un program principal care citește n,a(n) și b(n) și apoi afișează valoarea produsului scalar a·b, și dintr-un subprogram de tip funcție numită **produs** care realizează efectiv produsul scalar, folosind formula $a \cdot b = \sum_{i=1}^{n} a_i \cdot b_i$

```
program scalar                    p←0
integer n; real a(n),b(n)         for i=1,n
read n,a(n),b(n)                        p←p+a_i*b_i
write produs(a,b,n)               repeat
stop                              produs←p
end                               return
function produs (x,y,n)           end
integer i; real p,x(n),y(n);
```

Observație. Variabilele declarate în programul principal se numesc **variabile globale**, iar cele declarate în partea de declarații a unui subprogram de tip procedură sau funcție, se numesc variabile locale. În exemplul de mai sus, n, a(n) și b(n) sunt variabile globale, iar i și p sunt **variabile locale**.

Pentru a înțelege comportamentul unui program vis-a-vis de parametrii și variabilele sale, folosim noțiunea de **bloc**. Astfel, întregul program formează un bloc numit blocul programului principal.

Fiecare funcție sau procedură din acest bloc definește un alt bloc al său care începe cu **procedure** sau **function** și se încheie cu end.

Fiecare astfel de bloc are la începutul său o parte de declarații în care folosim diverse nume pentru a identifica variabilele blocului, numele și parametrii subprogramului.

Fiecare identificator are un așa numit **domeniu de vizibilitate**, adică o zonă de program în care, ori de câte ori apare, este recunoscut de compilator conform cu declarația sa. Regula este următoarea: "domeniul de vizibilitate al unui identificator îl reprezintă cel
mai exterior bloc în care acest identificator este declarat, neluând în seamă acele subblocuri în care el este redeclarat."

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu. În figura urmatoare avem un program compus din cinci blocuri.

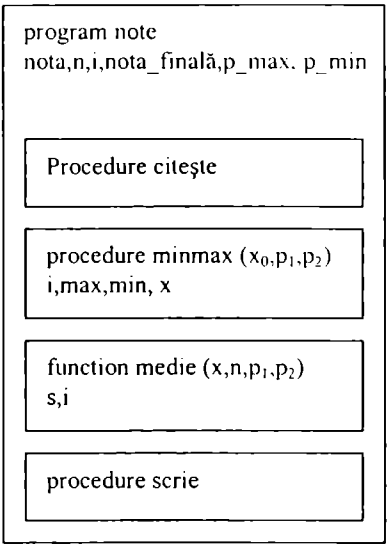
Procedurile citește și scrie nu au parametri. În acest caz calculele respective se fac direct asupra variabilelor globale.

În procedura **minmax** redeclarăm n ca fiind parametru, iar i ca fiind variabilă locală. Tot aici mai avem variabilele locale x, max și min.

În funcția **medie** redeclarăm x și n ca parametri. Tot aici avem parametrii p_1 și p_2 , iar ca variabile locale, s și i care este redeclarat.

Astfel, parametrul x va fi vizibil în procedura minmax și în funcția medie. Variabila s va fi vizibilă doar în funcția medie.

Variabila globală n va fi recunoscută în programul principal cât și în procedurile de citire și scriere.



Atunci, când într-un program procedurile și funcțiile sunt scrise una după alta și apelul lor se face de fiecare dată după momentul declarării, spunem că avem o **dezvoltare ascendentă** de jos în sus (sau bottom-up) a subprogramelor. Modalitatea rafinării calculelor de sus în jos (top-down) într-un program poartă numele de **dezvoltare descendentă** a subprogramelor.

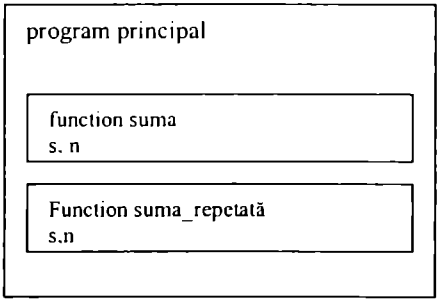
Combinând aceste două metode de programare, putem imagina pe cazul general, scrierea mai multor blocuri, fiecare dintre acestea, cu excepția blocului corespunzător programului principal având alt bloc drept "părinte". Se obține astfel o structură de blocuri sub forma unui arbore ce are în rădăcină blocul programului principal. Ca regulă generală, apelul unui subprogram (bloc) se poate face numai din blocul părinte sau din blocurile de după respectivul bloc ce au același bloc părinte (blocurile declarate pe același nivel dar după respectivul bloc).

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu Pentru n∈N* să se tipărească cifra obținută prin însumarea repetată a cifrelor acestuia până se ajunge la un număr de o cifră.

Soluție 1: (dezvoltare ascendentă)

```
program suma1
integer n
function suma(n)
integer s
s←0
while n≠0
  s←s+n mod 10
  n←n div 10
repeat
suma←s
return
end
function suma_repetata(n)
integer s
do
  s←suma(n)
  n←s
until n≤9
suma_repetata←n
return
end
read n
write suma_repetată(n)
stop
end
```

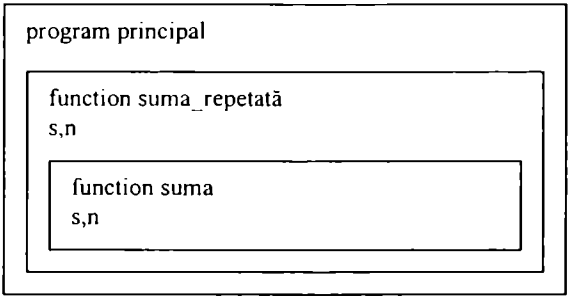


Aici am pus în evidență apelul unui subprogram din cadrul altui subprogram. Se observă că pentru a putea apela funcția suma din cadrul funcției suma_repetată, prima funcție trebuie scrisă în program înaintea celeilalte.

https://biblioteca-digitala.ro / https://unibuc.ro

Soluție 2. (dezvoltare descendentă)

```
program suma_2
integer n
function suma_repetată(n)
integer s
function suma(n)
integer s
s←0
while n≠0
  s←s+n mod 10
  n←n div 10
repeat
suma←s
return
end
do
s←suma(n)
n←s
until n≤9
suma_repetată←n
return
end
read n
write suma_repetată (n)
stop
end
```



Împreună cu programul principal sunt aici prezente 3 blocuri incluse unul în altul cum se vede în figura alăturată.

Avantajele folosirii subprogramelor

Folosind subprograme într-un program obținem o mai mare claritate a întregului program. Problema inițială se împarte în subprobleme mai simple, fiecare subprogram rezolvând un anumit pas al întregului algoritm.

Pe de altă parte, odată scris și testat un subprogram, ne va interesa doar ce face subprogramul, nu și cum face.

Prin utilizarea corectă a parametrilor, subprogramul devine independent de restul programului. El poate fi folosit fără modificări în alte programe care necesită un același pas de prelucrare a datelor. Prosibili-

https://biblioteca-digitala.ro / https://unibuc.ro

tatea apelării unui subprogram din mai multe locuri conduce la o economie de scriere a codului sursă al întregului program.

Prin folosirea subprogramelor şi mai ales prin folosirea metodei de dezvoltare descendentă a subprogramelor, este posibilă funcţionarea eficientă a lucrului într-o echipă de programatori. Unora li se va comunica doar funcţiile şi procedurile ce trebuie scrise, precum şi identificatorii globali: tipuri de date, identificatori ai variabilelor globale precum şi ai funcţiilor şi procedurilor respective.

2.4 Codificarea informaţiilor în memoria calculatoarelor Electronice [37, 90]

2.4.1 Sisteme de numeraţie [90]

Atât în faza de introducere a datelor în memoria unui calculator, cât şi în faza extragerii rezultatelor unei prelucrări este necesară o conversie a acestora într-o formă inteligibilă pentru calculator pentru primul caz, deoarece calculatorul înţelege doar limbajul binar, sau într-o formă accesibilă utilizatorului, pentru cazul al doilea.

Definiţie: Numim ***sistem de numeraţie*** o mulţime de simboluri însoţită de o mulţime de reguli de folosire a acestora pentru reprezentarea informaţiilor în memoria unui calculator.

În calculator se lucrează cu sisteme de numeraţie poziţionale, în care o cifră are semnificaţia dată de poziţia pe care o ocupă în numar şi de valoarea ce se atribuie fiecărei poziţii.

Numărul simbolurilor folosite de un sistem de numeraţie poziţional se numeşte *bază de numeraţie*.

Exemple.

- sistemul zecimal foloseşte cifrele 0÷9, deci baza=10
- sistemul binar foloseşte cifrele 0÷1, deci baza=2
- sistemul octal foloseşte cifrele 0÷7 deci baza=8
- sistemul hexazecimal foloseşte cifrele 0÷9, A, B, C, D, E, F şi astfel baza=16.

Calculatoarele lucrează cu un număr de stări stabile distincte, număr egal cu baza sistemului de numeraţie folosit. Se preferă un număr de stabile cât mai mic de tip închis-deschis, adică stări bistabile. De aici şi necesi-

https://biblioteca-digitala.ro / https://unibuc.ro

tatea folosirii sistemului binar şi al derivatelor sale octal, hexazecimal, etc.

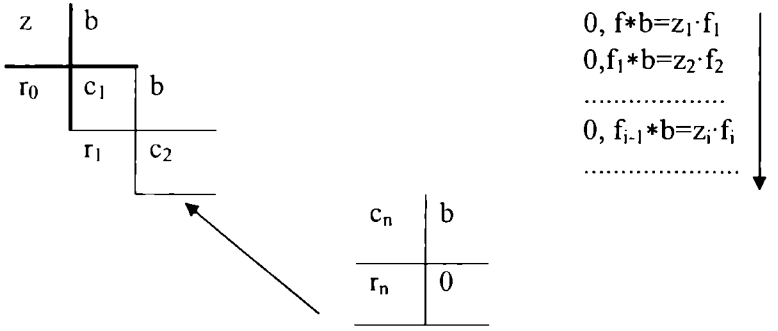
2.4.2 Conversia unui număr real din sistemul zecimal într-un sistem de bază dată şi invers

2.4.2.1 Conversia din zecimal în sistemul cu baza b≥2

Fie x=z. f ∈R şi b∈N, b≥2. Pentru conversia lui x în baza b se procedează astfel:

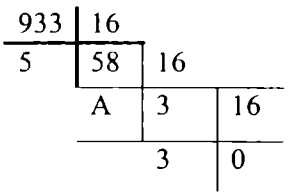
- partea întreagă z şi câturile c_1, c_2,... se împart succesiv la b până se obţine câtul zero
- partea fracţionară f şi părţile fracţionare f_1, f_2, ... se înmulţesc succesiv cu b până se obţine partea fracţionară zero sau până se obţine un număr specificat de cifre fracţionare.

Acest algoritm se poate schematiza prin:



Astfel, $x=z.f_{(10)} = (r_n r_{n-1}...r_1 r_0 . z_1 z_2...z_i...)_{(b)}$

Exemplu. x=933.25, b=16,



0.25*16=4.0

Astfel, $933.25_{(10)}=3A5.4_{(16)}$

https://biblioteca-digitala.ro / https://unibuc.ro

2.4.2.2 Conversia unui număr din baza b≥2 în zecimal

Fie $x=(r_nr_{n-1}...r_1r_0 \,.\, z_1z_2...z_i)_{(b)}$. El se convertește în zecimal folosind formula:

$$x=r_n\cdot b^n+r_{n-1}\cdot b^{n-1}+...+r_1\cdot b+r_0+z_1\cdot b^{-1}+z_2\cdot b^{-2}+...+z_i\cdot b^{-i}$$

efectuând calculele în baza 10.

Exemplu.

$$x=3AF.C_{(16)} = 3\cdot 16^2+A\cdot 16+F+C\cdot 16^{-1} = 3\cdot 256+10\cdot 16+15+\frac{12}{16}=943+0.75 =$$

$$=943.75_{(10)}.$$

2.4.2.3 Conversia binar→octal, hexazecimal şi invers a unui numar real

a. **conversia octal→binar şi invers** foloseşte faptul că o cifră octală se poate reprezenta prin 3 cifre binare astfel:

0=000	4=100
1=001	5=101
2=010	6=110
3=011	7=110

a_1) **conversia octal→binar**

Fiecare cifră octală se înlocuie cu secvența de 3 cifre binare corespunzătoare. Se elimină apoi toate zerourile din fața părții întregi a numărului şi toate zerourile de la sfârşitul părții fracționare a numărului Obținut $x=(r_nr_{n-1}...r_1r_0 \,.\, z_1z_2...z_i)_{(8)}$ obtinem:

$$x=(\underbrace{b_1b_2b_3}_{r_n}\ \underbrace{b_4b_5b_6}_{r_{n-1}}....\underbrace{b_{3n+1}b_{3n+2}b_{3n+3}}_{r_0} \,.\, \underbrace{c_1c_2c_3}_{z_1}\ \underbrace{c_4c_5c_6}_{z_2}\ ...\underbrace{c_{3i-2}c_{3i-1}c_{3i}}_{z_i})_{(2)}$$

Exemplu. $x=347.5_{(8)}=011100111.101_{(2)}=11100111.101_{(2)}$

a_2) **conversia binar→octal**

Fie $x=(r_nr_{n-1}...r_1r_0 \,.\, z_1z_2...z_i)_{(2)}$

Se grupează de la marca zecimală spre stânga pentru partea întreagă şi spre dreapta pentru partea fracționară, câte 3 cifre binare, completând la nevoie ultima grupă din stânga cu zerouri in fata numarului sau la dreapta cu zerouri după număr, zerouri care nu modifică valoarea numărului

https://biblioteca-digitala.ro / https://unibuc.ro

şi apoi se ia cifra octală corespunzătoare fiecărei grupe obţinute. Astfel, x devine spre exemplu:
$x=(00r_n)(r_{n-1}r_{n-2}r_{n-3})...(r_5r_4r_3)(r_2r_1r_0).(z_1z_2z_3)...(z_i00) =$
$= (a_pa_{p-1}...a_1a_0.b_1b_2...b_k)_{(8)}$, unde $a_0,...,a_p$, $b_1,...,b_k$
sunt cifre octale corespunzătoare grupelor obţinute mai sus.

Exemplu.
$x=1110011101.1011_{(2)}=(001)(110)(011)(101).(101)(100) = 1635.54_{(8)}$.

b)**conversia hexazecimal-binar şi invers** foloseşte faptul că o cifră hexazecimală se poate reprezenta prin 4 cifre binare conform tabelului următor:

Cifră hexazecimală	Secvenţă binară
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

b_1)**conversia hexazecimal-binar**

Fiecare cifră hexazecimală se înlocuie cu secvenţa de 4 cifre binare corespunzătoare. Se elimină apoi toate zerourile din faţa părţii întregi a numărului şi toate zerourile de la sfârşitul părţii fracţionare a numărului obţinut.

Pentru $x=(r_nr_{n-1}...r_1r_0.z_1z_2...z_i)_{(16)}$ avem:

$x=(b_1b_2b_3b_4 \ .... \ b_{4n+1}b_{4n+2}b_{4n+3}b_{4n+4} \ . \ c_1c_2c_3c_4 \ .... \ c_{4i-3}c_{4i-2}c_{4i-1}c_{4i})_{(2)}$.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu. $x=5AF4.3E_{(16)}=0100101011110100.00111110=$
$100101011110100.0011111_{(2)}$

b_2) **conversia binar-hexazecimal**

Fie $x=(r_nr_{n-1}...r_1r_0.z_1z_2...z_i)_{(2)}$.

Se grupează de la marca zecimală spre stânga pentru partea întreagă şi spre dreapta pentru partea fracţionară câte 4 cifre binare, completând la nevoie ultima grupă din stânga cu zerouri în faţa numărului sau la dreapta cu zerouri la sfârşitul numărului, zerouri care nu modifică valoarea numărului şi apoi se ia cifra hexazecimală corespunzătoare fiecărei grupe obţinute. Astfel, x devine spre exemplu:
$x=(000r_n)(r_{n-1}r_{n-2}r_{n-3}r_{n-4}),,,(r_3r_2r_1r_0).(z_1z_2z_3z_4)...(z_{i-1}z_i00) = (a_pa_{p-1}...a_1a_0$
$b_1...b_k)_{(16)}$, unde:

$a_0,...,a_p$, $b_1,...,b_k$ sunt cifre hexazecimale corespunzătoare grupelor obţinute mai sus.

Exemplu.
$x=100110101111100.00111101111_{(2)}=(0010)(0110)(1011)(1100).(0011)(1101)(1110)=26BC.3DE_{(16)}$.

2.4.3 Coduri pentru reprezentarea datelor [90]

Codurile asigură comunicarea dintre utilizator şi calculator. Utilizatorului îi este specifică gândirea şi operarea în sistemul zecimal, iar calculatorului îi este specifică gândirea şi prelucrarea datelor în binar sau în derivatele sale octal şi hexazecimal.

Definiţie. Numim ***cod*** o mulţime de simboluri elementare însoţită de o mulţime de reguli de formare a acestor simboluri.

Prin ***codificare*** înţelegem procesul stabilirii unui cod.

Fie $X=\{x_1, ... , x_n\}$ datele de intrare ale unui algoritm în forma accesibilă utilizatorului şi $Y=\{y_1,...,y_m\}$ mulţimea simbolurilor recunoscute de calculator.
Prin codificare, oricărui x din X i se asociază o secvenţă σ din

$$Y^*=\bigcup_{n\geq 0} Y^n$$

https://biblioteca-digitala.ro / https://unibuc.ro

Codul este orice funcție bijectivă $f:X \rightarrow Y^*$.

Funcția inversă f^{-1}, $f^{-1}:Y^* \rightarrow X$ reprezintă procesul de decodificare a secvențelor de simboluri folosite de calculator și transformarea lor într-o formă accesibilă omului.

În calculator trebuie reprezentate atât numerele, cât și literele alfabetului latin, dar și o mulțime de caractere speciale. Au apărut astfel *codurile alfanumerice* care folosesc un număr de cifre binare, suficient pentru a reprezenta întreaga gamă de caractere: cifre, litere, caractere speciale.

Calculatoarele folosesc coduri alfanumerice cu 7 și cu 8 cifre binare, care pot reprezenta $2^7=128$ și respectiv $2^8=256$ caractere prin combinații distincte de cifre binare care se pot forma pe 7, respectiv 8 poziții**Codul ASCII** (American Standard Code for Information Interchange) este un exemplu de cod cu 7 cifre binare. În acest cod cifrele 0÷9 se codifică prin reprezentările binare ale numerelor 48÷57, literele A÷Z se codifică prin reprezentările binare ale numerelor 65÷90, iar literele a÷z prin reprezentările binare ale numerelor 97÷122.

Reprezentările binare ale celorlalte numere de la 0÷127 sunt folosite pentru codificarea diferitelor caractere speciale.

Codul EBCDIC (Extended Binary Coded Data Interchange Code) este un exemplu de cod cu 8 cifre binare. Aici codificările se fac prin reprezentările binare ale numerelor 0÷255.

Codurile numerice, numite și coduri zecimale codificate binar, reprezintă cifrele sistemului zecimal printr-o secvență de 4 cifre binare. Acestea reduc numărul cifrelor binare necesare reprezentării cifrelor zecimale de la 8 ca un EBCDIC sau 7 ca în ASCII la 4. Cu acestea se realizează $2^4=16$ combinații distincte de cifre binare.

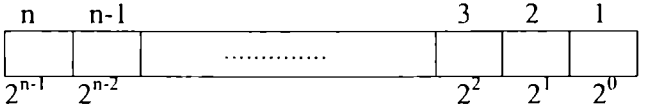
Codurile numerice sunt **ponderate** precum 8421, 2421, etc. sau **neponderate** precum Exces 3, Gray, etc.

2.4.4 Reprezentarea numerelor în calculator [90]

2.4.4.1 Reprezentarea numerelor naturale

Aceasta se mai numește și **reprezentare aritmetică**. Numerele naturale se reprezintă pe un număr fix n de poziții binare, unde n=8,16,32 sau 64. Alinierea în zona rezervată se face la dreapta cu eventuale zerouri nesemnificative plasate în fața numărului

https://biblioteca-digitala.ro / https://unibuc.ro



Dacă numărul pozițiilor binare este n atunci numărul maxim ce se poate reprezenta aritmetic se obține punând 1 în fiecare din cele n poziții binare găsind astfel numărul :

$2^{n-1}+2^{n-2}+...+2+1=2^n-1$.

Astfel, plaja de valori acceptata de reprezentarea aritmetică este funcție de numărul pozițiilor binare folosite și anume este $\{0,1,...,2^n-1\}$.

Pentru n=64 putem reprezenta aritmetic numerele $0,1,2,...,2^{64}-1$, unde $2^{66}-1$ este aproximativ 10^{20}.

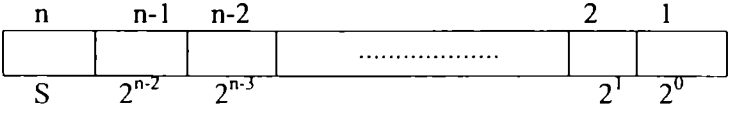
Exemplu. Pentru n=16, numărul 259 se reprezintă pe 16 poziții binare sub forma $259=2^8+2+1$

0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1

2.4.4.2 Reprezentarea numerelor întregi

Aceasta se mai numește și **reprezentarea algebrică**. Ea se aseamănă cu cea aritmetică cu deosebirea că prima poziție binară din zona rezervată este ocupată de semnul S al numărului, unde S=0 pentru numere≥0 și S=1 pentru numere <0.

Reprezentarea numerelor întregi pe n poziții binare are forma:



Intervalul maxim admis de reprezentare este: $-2^{n-1} \div 2^{n-1}-1$. Astfel, cel mai mic întreg ce se poate reprezenta algebric pe n poziții binare este -2^{n-1}, iar cel mai mare este $2^{n-1}-1$.

Pentru n=64 obținem intervalul $-2^{63} \div 2^{63}-1$

Numerele negative se pot reprezenta algebric sub 3 forme:

b_1. **în cod direct**, care coincide cu reprezentarea algebrică având 1 pe poziția semnului.

https://biblioteca-digitala.ro / https://unibuc.ro

b_2. **în cod invers**, obținut prin schimbarea fiecărei cifre binare a reprezentării algebrice a valorii absolute a numărului. Se mai numește și **cod complementar față de 1**.

b_3. **în cod complementar**, obținut prin schimbarea fiecărei cifre binare a reprezentării valorii absolute a numărului și apoi adăugarea cifrei 1 la cifra cea mai din dreapta a reprezentării obținute.. Se mai numește și **cod complementar față de 2**.

Exemplu. Pentru n=16, numărul –259 se reprezintă algebric astfel:

- în cod direct:

1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1

- în cod invers

1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0

- în cod complementar

1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1

2.4.4.3 Reprezentarea numerelor reale

Numărul real x se reprezintă în calculator prin așa numita **reprezentare în virgulă mobilă,** numită și **virgulă flotantă**.
Pentru aceasta, x se aduce la forma normală care este: $x=\pm 1.f * 2^{\pm e}$
unde f este **partea fracționară** a formei normale a lui x.
$\pm 1.f$ este **mantisa** lui x și trebuie să respecte condiția de nomalizare
$1 \le |1.f| < 2$.
$\pm e$ este **exponentul** bazei sistemului binar de numerație.

Exemplu: $25=11001_{(2)}=1.1001*2^4$ (forma normală)

$$\frac{31}{32}=\frac{2^4+2^3+2^2+2^1+1}{2^5}=2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}=0.11111_{(2)}=1.1111*2^{-1}$$

(forma normală).

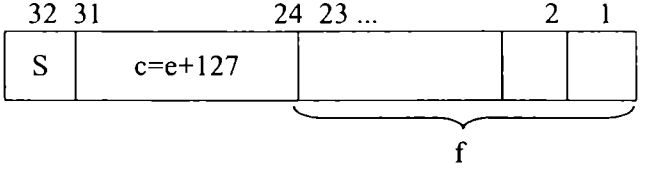
https://biblioteca-digitala.ro / https://unibuc.ro

$-\frac{1}{8} = -2^{-3} = -0.001_{(2)} = -1.0*2^{-3}$, forma normală.

Numerele reale pot fi reprezentate în virgulă mobilă cu **simplă precizie** sau cu **precizie dublă**.

2.4.4.3.1 Reprezentarea numerelor reale în virgulă mobilă cu simplă precizie

În această reprezentare un număr real x ocupă 32 poziții binare sub forma:



unde: S este semnul numărului, ocupă prima poziție din zona rezervată și poate fi 0 pentru număr≥0 sau 1 pentru număr <0.

c se numește **caracteristica** numărului x și ocupă 8 poziții binare. Astfel $c_{max} = 2^8-1 = 255$. Se ia c = e+127 pentru a nu mai fi necesară o poziție binară pentru semnul exponentului.

Din $0 \le c \le 255 \Leftrightarrow 0 \le e+127 \le 255 \Leftrightarrow -127 \le e \le 128$, rezultă $e_{min} = -127$ și $e_{max}=128$.

Ultimele 23 poziții binare sunt alocate partii fracționare aliniata la stânga în zona rezervată, prin completare cu zerouri în dreapta zonei..

Deoarece $2^{23}-1 \approx 10^7$ rezultă că f poate păstra 7 cifre zecimale.

Valoarea maximă a numărului real ce se poate reprezenta pe aceste 32 poziții este aproximativ 10^{38}.

Exemple. $x=25=1.1001*2^4 \Rightarrow$S=0 (deoarece x>0)
f=1001
$c=e+127=131=10000011_{(2)}$.

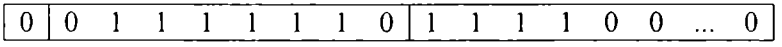
Obținem reprezentarea:

0	1	0	0	0	0	0	1	1	1	0	0	1	0	0	...	0

$x=\frac{31}{32} = 1.1111*2^{-1} \Rightarrow$S=0, f=1111, $c=-1+127=126=01111110_{(2)}$

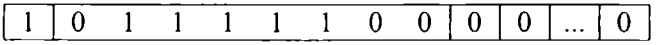
https://biblioteca-digitala.ro / https://unibuc.ro

Obținem reprezentarea:



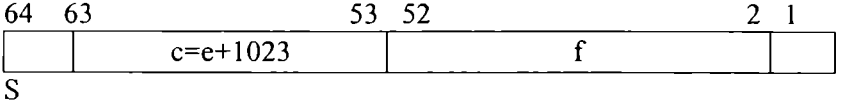
$x=-\frac{1}{8}=-1.0*2^{-3}\Rightarrow S=1,\ f=0,\ c=-3+127=124=01111100_{(2)}$

Obținem reprezentarea:



2.4.4.3.2 Reprezentarea numerelor în virgulă mobilă cu precizie dublă

În această reprezentare un număr real x ocupă 64 poziții binare sub forma:



Caracteristica c se reprezintă pe 11 poziții binare şi deci $c_{max}=2^{11}-1=2047$.
Din $0\le c\le 2047$ rezultă $0\le e+1023\le 2047\Rightarrow -1023\le e\le 1024$.
Astfel, $e_{min}=-1023$ şi $e_{max}=1024$

Partea fracționară f ocupă 52 de poziții binare şi cum $2^{52}-1\approx 10^{15}$ rezultă că în această reprezentare în dublă precizie, f poate păstra până la 15 cifre zecimale.

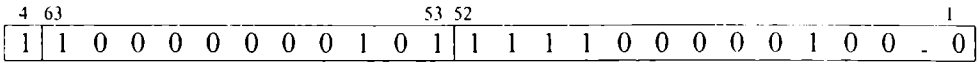
Valoarea maximă a numărului real ce se poate reprezenta pe aceste 64 poziții binare este aproximativ 10^{308}.

Exemple.

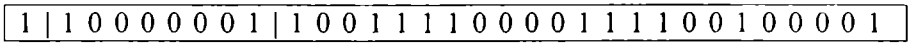
1. $x=-124.0625=-1111100.0001_{(2)}=-1.1111000001*2^6$ sub forma normală.

Astfel, $S=1$, $c=6+1023=1029=10000000101_{(2)}$, $f=1111000001_{(2)}$.
Obținem reprezentarea:

https://biblioteca-digitala.ro / https://unibuc.ro



2. Un număr x în virgulă mobilă are în simplă precizie reprezentarea următoare:



Pentru a obține forma zecimală a lui x observăm:

$S=1 \Rightarrow x<0$

$c=10000001=2^7+1=129 \Rightarrow e=c-127=2$; f=10011110000111100100001

Astfel,

$x = -1.10011110000111100100001*2^2=$

$= -110.01111000011110010000 1= -6.986913_{(10)}$

deoarece $01111000011110010000 1=2^{19}+2^{18}+2^{17}+2^{16}+2^{11}+2^{10}+2^9+2^8+2^5+1=986913$

Observație finală. La reprezentarea numerelor reale în virgulă mobilă, atât în simplă precizie cât şi în precizie dublă, cifra 1 din forma normalizată a numărului real, $\pm 1.f*2^{\pm e}$ nu se reprezintă.

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL III. Elaborarea algoritmilor

3.1 Introducere [18, 53]

In ultimii 30 de ani s-au pus în evidență tehnici și metode generale ce adesea furnizează algoritmi buni pentru rezolvarea unor clase largi de probleme. Printre acestea putem enumera: recursivitatea, metoda backtracking, metoda greedy, metoda divide et impera, programarea dinamică, metoda branch and bound, metode probabilistice, analiza amortizată, algoritmi genetici e.t.c.

Tehnicile de programare sunt caracterizate prin *reflectarea modului conceptual de rezolvare a unei probleme în cadrul structurii algoritmului rezultat* și, în această idee, trebuie să fie *independente în raport cu limbajul de programare folosit.*

De remarcat este faptul că activitatea specifică de implementare a unui algoritm destinat rezolvării unei probleme cu calculatorul fiind, prin însăși natura ei, creativă, *metodele de programare utilizate vor fi cu atât mai eficiente cu cât specificul problemei va fi mai bine explicitat.* Cunoștința și experiența programatorului în domeniul tehnicilor și metodelor de programare este un fapt necesar.

Adesea o problemă poate fi rezolvată prin mai mulți algoritmi ce aplică metode de programare diferite, având timpul de execuție, de implementare, necesarul de memorie, complexitatea, e.t.c. diferite. Alegerea celei mai adecvate metode este, uneori, o sarcină dificilă. Dacă dorim elaborarea celui mai performant algoritm din anumite puncte de vedere, de multe ori, combinarea a două sau mai multe metode de programare ne ajută să atingem acest scop.

Observații

1. Metodele de programare trebuie să ușureze rezolvarea unei probleme, să asigure, în funcție de cerințe, găsirea tuturor soluțiilor sau soluției celei mai bune, eventual în anumite restricții (corelări între date, condiții de timp, resurse hard, e.t.c.) și nicidecum nu sunt obligatorii a se folosi în toate situațiile.
2. Uneori, analizând cerințele unei probleme, constatăm că putem aborda rezolvarea ei prin prelucrări ale datelor anterior aplicării algoritmului propriu zis și, deși programul trebuie completat destul de mult, eficiența lui crește simțitor. Cu alte cuvinte, rezolvarea unei probleme ascunde

https://biblioteca-digitala.ro / https://unibuc.ro

uneori o altă problemă prin care, aplicând unele tehnici şi metode de programare adecvate, reuşim să scurtăm timpul de execuţie al algoritmului per ansamblu.
3. O importanţă deosebită o are cunoaşterea caracteristicilor fiecărei metode, a limitelor ei de aplicare, a complexităţii algoritmilor respectivi.

3.2 Indicii de abordare a unei probleme prin tehnicile de elaborare a algoritmilor [18, 38, 86]

1. Problema acceptă descompunerea în mai multe subprobleme.
a) Dacă subproblemele nu sunt de aceeaşi natură cu problema iniţială şi sunt disjuncte, ele ar putea fi abordate separat folosindu-se, eventual metode corespunzătoare de programare.
b) Dacă subproblemele sunt de aceeaşi natură cu problema iniţială s-ar putea aplica metoda divide et impera sau programarea dinamică.
b1) Dacă problemele nu sunt independente, adică au în comun sub-subprobleme, este potrivită aplicarea programării dinamice, rezolvându-se o singură dată fiecare sub-subproblemă şi memorând rezultatele acestora. Aplicarea metodei divide et impera, ce va rezolva fiecare sub-subproblemă de câte ori este întâlnită, va reduce performanţele algoritmului.
b2) Dacă problemele sunt independente (total disjuncte) s-ar putea aplica fie divide et impera, fie programarea dinamică (în cazul în care adaptarea unui algoritm cunoscut face facilă implementarea). În cazul unei implementări recursive pentru divide et impera se va ţine cont de complexitatea spaţiu (numărul de autoapeluri).

2. Problema cere o soluţie ce optimizează o anumită funcţie criteriu.
a) Dacă se acceptă doar soluţia optimă, se poate încerca o abordare prin metoda greedy sau programarea dinamică sau, în cazul în care căutarea soluţiei necesită generări de soluţii posibile într-un spaţiu de soluţii foarte extins şi natura problemei o impune, se poate folosi metoda backtracking, memorându-se mereu doar soluţia cea mai bună găsită la un moment dat. Dacă se cer toate soluţiile optime se poate combina backtracking cu greedy sau programarea dinamică; pentru unele probleme totuşi programarea dinamică este suficientă, eventual combinată cu backtracking.
b) Dacă se acceptă o aproximare a soluţiei optime, problema se încadrează în euristica greedy.

https://biblioteca-digitala.ro / https://unibuc.ro

3. Algoritmul problemei execută aceleaşi prelucrări pe mai multe nivele.

Acesta este un indiciu pentru a aborda problema prin tehnica recursivităţii sau prin backtracking.

4. Rezultatul cerut este un vector.

Dacă, pentru includerea în soluţie, un element este testat (maxim) o singură dată, ar putea fi potrivită strategia greedy, pe când dacă testarea se poate face de mai multe ori, se poate încerca cu metoda backtracking.

5. Problema admite o substructură optimă (*o soluţie optimă a problemei conţine soluţii optime ale subproblemelor).*

Acesta este un indiciu pentru aplicarea metodei greedy sau a programării dinamice.

6. Problema are proprietatea alegerii greedy (se *poate ajunge la o soluţie optimă global realizând alegeri (greedy) optime local).*

Acesta este un indiciu pentru aplicarea metodei greedy.

7. Prelucrările de bază se efectuează pe structura de date numită stivă.

Acesta este un indiciu pentru a aborda problema prin tehnica recursivităţii sau prin backtracking.

De cele mai multe ori, pentru o problemă suficient de complexă prima idee în rezolvare nu este cea mai bună. Pentru a o optimiza se pot găsi algoritmi ce aplică alte tehnici de programare, cu complexitate polinomială (liniară în cazul fericit) sau se pot combina două sau mai multe metode de programare. Algoritmii, la fel ca şi echipamentele hardware, sunt o tehnologie. Eficienţa globală a unui sistem de calcul depinde la fel de mult de alegerea unor algoritmi performanţi, ca şi de un hardware rapid. Evoluţia metodelor şi tehnicilor de programare este continuă, de maximă actualitate şi dinamica acestui domeniu de dezvoltare tehnologică crează şi va mai crea noi metode de analiză şi proiectare avansată a algoritmilor, generate de necesitatea rezolvării unor cerinţe tot mai complexe.

https://biblioteca-digitala.ro / https://unibuc.ro

3.3 Privire comparativa a tehnicilor de elaborare a Algoritmilor [18, 86, 53]

Acest capitol şi-a propus o abordare cât mai completă a principalelor aspecte legate de cele mai folosite tehnici şi metode de elaborare a algoritmilor. Aici se propune o modalitate de studiu a acestora dar modul de abordare a lor în cadrul procesului de elaborare a algoritmilor rămâne un subiect deschis.

Unul din mecanismele generale de elaborare a programelor este **recursivitatea**. ***Caracteristica principală** a unui subprogram (obiect) recursiv este faptul că el **se autoapelează** (se defineşte cu ajutorul unui obiect de acelaşi tip).* Atunci *când scriem un algoritm recursiv este necesar şi suficient să analizăm ce prelucrări se execută pentru un anumit nivel, deoarece la orice nivel, prelucrările sunt aceleaşi.* Acest tip de raţionament se aplică cu succes algoritmilor a căror operaţii se repetă identic pentru diferite nivele sau algoritmilor ce prelucrează obiecte definite recursiv. Recursivitatea este, în principal, de două tipuri: *directă* şi *indirectă. Pentru corectitudinea subprogramelor recursive sunt esenţiale două aspecte: **îndeplinirea condiţiei de consistenţă*** (pentru orice set valid de date de intrare autoapelurile trebuie să conducă spre situaţia/situaţiile în care subprogramul se execută direct) *şi **corectitudinea condiţiei de oprire***. Trebuie avute, de asemenea, în vedere atât *avantajele* (scrierea concisă şi relativ clară a subprogramelor, estimarea facilă a timpului de execuţie, descris adesea printr-o relaţie de recurenţă) cât şi *dezavantajele* (mărirea memoriei necesare pentru rularea aplicaţiei) acestei tehnici precum şi faptul că nu este recomandată decât în anumite situaţii. Decizia pentru o abordare recursivă sau iterativă este uneori dificilă, cunoscut fiind faptul că pentru orice algoritm recursiv există unul iterativ echivalent şi invers, pentru orice algoritm iterativ există unul recursiv echivalent. Pentru implementarea recursivităţii se foloseşte structura de date ***stivă***. Ca tehnică de programare, *recursivitatea se poate aplica în cadrul unor metode cum ar fi backtracking sau divide et impera.*

Scrierea recursivă a programelor este un caz particular de dezvoltare descendentă şi anume problema de rezolvat este redusă la probleme din ce în ce mai simple, însă identice ca structură cu problema iniţială. Algoritmii de tip **divide et impera**, în implementarea recursivă, implică trei paşi la fiecare nivel de recursivitate:

https://biblioteca-digitala.ro / https://unibuc.ro

Pasul 1: **Divide** problema în $n \in N^*$ subprobleme.
Pasul 2:**Stăpâneşte** cele n subprobleme prin rezolvarea lor în mod recursiv. Dacă dimensiunile acestora sunt suficient de mici soluţia va fi calculată direct.
Pasul 3:**Combină** soluţiile tuturor subproblemelor în soluţia finală pentru problema iniţială.

Bine-nţeles, decizia de a implementa recursiv sau iterativ un algoritm divide et impera este influenţată de mai mulţi factori cum ar fi: numărul şi complexitatea subproblemelor de rezolvat pentru o fază de descompunere, numărul sub-subproblemelor de rezolvat pentru aflarea soluţiei finale – corespunzător numărului de autoapeluri pentru cazul recursiv -, necesarul de memorie pentru rularea algoritmului, memoria ocupată pentru un autoapel în caz recursiv, natura problemei şi corelaţiile între date, e.t.c. Dacă la unii algoritmi, precum algoritmul de căutare binară, între varianta recursivă şi cea iterativă diferenţele nu sunt prea mari din punct de vedere al resurselor alocate şi al timpului de implementare, la alţi algoritmi varianta recursivă este net preferabilă datorită claritătii şi a necesarului de memorie relativ scăzut .

Pentru cazul particular, când la pasul 1 – Divide – n=1, problema, rezolvată recursiv, se încadrează într-un caz clasic de recursivitate directă. De asemenea, în general, pentru un subprogram direct recursiv, se poate formula un algoritm divide et impera (de exemplu problema numărării traseelor posibile parcurse de o piesă de şah de la poziţia (x_i,y_i) la (x_f,y_f), implementată recursiv), eventual cu observaţia că cele n subprobleme în care s-a descompus problema iniţială nu sunt în totalitate disjuncte.

Algoritmii divide et impera sunt în general rapizi deoarece prin descompunere, complexitatea prelucrărilor şi lungimea subproblemelor scade în progresie geometrică. O metodă de estimare a timpului de execuţie al unui algoritm din această clasă se bazează pe teorema master care furnizează o „reţetă" pentru rezolvarea recurenţelor de forma:

$T(n) = a \cdot T(n/b) + f(n)$ unde $a \geq 1$, $b > 1$ constante şi $f(n)$ este o funcţie asimptotic pozitivă.

Un algoritm de referinţă ce foloseşte divide et impera este sortarea rapidă pentru un şir de n elemente, ce are o comportare medie remarcabilă, anume timpul său de execuţie este $\theta(n \cdot \lg n)$, sortează pe loc (în spaţiul alocat şirului de intrare) şi lucrează foarte bine chiar şi într-un mediu de memorie virtuală.

https://biblioteca-digitala.ro / https://unibuc.ro

Metoda greedy, aplicată de obicei problemelor de optim care constau în *determinarea unei submulțimi B a unei mulțimi A* cu n elemente care să îndeplinească anumite condiții pentru a fi acceptată. De obicei *se construiește un algoritm care operează în etape, considerând pe rând câte un element din A, includerea lui în mulțimea B executându-se doar dacă este îndeplinită condiția de* ***optim local. Nu întotdeauna alegerea optimului local conduce la un optim global***. De aceea, în unele probleme strategia greedy conduce la soluții optime (de exemplu algoritmii lui Prim și Kruskal pentru determinarea unui arbore parțial de cost minim într-un graf ponderat) pe când altele, cele la care nu se ajunge la un optim global, formează clasa algoritmilor de tip greedy euristic (de exemplu problema colorării hărților sau problema comis voiajorului, tratate euristic). O importanță deosebită o constituie *analiza corectitudinii algoritmilor greedy* pentru a decide dacă soluția găsită este optimală sau o aproximare a soluției optimale și, în al doilea caz, în ce mod se realizează aproximarea. Algoritmii euristici sunt sensibili la modificarea datelor inițiale. Schimbări mici ale acestora pot avea ca efect apropierea sau îndepărtarea față de soluția optimă.

În general nu există o modalitate de a decide dacă un algoritm greedy rezolvă o problemă particulară de optimizare dar sunt două indicii (condiții necesare, nu și suficiente) pe care le au majoritatea problemelor ce aplică strategia greedy: proprietatea de alegere greedy și substructura optimă. **Proprietatea alegerii greedy** a unei probleme exprimă faptul că se poate ajunge la o soluție optimă global realizând alegeri (greedy) optime local. Aici intervine diferența dintre algoritmii greedy și cei ce aplică programarea dinamică. La rezolvarea unei probleme prin programare dinamică, la fiecare pas al algoritmului, se face câte o alegere dar ea **depinde de soluțiile subproblemelor**. Într-un algoritm greedy se realizează **orice** alegere care pare a fi cea mai bună la momentul respectiv, iar **subproblema rezultată este rezolvată după ce alegerea este făcută. Alegerea realizată de un algoritm greedy poate depinde de alegerile făcute până în momentul respectiv, dar nu poate depinde de alegerile ulterioare sau de soluțiile subproblemelor**. Astfel, spre deosebire de **programarea dinamică** prin care se rezolvă subproblemele în manieră **„bottom-up"**, o strategie **greedy** de obicei programează în manieră **„top-down"**, realizând alegeri greedy succesive și reducând iterativ dimensiunea problemei respective. O problemă evidențiază **o substructură op-**

https://biblioteca-digitala.ro / https://unibuc.ro

timă dacă o soluție optimă a problemei conține soluții optime ale subproblemelor.

Teoria greedoizilor constituie bazele teoretice ale metodei greedy și o parte a sa, teoria matroizilor va fi prezentată în paragraful următor.

Deoarece proprietatea de substructură optimă este exploatată atât de metoda greedy cât și de cea a programarii dinamice, poate exista tendința de a genera o soluție prin metoda programarii dinamice, deși un algoritm greedy ar fi suficient, sau invers, se poate decide în mod eronat că un algoritm greedy poate produce o soluție optimă, deși corectă ar fi aplicarea programării dinamice. Pentru a ilustra diferențele dintre cele două metode vom analiza în continuare două variante ale unei probleme clasice de optimizare: problema rucsacului.

Problema discretă a rucsacului se formulează astfel: Există n obiecte (n - dat) cărora li se cunoaște valoarea și greutatea, reținute în vectorii v, respectiv g, cu elemente de tip intreg. Se cere să se identifice care obiecte trebuie puse într-un rucsac, cu care se poate transporta o greutate maximă GMax dată, astfel încât valoarea încărcăturii să fie maximă. Un obiect nu poate fi luat de mai multe ori și, dacă este pus în rucsac, este pus în întregime.

Problema continuă (fracționară) a rucsacului are aceeași formulare, cu precizarea că un obiect poate fi „tăiat" și pus parțial în rucsac.

La ambele probleme se evidențiază *proprietatea de substructură optimă*. Pentru varianta discretă, considerăm încărcătura cea mai valoroasă cea având greutatea cea mai apropiată de GMax. Dacă din încărcătură se scoate un obiect j, ceea ce rămâne trebuie să fie o încărcătură având greutatea cel mult GMax-g[j] ce se poate lua în rucsac din cele n-1 obiecte (cele n inițiale, mai puțin j). Pentru problema fracționară vom considera că dacă îndepărtăm un obiect j de greutate g din încărcătura optimă, ceea ce rămâne trebuie să fie cea mai valoroasă încărcătură având greutatea cel mult GMax-g ce se poate lua din cele n-1 obiecte inițiale, plus greutatea g[j]-g din obiectul j.

Cu toate că problemele sunt similare, problema continuă a rucsacului poate fi rezolvată cu un algoritm greedy, pe când cea discretă nu admite o rezolvare corectă cu metoda greedy. Pentru a rezolva problema fracționară se calculează mai întâi vectorul x cu n elemente, unde x[i]=v[i]/g[i], se ordonează apoi descrescător și alegerea obiectelor se va face în ordinea corespunzătoare vectorului x sortat până la ocuparea completă a rucsacului, deci greutatea obiectelor alese va fi GMax.

https://biblioteca-digitala.ro / https://unibuc.ro

Ultimul dintre obiectele puse în rucsac ar putea fi parțial. Să remarcăm, de asemenea, că *problema continuă a rucsacului are proprietatea alegerii greedy*.

Pentru a arăta că în varianta discretă nu se poate aplica o strategie greedy vom considera următoarea instanță a problemei:

n=3; g=(10, 20, 30); v=(60, 100, 120); GMax=50.

Se poate calcula x=(6, 5, 4) și se poate cădea în capcana de a alege obiectul 1 pentru care x[1] este maxim. Soluțiile ce conțin obiectul 1 sunt:

S1: obiectul 1, obiectul 2 $\Rightarrow$ valoarea totală = 160

S2: obiectul 1, obiectul 3 $\Rightarrow$ valoarea totală = 180

Soluția optimă este alegerea obiectelor 2 și 3, pentru care valoarea totală este 220.

Considerarea obiectului 1 nu este valabilă în varianta discretă a problemei deoarece nu se poate umple rucsacul până la capacitatea lui maximă iar spațiul liber micșorează câtul valoare/greutate a încărcăturii. În problema discretă, atunci când analizăm posibilitatea punerii unui obiect în rucsac, trebuie să comparăm soluția subproblemei în care obiectul este inclus cu soluția subproblemei în care obiectul lipsește. Comparația trebuie realizată **înainte** de a face alegerea. Problema formulată în acest mod produce o serie de subprobleme care se suprapun – o caracteristică a programării dinamice, care poate fi într-adevăr folosită.

Programarea dinamică, asemenea metodei divide et impera, rezolvă problemele combinând soluțiile unor subprobleme. Spre deosebire de abordarea divide et impera, care partiționează problema în subprobleme independente pe care le rezolvă (recursiv) pentru a combina ulterior soluțiile găsite pentru rezolvarea problemei inițiale, **programarea dinamică este aplicabilă atunci când subproblemele nu sunt independente, adică ele au în comun sub-subprobleme**. În acest context, un algoritm de tipul divide et impera ar presupune mai multe calcule decât ar fi necesar dacă s-ar rezolva în mod repetat sub-subproblemele comune. **Un algoritm bazat pe programare dinamică rezolvă fiecare sub-subproblemă o singură dată și apoi memorează soluția acesteia într-un tablou**, prin aceasta evitând recalcularea soluției ori de câte ori respectiva sub-subproblemă apare din nou. Această idee simplă poate transforma cu ușurința algoritmi exponențiali în **algoritmi polinomiali**. **Metoda divide et impera** operează în stil top-down,

https://biblioteca-digitala.ro / https://unibuc.ro

descompunând probleme în subprobleme pe care le rezolvă separat. La programarea dinamică, programăm în stil bottom-up, pornind de la cele mai mici subprobleme, cărora le combinăm rezultatele pentru a obține soluții pentru probleme din ce în ce mai mari până se ajunge la soluția problemei inițiale.

Revenind la **metoda greedy**, trebuie menționat faptul că, o dată ce un element a fost ales, el rămâne în soluția optimă (curentă), în timp ce dacă un element a fost exclus din soluție, el nu va mai fi reconsiderat pentru includere în soluție. Acesta este principalul motiv pentru care **metoda găsește o singură soluție**, chiar dacă problema admite mai multe soluții. Elementul selectat la un moment dat poate să depindă de cele selectate până la momentul respectiv, dar nu depinde de elementele ce vor fi selectate ulterior. În opinia unor autori metoda greedy poate fi privită ca o particularizare a metodei backtracking în care se renunță la mecanismul de întoarcere. Una din consecințele acestui fapt (inexistența mecanismului de întoarcere) este că metoda greedy conduce la algoritmi **polinomiali** în timp ce metoda backtracking are timp de calcul exponențial. Ca o asemănare, remarcăm că ambele strategii oferă soluția sub formă de vector.

Metoda backtracking se aplică algoritmilor pentru rezolvarea următorului tip de probleme:

Fiind date $n \in N^*$ și mulțimile $A_1, A_2,...,A_n$, finite și ale căror elemente se pot ordona, se cer toți vectorii $x=(x_1, x_2,..., x_n) \in A_1xA_2x...xA_n$ astfel încât să fie îndeplinită o anumită relație $\varphi(x_1, x_2,..., x_n)$. Relația $\varphi(x_1, x_2,..., x_n)$ se numește **condiție internă**, mulțimea finită $S= A_1xA_2x...xA_n$ se numește **spațiul soluțiilor posibile**, iar vectorul x care îndeplinește condiția internă φ se numește **soluție rezultat**. Metoda **construiește soluția pas cu pas**, adăugând elementul x_k doar dacă este construită soluția parțială $x_1, x_2,..., x_{k-1}$. Prin metoda backtracking **se determină toate soluțiile rezultat**, dintre ele putându-se alege una care îndeplinește în plus o anumită condiție (de exemplu minimizează o funcție obiectiv dată), caz în care algoritmul poate fi îmbunătățit (nu se vor genera toate soluțiile pentru ca ulterior să se aleagă una optimă ci se va reține un optim curent cu care se va compara soluția curentă în construcție; generarea soluției se continuă doar în condițiile în care ea ar putea fi mai bună decât optimul curent); **se evită generarea tuturor soluțiilor posi-**

https://biblioteca-digitala.ro / https://unibuc.ro

bile prin verificarea, pentru un x_k ales, ce urmează să fie adăugat în vectorul soluție x, a condițiilor de continuare. Cu minime modificări în algoritmul general al metodei se pot genera un număr limitat de soluții. Metoda este printre puținele care poate fi standardizată. Dezavantajul major al metodei este **timpul de calcul exponențial**.

Domeniul de aplicabilitate al tehnicilor şi metodelor de programare este foarte larg. De exemplu, în teoria grafurilor multe din rezultatele importante se obțin prin mecanisme specifice tehnicilor şi metodelor de programare: pentru aflarea ciclurilor hamiltoniene folosim backtracking, pentru aflarea ciclurilor euleriene se foloseşte greedy, problemele ce prelucrează arbori, folosesc, în general, divide et impera, eventual în varianta recursivă, pentru aflarea drumurilor de lungime minimă într-un graf este folosită programarea dinamică, e.t.c. Algoritmii de sortare sau căutare, utili în rezolvarea a foarte multe probleme, folosesc de asemenea metode de programare (Quicksort – divide et impera; Bubblesort – greedy: găseşte, pe rând, mai mulți vectori, apropiindu-se tot mai tare de soluție; sortarea prin inserție binară – divide et impera; sortarea prin interclasare – poate folosi recursivitatea pe când la interclasarea optimă a n şiruri se aplică greedy).

3.4 Descrierea tehnicilor de elaborare a algoritmilor

3.4.1 Tehnica backtracking [37, 53]

3.4.1.1 Prezentare

Tehnica backtracking se foloseşte în rezolvarea problemelor care îndeplinesc simultan *condițiile*:

c1: Soluția poate fi codificată prin vectori cu n elemente, $S=(x_1,x_2,\ldots,x_n)$ cu $x_i \in A_i$, pentru i= 1,...,n;

c2: mulțimile A_1, $A_2,\ldots,A_n$ sunt mulțimi finite, iar elementele lor pot fi ordonate după o relație de ordine. Pentru fiecare A_i trebuie să se poată preciza primul element, succesiunea elementelor şi ultimul element.

c3: nu se dispune de o altă metodă de rezolvare, mai rapidă.

Observații:

1. Nu pentru toate problemele se cunoaste n de la început.
2. $x_1,x_2,\ldots,x_n$, pot fi la rândul lor date structurate (de exemplu vectori sau articole).

https://biblioteca-digitala.ro / https://unibuc.ro

3. În multe probleme $A_1=A_2=\ldots=A_n$.
4. *Metoda backtracking se bazează pe structura de stivă deoarece construirea soluției S se face prin adăugări și/sau eliminări de elemente din vectorul S conform disciplinei de stivă.*)LIFO = last-in, firs-out).

Principiul care stă la baza algoritmului backtracking este următorul:

- se construiește soluția pas cu pas, generând, pe rând, elementele vec-torului S, în ordine x_1, apoi $x_2,\ldots$,apoi x_{i+1} (următorul după x_i)... și în final x_n
- dacă se constată că, pentru o valoare aleasă corespunzătoare poziției k, nu se poate ajunge la soluție, se renunță la acea valoare și se continuă căutarea din punctul în care am rămas (se evită generarea a cât mai multor elemente din produsul cartezian $A_1xA_2x\ldots xA_n$).

Mulțimea $A_1xA_2x\ldots xA_n$ se numește **spațiul soluțiilor**. În fiecare problemă particulară se precizează anumite relații între elementele $(x_1,x_2,\ldots,x_n)$ și condiții pentru x_i , i= $\overline{1,n}$, acestea formând **condițiile interne**. Soluțiile ce satisfac condițiile interne se numesc **soluții rezultat**.

Concret se procedeaza astfel:
1. I se atribuie lui x_1 valoarea primului element din A_1.
2. Presupunând generate elementele $x_1,x_2,\ldots,x_k$ aparținând mulțimilor A_1, $A_2,\ldots,A_k$, se încearcă continuarea construirii soluției pe poziția k+1, putând să apară situațiile:
- În A_{k+1} nu există nici un element disponibil (netestat) care alături de $x_1,x_2,\ldots,x_k$ ar putea să conducă la soluția problemei. Se consideră situație de impas, când considerăm generate $x_1,x_2,\ldots,x_{k-1}$ și se reia 2 pentru $k\rightarrow k-1$.
- În A_{k+1} putem identifica x'_{k+1} cu proprietatea că $x'_{k+1}>x_{k+1}$ și nu există $x''_{k+1}\in A_{k+1}$ a.î. $x_{k+1}<x''_{k+1}<x'_{k+1}$, $x''_{k+1}\neq x'_{k+1}$. (x_{k+1} este cel mai mare element din A_{k+1} care a fost testat. Dacă în mulțimea A_{k+1} nu am testat nici un element, x'_{k+1} va fi primul element din A_{k+1} .)

În acest caz se execută $x_{k+1}\leftarrow x'_{k+1}$ și apoi se testează dacă x_{k+1} îndeplinește anumite condiții de continuare (ce sunt condiții necesare pentru a ajunge la o soluție-rezultat) apărând două posibilități:
=. Condițiile de continuare sunt îndeplinite, caz în care se testează dacă s-a ajuns la soluție.
Dacă s-a ajuns la soluție, se tipărește soluția și se reia algoritmul cu 2.
Dacă nu s-a ajuns la soluție, atunci $k\leftarrow k+1$ și reia algoritmul cu 2.

https://biblioteca-digitala.ro / https://unibuc.ro

=. Condițiile de continuare nu sunt îndeplinite, caz în care se reia algoritmul de la 2.

Algoritmul se termină atunci când nu mai există nici un element $x_1 \in A_1$ netestat.

Metoda backtracking are ca rezultat obținerea tuturor soluțiilor problemei. Dacă se cere o singură soluție, prin program se poate forța oprirea execuției, atunci când aceasta a fost găsită.

Considerăm că generarea soluțiilor se face într-o stivă, st și $x_i \in A_i$ se va găsi pe nivelul i al stivei, pentru i= 1,...,n.

Observații.

1. A continua construirea unei soluții pe poziția k înseamnă a înlocui $x_k \in A_k$ cu valoarea următoare a lui x_k din A_k (mulțimea A_k fiind ordonată). Când această continuare nu este posibilă (x_k fiind ultimul din A_k) apare situația de impas pe nivelul k în stivă și, în program, corespunzător se va executa k←k-1 („coborâre" în stivă), în încercarea de a rezolva impasul.
2. A începe construirea soluției S pe poziția k înseamnă a atribui lui x_k o valoare x_{k0} aflată logic înaintea primei valori din A_k. Dacă „<" este relația de ordine pe mulțimea A_k și x_{k1} este cel mai mic element din A_k, atunci x_{k1} trebuie să fie succesorul lui x_{k0} și $x_{k0} < x_{k1}$.
3. Condițiile de continuare sunt specifice fiecărei probleme și ele exprimă posibilitatea ca $x_1, x_2, \ldots, x_k$ să facă parte dintr-o soluție-rezultat a problemei.

Se recomandă ca, în rezolvarea unei probleme cu metoda backtracking, să se parcurgă următoarele **etape**:

E1. Identificarea mulțimilor $A_i = \{x_{i1}, x_{i2}, \ldots x_{in}\}$, pentru i=1,n.

Am considerat că:

- x_{i1} este " cea mai mică" valoare din A_i ce poate apărea pe poziția i în vectorul soluție st=$(x_1, x_2, \ldots, x_n)$
- $x_{i,j} <_i x_{i,j+1}$ pentru $j=\overline{1,n_i-1}$ conform relației „$<_i$" stabilită pe mulțimea A_i.
- n_i = numărul de elemente (distincte) ale mulțimii A_i

E2. Identificarea numerelor x_{k0} pentru k =1,n ce semnifică valorile ce se află logic înaintea lui x_{k1}, $x_{k0} <_k x_{k1}$.

Procedure INIT (k:integer; st:stack);

inițializează nivelul k al stivei st cu x_{k0}.

https://biblioteca-digitala.ro / https://unibuc.ro

E3. Implementarea procedurii cu antetul
Procedure SUCCESOR (as:boolean;st:stack;k:integer);
st[k] se identifică cu x_{kj}, unde $j \in \{0,1,\ldots,n_k\}$, pentru k=1,...,n. Dacă $j \neq n_k$ atunci st[k] are succesor, pe $x_{k,j+1}$ şi altfel st[k] nu are succesor.
Această procedură verifică dacă $st[k]=x_{k,n_k}$ (element ce a fost identificat în etapa E1) şi dacă aceasta se verifică atunci se execută as:=FALSE (adică st[k] nu are succesor) şi altfel se execută as:=TRUE şi $st[k]:=x_{k,j+1}$.
E4. Implementarea procedurii cu antetul
Procedure VALID (ev:boolean; st:stack; k:integer);
Când se execută această procedură, se ştie că în stivă, până la nivelul k-1 sunt îndeplinite condiţiile de continuare. Procedura calculează ev ce va avea valoarea true dacă şi numai dacă st[1], st[2],...,st[k-1], st[k] pot genera o soluţie a problemei (adică îndeplinesc condiţiile de continuare, specifice problemei). Verificarea se face asupra lui st[k].
E5. Implementarea funcţiei cu antetul
Function SOLUTIE (k:integer):boolean;
care întoarce valoarea true dacă şi numai dacă st[1], st[2],...,st[k] este soluţie a problemei. Vectorul st este variabilă globală pentru SOLUTIE.
E6. Implementarea procedurii cu antetul
Procedure TIPAR;
Este procedura de tipărire a unei soluţii, mai exact a valorilor st[1], st[2],...st[k-1], st[k].
E7. Scrierea programului folosind (eventual) urmatoarea rutina generală backtracking:

```
k:=1; INIT(1,st);
while k>0
      do
                  SUCCESOR (as,st,k);
                  if as then VALID (ev,st,k)
                        endif
      until (not as) or (as and ev);
            if as then
                        if  SOLUTIE (k) then TIPAR
                                          else k←k+1;
                                             INIT (k,st)
                        endif
                  else k←k-1
```

https://biblioteca-digitala.ro / https://unibuc.ro

endif
repeat;

E8. Rularea programului şi eliminarea eventualelor erori.

Observaţii asupra algoritmului backtracking iterativ:
1. Ordinul de complexitate al algoritmului BKT este $n_1 \cdot n_2 \cdot \ldots \cdot n_n$ unde n_i este cardinalul mulţimii A_i; dacă $A_1=A_2=\ldots=A_n$ şi $cardA_i = c$ atunci algoritmul BKT are complexitate c^n. Prin urmare, problemele rezolvate cu această metodă necesită un timp de rulare îndelungat (au *complexitate exponenţială*). Acesta este principalul dezavantaj al metodei şi deci ea se aplică doar în situaţiile în care nu se dispune de un alt algoritm mai eficient.
2. Metoda asigură găsirea *tuturor soluţiilor* problemei. Dacă se cere doar un anumit număr de soluţii algoritmul general poate fi modificat corespunzător.
3. Rezolvarea iterativă propusă încalcă principiul stivei la implementarea procedurilor VALID, TIPAR şi, în unele probleme funcţia SOLUTIE dar putem considera că st simulează funcţionarea unei stive deoarece operaţiile de adăugare şi extragere a elementelor se execută la un singur capăt. În program k va reprezenta mereu indicatorul de stivă.
4. Dacă soluţia problemei cerute are o lungime dată, n, ce nu se modifică, atunci funcţia SOLUTIE poate fi implementată fără parametri.
Un exemplu de problemă la care soluţia are lungime variabilă este:

« Se dă n = 4..20 şi k = 1..20, $k \leq n$. Să se tipărească toate modalităţile de a aranja pe o tablă de şah, de dimensiune n·n, minim k regine, care să nu se atace reciproc. O aranjare posibilă se va tipări sub forma:
$[l_1,c_1]$, $[l_2,c_2]$, ...,$[l_x,c_x]$ cu semnificaţia:
„s-au aranlat pe tablă x regine, R_i, i = 1,...,x; regina a i-a este pe linia l_i şi coloana c_i" ».
5. *Metoda poate fi aplicată şi problemelor care cer găsirea unei soluţii* $(x_{i1},x_{i2},\ldots,x_{in})$ *care optimizează o funcţie criteriu* $P(x_1,x_2,\ldots,x_n)$ cu $x_i \in A_i$, i= 1,n unde A_i este o mulţime finită în care elementele sale pot fi puse într-o ordine clară. Practic, se generează, în cel mai defavorabil caz, toate elementele produsului cartezian $A_1xA_2x\ldots xA_n$ şi, pentru fiecare element, se verifică dacă valoarea funcţiei criteriu P obţinută este „ mai bună" decât P_0, P_0 reprezentând valoarea lui P cea mai bună obţinută până

https://biblioteca-digitala.ro / https://unibuc.ro

atunci. Calculul valorii lui P nu trebuie făcut complet ci doar atât timp cât avem şansa să găsim o valoare mai bună decât P_0.
6. Observarea atentă a particularităţilor problemei, implementarea în subprogramele VALID, SOLUTIE, INIT, SUCCESOR a unor testări cât mai eficace, a iniţializărilor sau calculul lui as eficiente, pot conduce la micşorarea drastică a numărului de încercări în căutarea soluţiei şi deci la micşorarea foarte mult a timpului de calcul. De aceea, un *backtracking optimizat* poate avea, şi în cazul unui n relativ mare, pentru unele probleme, o utilitate practică.

3.4.1.2 Backtracking generalizat [37, 18, 53]

În cele mai multe cazuri, algoritmul backtracking, foloseşte o stivă cu elemente simple (nu cu tip structurat) deoarece mulţimile A_i sunt mulţimi de numere. În cazul în care elementele din A_i sunt vectori (cu două, trei,...x elemente) stiva în care se construieşte soluţia este o stivă dublă, triplă,...e.t.c. şi problema se încadrează în capitolul de backtracking generalizat. Un exemplu ar fi **Problema canibalilor şi misionarilor**:
Se dau c, m:integer ; $c,m \in N^*$.
Pe malul unei ape se găsesc c canibali şi m misionari. Ei doresc să treacă apa folosind o barcă cu două locuri. În situaţia în care, pe oricare mal, există la un moment dat mai mulţi canibali decât misionari, misionarii sunt mâncaţi.
Să se tipărească toate variantele de trecere a apei astfel încât nici un misionar să nu fie mâncat.

Analiza problemei: Există 5 modalităţi de a traversa râul cu barca:

1: trec doi misionari;
2: trec doi canibali;
3: trec un misionar şi un canibal;
4: trece un misionar;
5: trece un canibal;

Conform strategiei backtracking se va folosi o stivă care însă va avea drept elemente vectori cu câte 5 valori. Dacă s[k] este elementul de pe poziţia k din stiva soluţie s atunci:
s[k][1] reţine codul (număr natural între 1 şi 5) celei de a k-a trecere cu barca;
s[k][2] reţine numărul de misionari de pe malul 1 la începutul traversării k;

s[k][3] reţine numărul de canibali de pe malul 1 la începutul traversării k;

https://biblioteca-digitala.ro / https://unibuc.ro

s[k][4] reține numărul de misionari de pe malul 2 la începutul traversării k;
s[k][5] reține numărul de canibali de pe malul 2 la începutul traversării k;

Convenim că pentru k impar barca traversează râul de la malul 1 la malul 2 iar pentru k par barca trece de la malul 2 la malul 1. O variantă de trecere a apei va consta în tipărirea, una sub alta, a mai multor configurații reținute în s[1], s[2],...; s[1] este configurația inițială; s[1] = (cod_traversare, m, c, 0, 0); configurația finală va fi de forma (cod_traversare, 0, 0, m, c).
Configurația s[k+1] se calculează pe baza lui s[k] astfel:
- s[k+1][1] la început se inițializează cu zero (în procedura INIT) și eventual se modifică la execuția procedurii SUCCESOR)
- s[k+1][i], pentru i=2,5 se inițializează prin procedura INIT cu valori calculate anterior, verificate cu procedura VALID ca fiind bune, și transmise ca variabile globale în program (nc1, nm1, nc2, nm2).
Semnificația celor 4 variabile este:
nm1 = numărul curent al misionarilor de pe malul 1
nc1 = numărul curent al canibalilor de pe malul 1
nm2 = numărul curent al misionarilor de pe malul 2
nc2 = numărul curent al canibalilor de pe malul 2
Procedura VALID verifică dacă a k-a configurație poate face parte dintr-un șir de traversări în condițiile problemei; de asemenea, VALID modifică valoarea variabilelor nm1, nc1, nm2, nc2 în funcție de k și s[k][1].
Ca rezultat, se calculează ev:boolean cu semnificația

$$ev = \begin{cases} true, & \text{traversarea a k-a este posibilă} \\ false, & \text{traversarea a k-a nu este posibilă} \end{cases}$$

ev = false $\Leftrightarrow$ C1 OR C2 OR C3 OR C4
C1: Pe malul de unde se pleacă nu există numărul minim de misionari / canibali necesar efectuării trecerii
C2: Se prevede posibilitatea ca misionarii să fie mâncați
C3: S-a ajuns la o configurație întâlnită anterior
C4: Se încearcă să se efectueze traversarea anterioară (s[k][1]=s[k-1][1]) (lucru lipsit de sens în contextul problemei).

https://biblioteca-digitala.ro / https://unibuc.ro

O problemă asemănătoare ar fi tipărirea unei variante în care numărul de traversări cu barca ar fi minim. În acest caz programul anterior se poate adapta astfel:

- se declară nmin:longint; cu semnificația: număr minim de traversări, corespunzând celei mai bune variante întâlnite până la un moment dat; inițial, nmin:=1000000000;
- se declară sopt:stack în care se va reține soluția optimă găsită la un moment dat şi va fi o variabilă globală
- corpul procedurii VALID devine:

if k>nmin then ev←false
else ...{conținutul procedurii VALID anterioare}
endif

1 in programul principal vom avea si instructiunea:

„If SOLUTIE (k) then nmin←k;
so←s
endif"

- procedura TIPAR va avea k drept parametru de intrare
- înaintea end-ului final (din program) se va pune instrucțiunea:

„if nmin<>1000000000 then TIPAR (nmin)
else writeln ('Problema nu are soluție')
endif

Funcția criteriu P nu apare explicit în program; ea returnează numărul minim de traversări cu barca.

3.4.1.3 Backtracking recursiv [18, 53]

O caracteristică a strategiei backtracking este faptul că soluția se construieşte progresiv în stiva st şi pe fiecare nivel din stivă se execută aceleaşi prelucrări (bineânțeles asupra altor valori – în general). Pornind de la această observație este clar că putem implementa strategia backtracking folosind recursivitatea, mai exact, vom scrie în continuare variante generale de proceduri de tip backtracking recursiv care au parametru de intrare k:integer, cu semnificația : k=nivelul din stiva soluție st, pe care se fac prelucrările.

Pe fiecare nivel k sunt efectuate următoarele operații asupra componentei a k-a, st[k]:

https://biblioteca-digitala.ro / https://unibuc.ro

1. Inițializarea lui st[k] cu o valoare aflată logic înaintea primei valori din A_k.
2. Cât timp st[k] are succesor în A_k execută
   2.1. st[k] ← succesorul lui st[k] în mulțimea A_k.
   2.2. Dacă st[k] este valid atunci
        Dacă st[1], st[2], ...,st[k] este soluție
            atunci tipărește st[1], st[2], ...,st[k]
            altfel apelează procedura pentru k+1
        Sf_dacă;
   Sf_dacă;
Sf_ cât_ timp.

Observații.
1. Notațiile folosite au semnificațiile anterior prezentate.
2. În procedura BKTR scrisă în continuare, vom folosi următoarele subprograme:
* Procedure INIT (k:integer; st:stack);
care inițializează st[k] cu o valoare aflată înaintea primei valori din A_k.
* Function SUCCESOR (k:integer; st: stack):boolean;
care returnează valoarea false dacă st[k] nu are succesor în A_k (echivalent cu faptul că st[k]=$X_{k,nk}$) și returnează valoarea true dacă st[k] are succesor în A_k – adică ($\exists$) i<n_k și st[k]=$X_{k,i}$– caz în care st[k] se modifică astfel: st[k]←$x_{k,i+1}$.
* Function VALID (k:integer; st: stack):boolean;
care returnează valoarea true dacă st[1], st[2], ... ,st[k] ar putea conduce la soluție (sunt îndeplinite condițiile de continuare sau, altfel spus, st[k] este un element valid); funcția returnează valoarea false dacă st[k] nu este valid, adică st[1], st[2], ...,st[k] nu poate fi parte a unei soluții.
* Function SOLUTIE (k:integer):boolean;
care returnează valoarea true dacă st[1], st[2], ...,st[k] este soluție, respectiv false dacă st[1], st[2], ...,st[k] nu este soluție
* Procedure TIPAR (k:integer);
care tipărește soluția st[1], st[2], ...,st[k] în forma dorită.

Algoritmul general backtracking recursiv este:
Procedure BKRT (k:integer);
INIT (k,st);
while SUCCESOR (k,st)

https://biblioteca-digitala.ro / https://unibuc.ro

```
    if VALID (k,st) then
                    if SOLUTIE (k) then TIPAR (k)
                                        {*}
                                else BKTR (k+1)
                    endif
     endif
end;
```

Observații asupra procedurii BKTR:

1. Subprogramele INIT, SUCCESOR şi VALID pot rămâne doar cu k drept parametru de intrare şi st va fi variabilă globală în program (S-a preferat scrierea anterioară pentru a se putea observa mai uşor legătura între algoritmul backtracking iterativ, general şi procedura BKTR). Dacă lungimea soluției este o valoare cunoscută, subprogramele SOLUTIE şi TIPAR pot să nu aibă nici un parametru.
2. În programul principal se va apela BKTR (1) pentru că se va începe cu generarea lui st[1].
3. Pentru generarea unei soluții st, pas cu pas, trecerea la nivelul următor în stivă se face prin autoapelul procedurii în linia marcată cu {*}, mai exact în cazul în care st[1], st[2],... ,st[k] îndeplineşte condițiile de continuare dar nu este soluție finală.

La nivelul k în stivă se testează, pe rând, toate valorile din A_k, acest lucru fiind asigurat de instrucțiunea „while SUCCESOR (k,st) ..." . În momentul în care funcția SUCCESOR(k,st) returnează valoarea false, echivalent cu terminarea testării tuturor valorilor din A_k, execuția procedurii BKTR (k) se termină şi, dacă k=1 înseamnă că algoritmul de generare al soluțiilor s-a încheiat, iar dacă k>1 înseamnă că BKTR (k) s-a încheiat, adică trebuie să se revină în procedura apelantă, caz în care k←k-1 şi se continuă execuția procedurii BKTR, pe acest nivel k, de unde a rămas.

4. O altă variantă de procedură backtracking recursiv, la care remarcăm condiția de test (SOLUTIE (k)) pentru apelul recursiv, scrisă la început, este:

```
Procedure BKTR (k:integer);
    if SOLUTIE (k) then tipar (k)
            else INIT (k,st);
                    while SUCCESOR (k,st)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                                    if VALID (k,st) then BKTR (k+1)
                                    endif
                                repeat
        endif
  end;
```

5. Considerând $A_k = \{x_1, x_2,\ldots,x_{n[k]}\}$, alte variante ale procedurii backtracking recursiv sunt:

```
Procedure BKTR (k:integer);
integer i;
for i:=1 to n[k]                    {n[k]=numărul de elemente din Ak}
    st[k]←x[i];
    if VALID (k,st) then
                        if SOLUTIE (k) then TIPAR (k)
                                       else BKTR (k+1)
                        endif
    endif
repeat
end;
```

```
Procedure BKTR (k: integer);
integer i;
if SOLUTIE (k-1) then TIPAR (k-1)
   else for i:=1 to n[k]
              st[k]←x[i];
                    if VALID (k) then BKTR (k+1)
                    endif
        repeat;
endif
end
```

Considerăm SOLUTIE (0)= false.

Variabila i, care va ajuta la parcurgerea tuturor elementelor din A_k, este important să fie locală procedurii BKTR.

6. Remarcăm faptul că procedura BKTR, în orice variantă a sa, are avantajul de a fi uşor de scris, scurtă şi clară dar în acelaşi timp implementarea ei trebuie să ţină cont că, datorită autoapelurilor repetate ea nu poate fi aplicată decât pentru lungimi relativ mici ale stivei soluţie st.

https://biblioteca-digitala.ro / https://unibuc.ro

3.4.2 Recursivitatea [18, 37, 53]

3.4.2.1 Prezentare

Recursivitatea este un mecanism general de elaborare a programelor, utilizat mai frecvent după anii '80. Deşi este una dintre noţiunile fundamentale ale informaticii, unele limbaje de programare, chiar evoluate (Fortran, Cobol) nu permit scrierea programelor recursive.

Recursivitatea a apărut din necesităţi practice cum ar fi transcrierea directă a formulelor matematice recursive sau prelucrarea unor structuri de date definite recursiv.

Caracteristica principală a unui subprogram (obiect) recursiv este faptul că el *se autoapelează* (se defineşte cu ajutorul unui obiect de acelaşi tip).

Un algoritm recursiv are la bază un *mecanism de gândire specific*.

Atunci când scriem un algoritm recursiv este suficient să analizăm ce prelucrări apar pentru un anumit nivel, deoarece la orice nivel prelucrările sunt aceleaşi. Acest tip de raţionament se aplică cu succes algoritmilor a căror operaţii (prelucrări) se repetă identic pentru nivele diferite sau algoritmilor ce prelucrează obiecte definite (de cele mai multe ori riguros matematic) recursiv.

Scrierea recursivă a programelor este un caz particular de dezvoltare descendentă, şi anume problema de rezolvat este redusă la probleme din ce în ce mai simple, însă *identice ca structură*, cu problema iniţială.

Vom exemplifica în continuare scrierea unui subprogram recursiv prin scrierea unei funcţii numită funcţia lui Ackermann Ack : N x N \{(0,0)} → N definită prin

$$Ack(m,n) = \begin{cases} n+1 & , \text{ dacă } m=0 \text{ şi } n\neq 0 \\ Ack(m-1,1) & , \text{ dacă } n=0 \text{ şi } m\neq 0 \\ Ack(m-1, Ack(m,n-1)) & , \text{ dacă } m\cdot n\neq 0 \end{cases}$$

Subprogramul recursiv este urmatorul:

```
Function ACK(m,n:integer):integer;
if m=0 then Ack←n+1
  else if n=0 then Ack←Ack(m-1,1)
         else Ack←Ack(m-1,Ack(m,n-1))
       endif
end;
```

https://biblioteca-digitala.ro / https://unibuc.ro

Observații

1. De la început este verificată *condiția de oprire a autoapelului* (m=0) și *execuția funcției conduce spre „apropierea" de această condiție* și în final se returnează rezultatul dorit.
2. Autoapelul funcției apare în 3 locuri în subprogram dar execuția va păstra , în esență, toate caracteristicile unui apel de funcție.
3. În mod analog, prin transcrierea relațiilor recursive, se pot descrie n!=n·(n-1)!, Fib (n)=termenul al n-lea din șirul lui Fibonacci

$$Fib(n) = \begin{cases} n & \text{, dacă } n \le 1 \\ Fib(n-1)+Fib(n-2) & \text{, dacă } n > 1 \end{cases}$$

$$Cmmdc(a,b) = \begin{cases} cmmdc(a-b,b) & \text{, dacă } a > b \\ cmmdc(a,b-a) & \text{, dacă } a < b \end{cases}$$

sau

$$Cmmdc(a,b) = \begin{cases} a & \text{, dacă } a = b \\ cmmdc(a, a \text{ MOD } b) & \text{, dacă } a \neq b \end{cases}$$

(pentru $a,b \in N^*$) și altele.

3.4.2.2 Tipuri de recursivitate [18]

Un subprogram este recursiv dacă el se autoapelează. Când apelul apare în instrucțiunea compusă a subprogramului (notat P1), recursivitatea se numește **directă**.

```
Procedure P1 (lista parametrilor formali);
...
call P1 (listă de parametrii actuali); ...
end
```

Dacă apelul subprogramului P1 apare în instrucțiunea compusă a unui alt subprogram (notat P2), care se apelează din subprogramul P1, recursivitatea se numește **indirectă**, iar subprogramele P1 și P2 se numesc *mutual recursive*. În pseudocod, putem scrie schematic următoarea structură în cadrul declarațiilor:

https://biblioteca-digitala.ro / https://unibuc.ro

```
Procedure P2 (lista parametrilor formali); FORWARD;
Procedure P1 (lista parametrilor formali);
...
call P2 (lista de parametrii actuali )...
end;
...
Procedure P2; {nu se mai scrie lista parametrilor formali}
...
call P1 (lista de parametrii actuali )...
end
```

Observații:

1. Clauza FORWARD anunță compilatorul că subprogramul respectiv va fi definit ulterior.
2. Cazul cel mai general de recursivitate indirectă este când P1 apelează (în corpul lui P_1) pe P_2 și P_2 apelează (în corpul lui P_2) pe P_3 care apelează pe P_4 și așa mai departe... P_{n-1} apelează pe P_n iar în corpul lui P_n se apelează P_1.
3. Sunt situații când cele două tipuri de recursivitate se combină în același program. În acest caz vorbim despre recursivitate mixtă.

Exemplele anterioare corespund recursivității directe.

Un exemplu pentru recursivitatea indirectă este calculul valorilor în $x_o \in R$ pentru două funcții $f,g:R \to R$, definite una prin intermediul celeilalte.

$$f(x)=\begin{cases} 2\cdot g(x)-1 & \text{, dacă } x \le 5 \\ x^2-x+1 & \text{, dacă } x>5 \end{cases}$$

$$g(x)=\begin{cases} 1 & \text{, dacă } x < 0 \\ f(x-3)+7\cdot x & \text{, dacă } x \ge 0 \end{cases}$$

Pseudcodul pentru acest calcul este:

```
Procedure recind
 real x0,x;
 Function g(x:integer):integer; FORWARD;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
Function f(x:integer):integer;
 if x>5 then f←x*x-x+1
        else f←2*g(x)-1
 endif
Function g;
 if x < 0 then g←1
          else g←f(x-3)+7*x
 write ('xo= '); Read(x0);
 write ('f (',xo,')=',f(xo));
 write ('g (',xo,')=',g(xo));
end.
```

Un exemplu pentru recursivitatea mixtă îl reprezintă şirul mediilor lui Gauss. Considerându-se a_0, $b_0 \in R_+$ şi $n \in N^*$, se definesc şirurile:

$$a_n = \frac{a_{n-1} + b_{n-1}}{2} \quad ; \quad b_n = \sqrt{a_{n-1} * b_{n-1}}$$

Se pot scrie două funcţii, a şi b, cu parametrul de intrare n. Fiecare dintre ele se autoapelează dar o apelează şi pe cealaltă.

3.4.2.3 Execuţia programelor recursive [53]

La activarea unui program, se salvează în stiva procesorului (stack) :
a. contextul subprogramului (adică sunt locaţii de memorie unde se reţin:

- valorile parametrilor transmişi prin valoare
- adresele parametrilor de tip variabilă (transmişi prin adresă)
- valorile variabilelor locale în subprogram) şi

b. adresa de revenire din subprogram (ce indică instrucţiunea din programul / subprogramul apelant ce urmează a fi executată la sfârşitul execuţiei subprogramului apelat).

Pentru subprogramele recursive mecanismul este acelaşi. La fiecare apel al subprogramului, se va aloca în stivă un nou nivel unde se va salva setul de valori (parametri, variabile locale, adresa de revenire) corespunzătoare stării curente a execuţiei subprogramului. Se remarcă faptul că, deşi parametrii şi variabilele locale au, pentru toate apelurile, aceiaşi identificatori, valorile lor diferă de la un nivel de execuţie la altul. În referirile la identificatori, valorile acestora corespund întotdeauna ultimului set de valori salvate. La terminarea execuţiei unui apel recursiv,

https://biblioteca-digitala.ro / https://unibuc.ro

se reia ultima execuție întreruptă, în stivă fiind activ setul de valori corespunzătoare acesteia.

Vom prezenta în continuare o situație relativ simplă a unui apel de subprogram pe care, analizând-o, vom evidenția caracteristici generale ale apelurilor de subprograme:

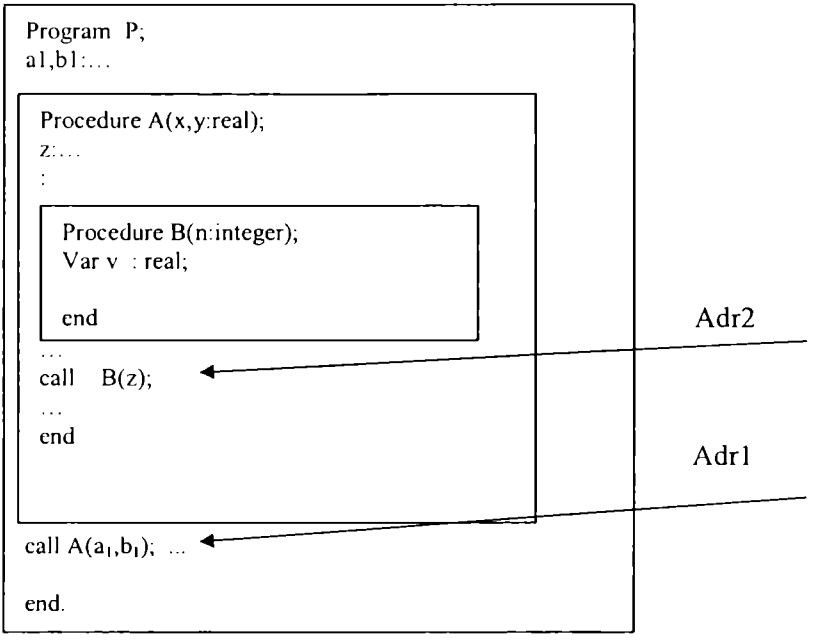
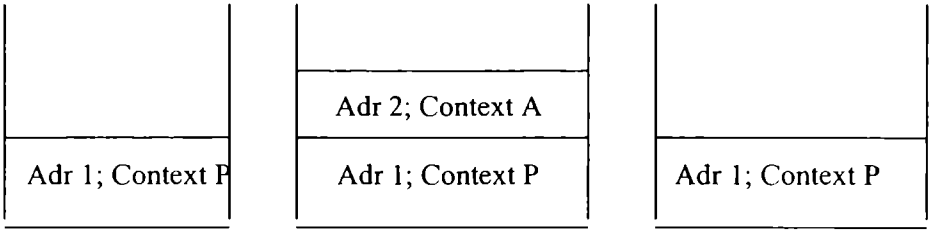


Fig. 1.16

Observație: Din P nu se poate apela B.

Evoluția stivei gestionată de compilator:

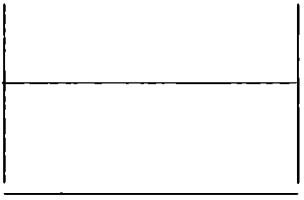


a) La apelarea procedurii A

b) La apelarea lui B din A

c) La terminarea execuției procedurii

https://biblioteca-digitala.ro / https://unibuc.ro



d) Stiva golită la încheierea procedurii A

Fig. 1.17

Fluxul de control (ordinea execuției instrucțiunilor) este:

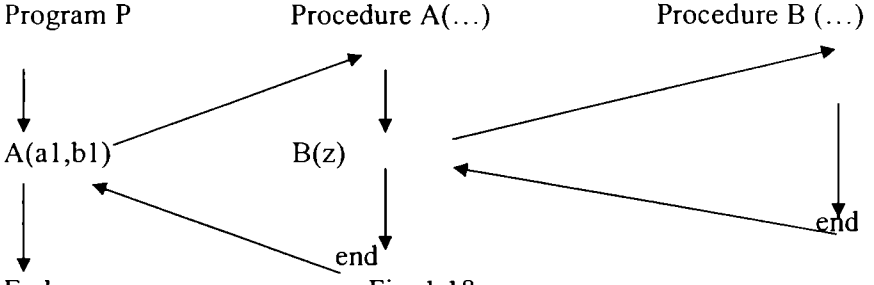


Fig. 1.18

Observații asupra recursivității

1. În orice subprogram recursiv trebuie să existe măcar o situație în care execuția subprogramului nu are autoapelul adică în care operațiile ce vor fi executate conduc direct la rezultat. Această situație este condiționată de o expresie logică, numită *condiție de oprire* (a autoapelului) ce se recomandă a fi verificată cât mai la începutul corpului subprogramului recursiv şi se recomandă să fie scrisă explicit, chiar dacă în acel caz execuția programului constă în instrucțiunea vidă.

2. Un subprogram recursiv corect trebuie să îndeplinească *condiția de consistență* adică autoapelurile trebuie să conducă spre situația (situațiile) în care subprogramul se execută direct (fără autoapel). Acest principiu este o consecință a faptului că algoritmii se caracterizează prin finitudine. Un exemplu în care condiția de consistență nu este îndeplinită este următoarea funcție prin care se dorește calcularea termenului de rang n, n≥0, din şirul lui Fibonacci.

```
Function Fibo(n:integer):integer;
 if n=0 then Fibo←1;
 endif
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
 if n=1 then Fibo←1
    else Fibo←Fibo(n-1)+Fibo(n-2)
 endif
end
```

Aparent corectă, funcția nu va returna rezultatul bun decât pentru n=1, în rest ajungându-se mereu să se apeleze Fibo(0), caz în care Fibo(0)←Fibo(-1)+Fibo(-2), Fibo(-k) unde k∈N* apelând Fibo (-k-1) și Filo (-k-2) și rezultatul este că se „umple" stiva de memorie fără a se returna valoarea funcției (teoretic, subprogramul rulează la nesfârșit).

3. Condiția de consistență trebuie să fie îndeplinită pentru orice set valid de date de intrare.

4. Puterea de exprimare a recursivității constă în posibilitatea definirii unei mulțimi infinite de obiecte sau prelucrări cu ajutorul unui număr finit de afirmații.

5. Definițiile recursive ale funcțiilor se pot clasifica în:

* *Recursie liniară,* cu forma generală:

$$f(x)=\begin{cases} g(x) & , c(x) \text{ adevărată} \\ \rho(f(h(x)),h(x)) & , c(x) \text{ falsă} \end{cases}$$

Exemplu:
- funcția factorial

* *Recursie neliniară, tip cascadă* cu forma generală:

$$f(x)=\begin{cases} g(x) & , c(x) \text{ adevărată} \\ \rho(f(h(x)),f(i(x))) & , c(x) \text{ falsă} \end{cases}$$

Exemplu:
- funcția Fibonacci

* *Recursie liniară de tip împachetat,* cu forma generală:

$$f(x)=\begin{cases} g(x) & , c(x) \text{ adevărată} \\ f(...f(...f(...)...)...) & , c(x) \text{ falsă} \end{cases}$$

Forma cea mai generală a acestor recursii se bazează pe enunțul case și este următoarea:

$$f(x)=\begin{cases} g(x) & , c(x) \text{ adevărată} \\ \left.\begin{array}{l} \text{case} \\ \quad c_1(x) : f_1 \\ \quad c_2(x) : f_2 \\ \quad \ldots \\ \quad c_n(x) : f_n \\ \text{endcase} \end{array}\right\} & , \quad c(x) \text{ falsă.} \end{cases}$$

https://biblioteca-digitala.ro / https://unibuc.ro

In aceste expresii c(x) este o condiție (predicat) ce cuprinde argumentele funcției f.

În această categorie se încadrează funcțiile Manna-Pnueli și Ackermann.

3.4.2.4 Algoritm pentru eliminarea recursivității liniare [37]

Recursivitatea liniară se caracterizează prin faptul că mai multe apeluri recursive pot apare doar pe ramuri diferite ale execuției programului, adică pe un anumit nivel apare doar un singur apel recursiv.

Recursivitatea se poate transforma întotdeauna în iterație, acest lucru fiind necesar pentru reducerea timpului de execuție și a necesarului de memorie, eficiența programului crescând prin eliminarea operațiilor de salvare de context ce apar în cazul abordărilor recursive.

Având în vedere modul în care funcționează un apel de subprogram, stabilirea corespondenței între varianta recursivă și cea iterativă, în cazul recursivității directe, se poate face prin parcurgerea următoarelor *etape*:

1. La începutul subprogramului se inserează o secvență de instrucțiuni prin care se declară o stivă și se inițializează ca fiind vidă. În cazul cel mai general, pe stivă se vor păstra valorile parametrilor formali, variabilele locale, indicele adreselor de revenire și eventual valoarea funcției (dacă subprogramul este de tip function), pentru fiecare apel recursiv. Se pot folosi mai multe stive, corespunzătoare fiecărui element.
2. Primei instrucțiuni executabile din subprogram i se atașează eticheta 1.

În continuare, se înlocuiește fiecare apel recursiv cu un grup de instrucțiuni, respectându-se următoarele reguli:

3. Se depun pe stivă valorile parametrilor transmiși prin valoare și a variabilelor locale. Pointerul la vârful stivei se consideră ca fiind global
4. Se creează eticheta i și se memorează i în stivă. Valoarea i va fi folosită ulterior ca adresă de revenire. Această etichetă este plasată în program în conformitate cu regula 7.
5. Se evaluează argumentele corespunzătoare transferului prin valoare și se atribuie aceste valori parametrilor formali.
6. Se inserează un salt necondiționat la eticheta 1.
7. Dacă subprogramul este de tip procedure, se va atașa eticheta creată prin regula 4 instrucțiunii care urmează imediat saltului necondiționat

https://biblioteca-digitala.ro / https://unibuc.ro

(dacă această instucțiune este deja etichetată, se schimbă această etichetă şi toate referințele la ea prin eticheta creată de regula 4). Dacă subprogramul este o funcție, se ataşează eticheta i creată la punctul 4 unei instrucțiuni ce preia de pe stivă valoarea funcției şi se adaugă codul din subprogramul recursiv ce prelucrează această valoare.

Pentru fiecare instrucțiune care semnifică revenirea din subprogramul recursiv se efectuează :

8. Dacă stiva recursivității este vidă se revine în subprogramul apelant
   altfel
8.1 Se preia de pe stivă indicele adresei create cu regula 4 şi se atribuie valoarea indicelui unei variabile neutilizate.
8.2 Se preiau de pe stivă valorile parametrilor transmişi prin valoare şi a variabilelor locale.
8.3 Se evaluează valoarea expresiei care constituie valoarea returnată de funcția recursivă, această valoare depunându-se pe stivă.
8.4 Dacă instrucțiunea de sfârşit este etichetată, se execută salt necondiționat la eticheta creata la etapa 4.

Precizăm faptul că *subprogramele ce nu sunt liniar recursive se pot transforma echivalent devenind liniar recursive* şi apoi se poate aplica algoritmul de mai sus.

3.4.2.5 Concluzii [53]

Prezentam câteva avantaje şi dezavantaje ale recursivității şi locul pe care îl ocupă între tehnicile şi metodele de programare.

Recursivitatea este strâns legată de iterație, presupunând execuția repetată a unei porțiuni de program, dar spre deosebire de aceasta, condiția de terminare a repetiției fiind verificată în cursul execuției secvenței respective (recomandabil cât mai la începutul subprogramului recursiv). Deosebit de importantă este satisfacerea *consistenței* unui subprogram recursiv.

În general, unui subprogram i se asociază un set de obiecte locale, ce nu au semnificație în afara lui. La fiecare apel recursiv al subprogramului se creează un nou set de asemenea obiecte locale care, deşi au acelaşi nume ca şi cele corespunzătoare lor din instanța anterioară a subprogramului, au valori distincte depuse pe stivă pe nivelul următor. *Datorită mecanismului de stivă, identificatorii se referă întotdeauna la setul de obiecte cel mai recent creat şi, considerând generarea apelurilor într-un*

https://biblioteca-digitala.ro / https://unibuc.ro

sens, execuția se va face în sens invers, adică ultimul apel generat va fi primul executat.

În practică este necesar nu numai ca adâncimea recursivității să fie finită, ci și să fie relativ mică, deoarece fiecare apel recursiv necesită memorarea instanței programului. Deși într-o abordare recursivă de obicei se simplifică scrierea programelor, acestea preluând în mod natural procesarea unor date (definite eventual recursiv), algoritmii fiind caracterizați prin exprimări compacte și clare, nu trebuie uitat că operațiile cu stiva presupun un consum suplimentar de timp și memorie. De aceea, în multe situații, „disputa" între recursivitate și iterație este soluționată în favoarea celei de a doua. Justificăm acest lucru prin doua exemple clasice.

Exemplul 1. Pentru șirul lui Fibonacci varianta recursivă este mai ineficientă deoarece se calculează în mod inutil (dar necesar datorită recursivității) de mai multe ori același termen. Dacă, considerăm fib:N→N

$$fib(n) = \begin{cases} 1 & , \text{dacă } n<2 \\ fib(n-2)+fib(n-1) & , \text{dacă } n\geq 2 \end{cases}$$

pentru n=7 se va calcula fib(0) de 8 ori, fib(1) de 13 ori, fib(2) de 8 ori, fib(3) de 5 ori, fib(4) de 3 ori, fib(5) de 2 ori și fib(6) o dată. Comentariile sunt de prisos.

Obsevație. O astfel de recursivitate se numește recursivitate în cascadă

Exemplul 2. Binecunoscuta funcție a lui Ackermann are valori foarte mari chiar și pentru valori mici ale parametrilor. De exemplu $Ack(4,2) \approx 10^{20000}$. Pe lângă faptul că este necesară implementarea operațiilor pe numere mari, varianta recursivă necesită timp de calcul mare (datorită numărului mare de autoapeluri) și un spațiu de memorie prohibitiv, datorită adâncimii arborelui de apeluri. Câteva formule cunoscute, ca:

$Ack(1,n)=1+n$;

$Ack(2,n)=2*n+3$;

$Ack(3,n)=2^{(n+3)}-3$; ne previn că funcția are o creștere rapidă.

Pentru m=3 și n=8, în varianta iterativă, se obține rezultatul 2045, pe când implementarea recursivă va conduce la mesajul de eroare „Stack overflow error".

https://biblioteca-digitala.ro / https://unibuc.ro

Totuşi, nu trebuie minimizate importanţa şi utilitatea tehnicii recursivităţii în programare. Este greu să ne imaginăm unele aplicaţii grafice (exemplu: fractali) sau chiar unele aplicaţii clasice (exemplu:problema turnurilor din Hanoi) fără recursivitate. Dacă memoria necesară rulării programului se încadrează în segmentul de stivă tehnica recursivităţii se poate aplica cu success, combinată cu oricare dintre metodele de programare tratate în acest capitol.

De remarcat că există variante recursive standardizate ale rutinei backtracking, că abordarea divide et impera, prin însăşi caracteristica ei de bază, permite implementarea recursivă (sunt mulţi algoritmi foarte performanţi în timp şi spaţiu; exemplu: Quicksort) şi de asemenea multe probleme ce aplică metoda greedy sau programarea dinamică folosesc şi recursivitatea.

3.4.3 Metoda divide et impera [18, 37, 53, 44]

„Divide et impera" este o metodă generală de elaborare a algoritmilor ce se bazează pe un *principiu* simplu şi anume: *descompunerea problemei în două sau mai multe probleme de acelaşi tip, dar de dimensiuni mai mici, rezolvarea lor şi combinarea soluţiilor acestora pentru a obţine soluţia problemei iniţiale.* Subprogramele se rezolvă fie direct, dacă dimensiunea lor permite aceasta, fie se împart la rândul lor în alte subprobleme, utilizând acelaşi procedeu.

Algoritmul general al metodei fiind de natură repetitivă, admite o implementare elegantă în mod recursiv fapt pentru care, deşi recursivitatea nu este obligatorire, în cele mai multe cazuri este preferabilă.

Etapele rezolvării unei probleme prin această metodă sunt:

1.(***Divide***) Descompunerea problemei în două sau mai multe subprobleme.

2.***(Rezolvarea subproblemelor)*** Se rezolvă fiecare din subprobleme direct – dacă acestea sunt simple – sau prin reducerea acestora la alte subprograme de acelaşi tip, dar mai simple.

https://biblioteca-digitala.ro / https://unibuc.ro

3.(***Impera***) Combinarea soluțiilor subproblemelor pentru obținerea soluției problemei inițiale.

Observații.

1. Nu orice problemă poate fi rezolvată prin utilizarea acestei metode deoarece condiția ca problema să admită o descompunere repetată în subprobleme de aceeași natură (dar mai simple, evident) este satisfăcută de un număr relativ mic de probleme.
2. Pentru implementarea recursivă, o *procedură generală* a metodei este:

```
Procedure DIVIDE_ET_IMPERA (d:dim, Sol);
 array ds[N1MaxSubpb]
 integer ns, i
 array s[N2MaxSubpb]
     if d<eps then REZOLVA (d, SolP)
         else
                 DIVIDE (d,ns,ds);
                 for i=1 to ns  DIVIDE_ET_IMPERA (ds[i],s[i]);
                 repeat
                 COMBINA (ns,s,Sol);
     endif
end
```

In aceasta procedura:

* tipul dim este de obicei un tip simplu (integer sau subtip al tipului integer) dar poate fi și structurat în funcție de natura problemei deoarece dimensiunea problemei se poate caracteriza prin mai multe valori (de exemplu, la Quicksort, dimensiuna vectorului de sortat se caracterizează prin două valori: poziția de început și cea de sfârșit a subvectorului ce trebuie sortat).
* parametrul Sol exprimă rezultatul (soluția) problemei .
* NrMaxSubpb este numărul maxim de subprobleme în care se poate descompune problema dată.
* ds[i] semnifică dimensiunea subproblemei a i-a (considerăm că problema s-a descompus în ns subprobleme) iar s[i] este soluția subproblemei de ordin i .
* eps este o valoare cunoscută, relativ mică, reprezentând dimensiunea maximă a unei probleme de tipul inițial ce poate fi rezolvată în mod

https://biblioteca-digitala.ro / https://unibuc.ro

direct (fără descompunere în subprobleme). Dacă tipul dim nu este numeric, inegalitatea „d < eps" va trebui explicitată.

* REZOLVA este o procedură cu antelul:
  Procedure REZOLVA (d:dim, Sol);
  ce calculează soluția Sol a unei probleme elementare, de dimensiune d.
* DIVIDE este o procedură cu antelul:
  Procedure DIVIDE (d:dim, ns:t1; ds:dim);
  unde t1= 1.. NrMaxSubpb;

Preluând datele problemei (ca variabile globale), în funcție de d, procedura DIVIDE împarte problema de dimensiune d în ns subprobleme, de dimensiuni reținute în ds (ns şi ds sunt parametrii de tip variabilă). Apelul acestei proceduri corespunde etapei 1.

* Instrucțiunea For corespunde etapei 2 din algoritmul general.
* COMBINA este o procedură cu antetul:

Procedure COMBINA (ns: t1; s: t2; Sol);
unde t2= array [t1]
Ea asigură combinarea celor ns subprobleme ce au soluțiile în vectorul s, corespunzătoare etapei a 3-a.

3.O aceeaşi problemă, rezolvată cu divide et impera prin descompunerea în aceleaşi subprobleme rezolvată în manieră iterativă şi recursivă, poate să dea diferențe în ceea ce priveşte necesarul de memorie (la rezolvarea recursivă folosindu-se de regulă mai multă memorie), timpul de execuție (dacă la recursivitate unele prelucrări se execută de mai multe ori) şi ordinea în care se rezolvă subproblemele.

4. În cazul implementării recursive este important să asigurăm condițiile de terminare (d<eps).
Procesul de descompunere în subprobleme se opreşte dacă şi numai dacă acestea permit o rezolvare directă.

5. Algoritmii divide et impera sunt în general rapizi, deoarece prin descompunere, complexitatea prelucrărilor şi lungimea subproblemelor scade în progresie geometrică.

6. Algoritmul de căutare binară este un exemplu în care metoda divide et impera se aplică în mod particular pentru că problema nu se rezolvă clasic (prin descompunere în alte subprobleme care se rezolvă şi a căror soluții se combină ulterior) ci prin *reducerea* la una din cele două subprobleme (căutare în prima jumătate a vectorului, respectiv în a doua jumătate a vectorului). Acest raționament este de tip *divide et impera degenerativ*, această particularitate fiind benefică din punctul de vedere al

https://biblioteca-digitala.ro / https://unibuc.ro

vitezei de execuție (se câștigă timpul în care soluțiile eventualelor subprobleme se combină pentru obținerea soluției problemei inițiale și mărimea subproblemei la care se reduce problema de rezolvat scade exponențial).
7. Varianta recursivă a tehnicii divide et impera poate fi considerată o subtehnică a tehnicii recursivității.

Vom exemplifica această tehnică pentru **Problema turnurilor din Hanoi**

Considerăm trei tije numerotate cu 1, 2 și 3 și n discuri de diametre diferite, plasate inițial pe tija 1 în ordinea descrescătoare a diametrelor, considerând sensul de la bază spre vârf. Se cere să se tipărească toate mutările ce trebuie efectuate pentru a trece toate discurile de pe tija 1 pe tija 2 folosind ca tijă de manevră tija 3 și respectând următoarele reguli:

- la fiecare mutare se mută un singur disc;
- orice disc poate fi așezat ca bază, pe o tijă goală
- dacă un disc, D_1, se așează peste un altul D_2, diametrul discului D_1 trebuie să fie mai mic decât diametrul discului D_2.

Analiza problemei: Dacă n=1 este suficientă o mutare a singurului disc de pe tija 1 pe tija 2; pentru n>1 se observă că trecerea celor n discuri de pe tija 1 pe tija 2 folosind tija intermediară 3 este compusă din următoarele șiruri de mutări:
a) trecerea celor n-1 discuri din vârful tijei 1 pe tija 3, utilizând ca tijă intermediară tija 2;
b) mutarea discului rămas pe tija 2;
c) mutarea a n-1 discuri de pe tija 3 pe tija 2 folosind tija 1 ca tijă intermediară.

Considerăm că o mutare se precizeaza prin „i→j", unde $i,j \in \{1,2,3\}$, $i \neq j$, cu semnificația „discul din vârful tijei i se mută pe tija j". Dacă i,j,k sunt cele trei tije (distincte) într-o ordine oarecare, i+j+k=6. Notând cu H(x,i,j) șirul de mutări necesare pentru a muta x discuri de pe tija i pe tija j se observă că:

$$H(x,i,j)=\begin{cases} i\rightarrow j & \text{, dacă } x=1 \\ H(x-1,i,6-i-j)\ i\rightarrow j\ H(x-1,6-i-j,j) & \text{,dacă } x>1 \end{cases}$$

https://biblioteca-digitala.ro / https://unibuc.ro

Programul următor redă algoritmul descris mai sus folosind procedura recursivă MUTARI prin care se tipăreşte şirul de mutări a x discuri de pe o tijă sursă ts pe tija destinţie td. Este important ca n citit (în programul principal) să fie strict pozitiv, în caz contrar procedura autoapelându-se până la umplerea stivei de memorie şi programul terminându-se cu eroare. Se presupune definit tipul de date **tija**.

```
Procedure MUTARI (x:integer; ts,td:tija);
if x=1 then write (ts,x '->',td)
    else
        MUTARI (x-1,ts,6-ts-td);
        write (ts,x '->',td);
        MUTARI (x-1,6-ts-td,td);
endif;
end
```

In program această procedură se apelează prin: Mutări(n,1,2)
Implementarea algoritmului descris aici se găseşte în partea a doua capitolul II la paragraful 2.14.2 problema 4.
Observaţie: Problema iniţială a fost descompusă în trei subprobleme de aceeaşi natură, dintre care una elementară iar celelalte două având dimensiuni egale şi mai mici decât ale problemei iniţiale. De remarcat că toate cerinţele problemei iniţiale sunt satisfăcute la orice pas şi de subprobleme.

3.4.4 Metoda greedy [18, 37, 35, 53]

3.4.4.1 Prezentare

Problemele ce se rezolvă cu metoda greedy nu pot fi încadrate într-un algoritm general riguros dar prezintă toate anumite *caracteristici* şi anume:
- trebuie prelucrată o mulţime A cu n elemente
- se cere selectarea unei submulţimi B a lui A sau o ordine de prelucrare a elementelor lui A care să optimizeze o funcţie obiectiv dată sau dedusă. Practic, valoarea funcţiei obiectiv, pentru elementele selectate, va trebui să fie minimă sau maximă.
- se cere o singură soluţie.

https://biblioteca-digitala.ro / https://unibuc.ro

- elementele mulțimii A se prelucrează pe rând, după o eventuală rearanjare a lor în vederea testării lor pentru adăugarea acestora în mulțimea de selecție B.

Un algoritm greedy construiește soluția pas cu pas astfel:

a) B se inițializează cu Ø

b) Cât timp valoarea curentă a funcției obiectiv corespunzătoare lui B poate fi îmbunătățită (mărită sau micşorată în funcție de cerințele problemei) se execută:

- se identifică în A \ B elementul cel mai promițător, x
- B←B ∪ {x}

Considerăm că funcția cu antetul

Function POSIBIL (b:multime;i:integer): boolean;

- Returnează valoarea false în cazul în care în mulțimea B nu mai este posibil a se adăuga un nou element şi i rezultat este zero şi
- Returnează valoarea true în cazul în care în mulțimea A – ale cărei elemente sunt reținute în vectorul x – s-a identificat x[i] ca fiind cel mai promițător element pentru pasul curent (care ajută cel mai bine la îmbunătățirea funcției obiectiv) şi i va fi indicele elementului ce va trebui adăugat în B.

Procedura cu antetul

Procedure ADAUG (y:element;B:multime);

- va executa practic completarea mulțimii B, B←B∪{y}.

O formă generală a metodei este următoarea:

Pasul 1. Aranjează, dacă este util, elementele lui A.
(Presupunem că după pasul 1 A devine {x[1], x[2], ..., x[n]})

Pasul 2. B←Ø; nb←0; {nb=numărul de elemente ale lui B}
Cât timp POSIBIL (B, i) execută
ADAUG (x[i], B);
nb←nb+1
Sf_cât_timp;

Pasul 3. Pentru i=1, nb execută
Tipărește al i-lea element din B
Sf_pentru.

Observații.

1. După cum se poate observa de mai sus, în cei mai mulți algoritmi greedy, mulțimile A şi B sunt ordonate (pe ele se stabileşte o relație de ordine). Mulțimea *A trebuie sortată înainte de pasul 2*, pe când mulțimea

https://biblioteca-digitala.ro / https://unibuc.ro

B se construieşte. Pentru ordonarea elementelor mulțimii A se poate folosi un algoritm cu complexitate medie de $O(n \cdot \log_2 n)$ iar timpul pentru pasul 1 se va aduna la timpul de execuție al pasului 2 (şi 3, ce e nesemnificativ).

2. *Cu metoda greedy nu se determină întotdeauna soluția optimă*. Dacă se cere explicit aceasta, în cazul în care particularitățile problemei acceptă aplicarea unui algoritm greedy ce conduce la soluția optimă, algoritmul trebuie însoțit de o *demonstrație de corectitudine*. De multe ori însă, în practică, se aplică metoda greedy chiar dacă rezultatele furnizate de ea nu sunt optime, dar sunt considerate acceptabile. Este situația în care, pentru problema în cauză, nu există algoritmi de rezolvare decât exponențiali. Aceasta deoarece *algoritmii greedy sunt foarte rapizi, având complexitate polinomială*.

3. Algoritmii greedy despre care nu se poate demonstra că determină (în mod sigur) soluții optime se numesc algoritmi euristici şi formează aşa numitul capitol „greedy euristic" sau „euristica greedy". Pentru ei trebuie să existe anumite caracteristici măsurabile ale compromisului acceptat pentru a se şti cât de aproape de soluția optimă este soluția furnizată sau dacă coincide cu ea (eventual în ce condiții).

4. Nu există un criteriu general conform căruia să se poată stabili că un algoritm greedy rezolvă sau nu o problemă de optimizare. Totuşi, avem două „indicii" ce sugerează rezolvarea unei probleme prin strategia greedy: proprietatea alegerii greedy şi substructura optimă.

Proprietatea alegerii greedy: se poate obține o soluție optimă global prin selectarea unui element optim local.

Pentru demonstrarea acestei proprietăți pentru o problemă se foloseşte de obicei inducția matematică, pornindu-se de la examinarea unei soluții optime global, apoi arătându-se că ea poate fi modificată astfel încât la fiecare pas este realizată o alegere greedy, această alegere reducând problema la una similară dar de dimensiuni mai mici. Prin inducție se arată că o alegere greedy poate fi utilizată la fiecare pas. Demonstrația corectitudinii se reduce, în esență, la verificarea faptului că o soluție optimă evidențiază o substructură optimă.

Substuctura optimă: soluția optimă a unei probleme conține soluțiile optimale ale subproblemelor.

Această proprietate este un indiciu că se poate aplica metoda greedy sau metoda programării dinamice. Pentru demonstrarea faptului

https://biblioteca-digitala.ro / https://unibuc.ro

că o problemă are această proprietate se foloseşte de obicei metoda reducerii la absurd.
5. Pentru situaţia în care o problemă admite în rezolvare o strategie greedy ce conduce la soluţia optimă, soluţia găsită ar putea să nu fie unică. Practic, problema ar putea admite mai multe soluţii optime. Pentru găsirea tuturor soluţiilor optime se poate aplica tehnica backtracking combinată cu strategia greedy.
6. Metoda greedy are un raţionament specific, aplicabil pentru o clasă relativ mare de probleme şi algoritmul propus nu poate fi încadrat într-o altă tehnică sau metodă de programare.
7. Termenul de „mulţime" referitor la A şi B nu este cel clasic, din matematică. Mai potrivită ar fi folosirea de „secvenţă de elemente" cu anumite valori, unele din aceste valori putându-se repeta şi astfel termenul de multiset ar fi cel mai indicat.

3.4.4.2 Bazele teoretice ale metodei greedy [18, 53]

Algoritmii aplicaţi problemelor de optimizare sunt, în general, compuşi dintr-o secvenţă de paşi, la fiecare pas existând mai multe alegeri posibile. De multe ori metoda programării dinamice aplicată pentru determinarea celei mai bune soluţii este prea complicată. Prin strategia greedy se va alege la fiecare pas soluţia ce pare a fi cea mai bună la momentul respectiv. Deci este vorba de o alegere a unui **optim local** (nu întotdeauna unic) cu speranţa că se va ajunge la un **optim global**. Putem decide că un algoritm greedy rezolvă o problemă de optimizare dacă acesta are proprietatea de alegere greedy şi substructură optimă. Aceste două indicii sunt suficiente dar nu şi necesare.

3.4.4.2.1 Matroizi

Structurile combinatoriale numite matroizi constituie bazele teoretice ale metodei greedy şi au fost studiate de Eugene L. Lawler (în Combinatorial Optimization: Networks and Matroids – 1976) şi C. H. Papadimitrion şi K. Steigletz (în Combinatorial Optimization: Algoritms and Complexity) deşi începutul îl constituie un articol din 1935 al lui H. Whitney. O extensie a teoriei matroizilor la *teoria greedoizilor* a fost introdusă de Korte şi Lovasz prin care se generalizează teoria ce va fi prezentată în continuare.

https://biblioteca-digitala.ro / https://unibuc.ro

Definiția 1. Un **matroid** este o pereche ordonată M=(S,I) care satisface următoarele condiții:

1. S este o mulțime finită nevidă;
2. I este o familie nevidă de submulțimi ale lui S, numite *submulțimi independente ale lui S*, astfel încât dacă $B \in I$ și $A \subset B$, atunci $A \in I$. Spunem că *I este ereditară* dacă satisface această proprietate.
   Observație: Pentru orice matroid M=(S,I) , $\emptyset \in I$.
3. Dacă $A \in I$, $B \in I$ și $|A| < |B| \Rightarrow (\exists)\, x \in B-A$ a.î. $A \cup \{x\} \in I$. Spunem că *M satisface proprietatea de schimb.*

Observații:

1. Un tip particular de matroizi sunt cei matriciali, studiați prima dată de Hassel Whitney, în care elementele din S sunt liniile dintr-o matrice dată și o submulțime din S este independentă dacă liniile din mulțime sunt liniar independente în sensul uzual.

De exemplu, dacă T este o matrice pătratică de numere reale, de ordin n, S fiind mulțimea coloanelor lui T, atunci (S,I), cu $A \in I$ este matroid dacă și numai dacă elementele (coloanele) din A sunt liniar independente.

2. Fie S o mulțime finită și I_k = formula tuturor submulțimilor lui S cu cel mult k elemente, unde $k < |S|$. Se demonstrează ușor că (S, I_k) este un matroid.

3. Fie (S,I) un matroid și I' = {A' | S \ S' conține mulțimi maximale $A \in I$}. Atunci se demonstrează ușor că și (S,I') este un matroid.

Consecință: Mulțimile independente maximale ale lui (S',I') sunt exact complementarele mulțimilor independente maximale ale lui (S,I).

Considerăm *matroidul grafic* $M_G = (S_G, I_G)$ definit în termenii unui graf neorientat G=(V,E) după cum urmează:

- Mulțimea S_G, este mulțimea E a muchiilor lui G
- Dacă $A \subseteq E$ atunci ($A \in I_G \Leftrightarrow$ A nu are cicluri).

Aceasta înseamnă că o mulțime de muchii este independentă dacă și numai dacă formează o pădure.

Matroidul grafic este strâns legat de problema arborelui de acoperire minimă.

https://biblioteca-digitala.ro / https://unibuc.ro

Teorema 1. Dacă G=(V,E) este un graf neorientat, atunci $M_G = (S_G, I_G)$ este un matroid.

Demonstrație:

S_G= E este finită.

O submulțime a unei păduri este pădure căci eliminarea unor muchii dintr-o mulțime de muchii ce nu formează cicluri nu poate crea noi cicluri. Deci $\Rightarrow I_G$ *este ereditară.*

Vom arăta că M_G satisface proprietatea de schimb.

Presupunem că A și B sunt păduri ale lui G și $|B| > |A|$. Conform teoremei de caracterizare a arborilor, se deduce că o pădure având K muchii conține exact $|V| - K$ arbori (Pentru a arăta acest lucru se pleacă cu $|V|$ arbori și nici o muchie. Apoi fiecare muchie care este adăugată pădurii, reduce cu 1 numărul de arbori).

Astfel pădurea A conține $|V| - |A|$ arbori și pădurea B conține $|V| - |B|$ arbori.

Cum pădurea B are mai puțini arbori decât A, ea trebuie să conțină un arbore T cu vârfuri aflate în arbori diferiți din pădurea A. Mai mult, T fiind conex, el trebuie să conțină o muchie (u,v) astfel încât vârfurile u și v să fie în arbori diferiți din pădurea A. Din acest motiv (u,v) poate fi adăugată pădurii A fără a forma un ciclu. Deci M_G *satisface proprietatea de schimb.*

Definiția 2. Fie M=(S,I) matroid. Elementul $x \notin A$ se numește *extensie a lui* $A \in I$ dacă x poate fi adăugat la A cu păstrarea independenței; deci x este o extensie a lui A dacă $A \cup \{x\} \in I$.

Exemplu: Fie M_G un matroid grafic. Dacă A este o mulțime independentă de muchii, atunci muchia e este o extensie a lui A dacă și numai dacă $e \notin A$ și $A \cup \{e\}$ nu conține cicluri.

Definiția 3. Fie A o submulțime independentă într-un matroid M. Spunem că A este **maximală** dacă nu are extensii.

Observație: A este maximală dacă nu este inclusă în nici o submulțime independentă mai mare a lui M.

Teorema 2. Toate submulțimile independente maximale dintr-un matroid au aceeași dimensiune.

https://biblioteca-digitala.ro / https://unibuc.ro

Demonstrație:
Presupunem, prin reducere la absurd, că A este o submulțime maximală a lui M şi există o altă submulțime B a lui M cu aceleaşi proprietăți. Atunci, din proprietatea de schimb, rezultă că A este extensibilă la o mulțime mai mare, $A \cup \{x\}$, pentru un $x \in B \setminus A$, ceea ce contrazice presupunerea că A este maximală.

Observație: Fie M_G=matroid grafic pentru un graf conex neorientat G. Fiecare submulțime independentă maximală a lui M_G poate fi un arbore cu exact $|V|-1$ muchii ce leagă toate vârfurile din G. Un astfel de arbore se numeşte *arbore de acoperire a lui G.*

Definiția 4. Matroidul M=(S,I) este **ponderat** dacă are asociată o funcție de pondere $w : S \to R_+$. Funcția de pondere w se extinde la submulțimile lui S prin însumare

$$w(A)=\sum_{x \in A} w(x) \quad \text{pentru orice } A \subseteq S.$$

Exemplu: Dacă w(e) desemnează lungimea unei muchii e dintr-un matroid grafic M_G, atunci w(A) este suma lungimilor muchiilor din A.

3.4.4.2.2 Algoritmi GREEDY pe un matroid ponderat

Numeroase probleme care pentru un algoritm greedy furnizează soluții optime pot fi formulate astfel încât să conducă la determinarea unei submulțimi independente de pondere maximă într-un matroid ponderat. Deci, se dă un matroid M=(S,I) şi se doreşte determinarea unei submulțimi independente $A \in I$ astfel încât w(A) să fie maximă. O astfel de submulțime (independentă şi de pondere maximă posibilă) este *submulțimea optimă a matroidului*. Deoarece $w(x) \geq 0$, $(\forall)\ x \in S$, o submulțime optimă este întotdeauna o submulțime maximală independentă.

Este necesar ca A să fie cât mai mare cu putință.

În cazul arborelui de acoperire minim se consideră un graf neorientat, conex G=(V,E) şi funcția lungime w astfel încât w(e) este lungimea (pozitivă) a muchiei e. Se cere determinarea unei submulțimi de muchii care uneşte toate vârfurile şi are lungimea totală minimă. Pentru a privi aceasta ca o problemă de determinare a submulțimii optime a unui matroid, fie M_G cu funcția de pondere w', unde $w'(e)= w_o - w(e)$ şi w_o se alege astfel încât $w_o > lmax$, $lmax=\max\{w(e) \mid e \in E\}$. În acest matroid

https://biblioteca-digitala.ro / https://unibuc.ro

ponderat $w'(e) > 0$, $(\forall)\ e \in E$ şi o submulţime optimă este un arbore de acoperire având lungimea minimă în graful original.

Mai precis, fiecare submulţime independentă maximală A corespunde unui arbore de acoperire şi deoarece

$$w'(A)=(|N|-1)\ w_0 - w(A)$$

pentru orice submulţime independentă maximală A, submulţimea independentă ce maximizează w'(A) trebuie să minimizeze w(A). În consecinţă, orice algoritm care poate găsi o submulţime optimă A într-un matroid arbitrar, poate rezolva problema arborelui de acoperire minimă, numit si arbore partial de cost minim.

În continuare prezentăm un algoritm greedy ce funcţionează pe orice matroid ponderat M=(S,I) cu o funcţie asociată w de pondere pozitivă şi returnează o submulţime optimă A.

Notăm S[M] şi I[M] componentele lui M.

```
GREEDY (M,w)
1: A:=0
2: ordonează S[M] descrescător după ponderile w
3: Pentru fiecare x∈S[M], în ordine descrescătoare după w(x)
        execută
4:      Dacă A∪{x}∈I(M) atunci
5:              A← A∪{x}
        Sfdacă
   Sfpentru
6: Tipăreşte elementele lui A.
```

Algoritmul aplică strategia greedy deoarece ia în considerare pe rând fiecare element $x \in S$ în ordinea descrescătoare a ponderilor şi dacă extensia $A \cup \{x\}$ este independentă, atunci se adaugă x la A ($A \leftarrow A \cup \{x\}$). Prin inducţie se poate demonstra că A obţinută este întotdeauna independentă. Vom demonstra în continuare că A este o submulţime de pondere maximă posibilă, deci A este o submulţime optimă şi pentru a ajunge la acest rezultat vom analiza, în etape, mai multe aspecte.

Timpul de execuţie al algoritmului GREEDY

Fie $n=|S|$. Etapa de sortare se poate realiza în $O(n \cdot \lg n)$.

Linia marcată cu 4 se execută de n ori, o dată pentru fiecare element din S. Fiecare execuţie a liniei 4 necesită verificarea independenţei mulţimii $A \cup \{x\}$, aceasta realizându-se în $O(f(n))$.

Timpul mediu necesar execuţiei algoritmului GREEDY este deci $O(n \cdot \lg(n) + n \cdot f(n))$.

https://biblioteca-digitala.ro / https://unibuc.ro

Lema 1. (*Matroizii evidențiază proprietatea alegerii greedy*).

Fie M=(S,I) matroid ponderat cu funcția de pondere w, și presupunem S ordonat descrescător după ponderi.

Fie x primul element din S astfel încât {x} este independentă, dacă un astfel de element x există.

Dacă x există, atunci există o submulțime optimă A a lui S care îl conține pe x.

Demonstrație.

Cazul I: nu există $x \in S$ astfel încât {x} este independentă, atunci singura submulțime independentă este Ø.

Cazul II: S conține elemente x cu proprietatea {x} -independentă .

Fie x elementul din S cu w(x) maxim și {x} independentă.

Fie $B \neq \emptyset$, B - optimă

Presupunem că $x \notin B$ (altfel luăm A=B, și demonstrația se încheie). Nici un element din B nu are ponderea mai mare ca w(x).

$$\left.\begin{array}{l} B \in I \\ I \text{ este ereditară} \\ y \in B \end{array}\right\} \Rightarrow y \text{ este independentă}$$

Alegerea făcută asupra lui x asigură

$w(x) \geq w(y)$ pentru orice $y \in B$.

Mulțimea A se construiește astfel:

- inițial $A=\{x\}$

Din alegerea lui x, A este independentă.

- Folosind proprietatea de schimb în mod repetat, se găsește un nou element al lui B care poate fi adăugat la A pînă când $|A| = |B|$, astfel încât independența lui A este mereu păstrată.

Atunci,

$$\left.\begin{array}{l} A=B-\{y\}\cup\{x\} \text{ pentru } y \in B \text{ și deci } w(A)=w(B) - w(y) + w(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w(x) \geq w(y) \end{array}\right\} \Rightarrow$$

$$\Rightarrow w(A) \geq w(B)$$

Deoarece B este optimă, A trebuie să fie de asemenea optimă și cum $x \in A \Rightarrow$ q.e.d.

Lema 2. Fie M=(S,I) matroid, $x \in S$ a.î. x nu este o extensie a lui Ø. Atunci x nu este o extensie a nici unei submulțimi independente A a lui S.

Demonstrație: Se folosește metoda reducerii la absurd. Presupunem deci că x este o extensie a lui A dar nu extensie pentru Ø.

https://biblioteca-digitala.ro / https://unibuc.ro

$\Rightarrow A \cup \{x\}$ independentă
I este ereditară .
$\Rightarrow \{x\}$ independentă $\Rightarrow$ x este extensie pentru Ø

Acastă contradicție cu ipoteza făcută arată că aceasta a fost falsă $\Rightarrow$ q.e.d.

Consecință. Lema arată că orice element care nu poate fi folosit la un anumit moment, nu mai poate fi folosit niciodată ulterior. Deducem deci că algoritmul GREEDY nu poate genera erori datorită trecerii peste elementele inițiale din S care nu sunt extensii ale lui Ø, deoarece ele nu mai pot fi folosite niciodată.

Lema 3. (*Matroizii pun în evidență proprietatea de substructură optimală*).

Fie x primul element ales de algoritmul GREEDY din S pentru matroidul ponderat M=(S,I). Problema rămasă, de determinare a unei substructuri independente de pondere maximă care îl conține pe {x} se reduce la găsirea unei submulțimi independente de pondere maximă a matroidului ponderat M'=(S',I') unde

$$S' = \{ y \in S \mid \{x, y\} \in I \},$$
$$I' = \{ B \subseteq S - \{x\} \mid B \cup \{x\} \in I \}$$

și funcția de ponderare pentru M' este funcția de ponderare pentru M restricționată la S'.

Observație. M' se numește *contracția lui M prin elementul x*.

Demonstrație. Fie A o submulțime independentă de pondere maximă a lui M, care-l conține pe x. Atunci $A' = A \setminus \{x\}$ este o submulțime independentă a lui M'.

Invers, orice submulțime independentă A' a lui M' conduce la o submulțime independentă $A = A' \cup \{x\}$, a lui M.

Cum în amble cazuiri avem $w(A) = w(A') + w(x)$, o soluție de pondere maximă în M, care îl conține pe x, conduce la o soluție de pondere maximă în M și invers.

Teoremă (Corectitudinea algoritmului GREEDY pe matroizi)

Dacă M=(S,I) este matroid ponderat cu funcția de ponderare w, atunci apelul GREEDY(M,w) returnează o submulțime optimă.

Demonstrație: Conform lemei 2, elementele care nu sunt considerate inițial deoarece nu sunt extensii ale lui Ø, nu mai sunt folosite. Odată ce primul element x este selectat, din lema 1 rezultă că algoritmul GREEDY nu poate greși adăugând pe x la A pentru că există o submulțime optimă ce îl conține pe x. Din lema 3 deducem că problema rămasă este una de

https://biblioteca-digitala.ro / https://unibuc.ro

determinare a unei submulțimi optime în matroidul M', care este contracția lui M prin x.

După ce procedura GREEDY atribuie lui A pe {x}, toți ceilalți pași pot fi interpretați ca acționând în matroidul M' = (S', I'), deoarece B este independentă în M' dacă și numai dacă B ∪ {x} este independentă în M, pentru orice B∈I'.

Deci operațiile din algoritmul GREEDY vor determina o submulțime independentă de pondere maximă pentru M' și toate operațiile algoritmului GREEDY vor găsi o submulțime independentă, de pondere maximă, pentru M.

Nu totdeauna strategia GREEDY conduce la solutie optima.

Există multe probleme a căror rezolvare exactă impune cu necesitate un algoritm exponențial. Este știut însă faptul ca astfel de algoritmi, chiar și pentru un volum de date relativ mic, au un timp de calcul mult prea mare pentru a fi considerați utili în practică. În aceste situații este preferabil să renunțăm la optimalitate pentru a aplica algoritmi polinomiali ce obțin soluții suficient de apropiate de cea optimă. De multe ori, în astfel de cazuri, este utilă metoda greedy.

Vom exemplifica în continuare, pe scurt, câteva astfel de probleme:

Exemplul 1. Se cere să se găsească un traseu al unui cal pe o tablă de șah, cu n linii și n coloane (n = 3..25 dat) care, pornind din poziția (1,1) să acopere toate cele nxn pătrățele, trecând prin fiecare o singură dată.

Metoda pentru determinarea soluției optime este backtracking dar aceasta, pentru n>7, are un timp de calcul foarte mare. Se poate concepe un algoritm ce aplică strategia greedy în care, la fiecare pas se alege acea mutare care plasează calul într-o poziție din care, la următoarea mutare, vor exista un număr minim de posibilități de a muta din nou. Deși, la prima vedere, criteriul nu pare foarte puternic, algoritmul conduce în practică la rezultate excelente, lungimea traseului obținut fiind, în general, foarte aproape de nxn, iar pentru unele valori ale lui n, obținându-se chiar soluția optimă. O explicație posibilă ar fi faptul că piesa ocupă mereu poziții cât mai puțin accesibile, care la mutări ulterioare ar fi mai greu de atins.

Exemplul 2. Problema clasică a comis-voiajorului cere de fapt generarea unui ciclu hamiltonian de cost minim într-un graf ponderat fapt pentru care, cu backtracking, se pot genera toate ciclurile elementare memorându-se mereu cel cu suma minimă a ponderilor muchiilor ce-l compun. Acest algoritm nu are o utilitate practică, lucrând, chiar și cu

https://biblioteca-digitala.ro / https://unibuc.ro

îmbunătățiri, într-un timp prea mare (are complexitate exponențială). Pentru a găsi soluția exactă nu există algoritmi polinomiali și de aceea vom renunța la optimalitate pentru găsirea unui ciclu hamiltonian cu un cost total cât mai mic (nu neapărat minim). În această situație strategia greedy este utilă deoarece, pornind de la un nod inițial, se poate construi un ciclu elementar în care, la fiecare pas, selectăm acel nod, neselectat încă, cel mai apropiat de nodul curent. Acest compromis sugerează următorul algoritm:

- se citește matricea ponderilor și nodul inițial v_1 (al cărui grad este mai mare ca 1); nodul curent devine v_1;
- cât timp se mai pot introduce noduri în traseul curent (cu un capăt în v_1 și celălalt în nodul curent) se execută:
  - dintre nodurile adiacente nodului curent se alege cel aflat la distanța minimă de nodul curent;
  - se introduce acest nou nod în drum și se actualizează nodul curent ca fiind nodul nou introdus;
  - se actualizează lungimea totală a lanțului;
- se completează lanțul cu muchia [nod_curent,v_1], se calculează lungimea totală a ciclului (lungimea totală a lanțului adunată cu costul muchiei [nod_curent,v_1]) și se tipăresc rezultatele dorite.

Algoritmul poate fi îmbunătățit prin aplicarea repetată a rutinei descrise mai sus considerând, pe rând, ca noduri inițiale toate nodurile din graf cu grad cel puțin 2. Se va păstra în memorie doar cel mai bun (scurt) traseu, eventual abandonând acele trasee ce se dovedesc mai lungi, chiar înainte de a le finaliza.

Se observă că, pentru unele situații, algoritmul nu furnizează nici o soluție, deși problema ar putea avea chiar mai multe soluții. Faptul că este foarte rapid și probabilitatea de a furniza o soluție foarte apropiată de cea optimă (sau uneori chiar cea optimă) este relativ mare face ca algoritmul descris mai sus să fie util în multe situații practice.

Timpul de execuție în cazul cel mai defavorabil are ordinul $O(n \cdot m^2)$ unde m reprezintă numărul de muchii iar n este numărul nodurilor.

O altă strategie a algoritmilor euristici constă în modificarea parțială (locală) a unei soluții date.

https://biblioteca-digitala.ro / https://unibuc.ro

3.4.5 Programarea dinamică [18, 37, 38, 53]

3.4.5.1 Prezentare

Programarea dinamică este o tehnică de programare de nivel înalt în sensul că necesită o îmbinare între intuiție și rigurozitate matematică. Alături de GREEDY, ea conduce, de cele mai multe ori, la un timp de calcul polinomial. În plus, prin programarea dinamică se obține întotdeauna **soluția optimă**. Ca și celelalte tehnici de programare ea nu se poate aplica tuturor problemelor, ci doar celor de optimizare ce îndeplinesc anumite **condiții**, mai exact:

a) Rezultatul cerut de problemă se obține în urma unui șir de decizii $D_1,D_2,\ldots,D_n$. În urma deciziei D_i, sistemul evoluează din starea S_{i-1} în starea S_i, unde $i=\overline{1,n}$.

b) Trebuie să se verifice „**principiul optimalității**". El reflectă, intuitiv, faptul că atunci când dorim să obținem o situație favorabilă dintr-un anumit punct de vedere este bine ca ea să rezulte tot din situații favorabile.

Un exemplu simplu și clar este exprimat de algoritmul lui Roy-Floyd. Fie un graf orientat dat, cu n noduri și i,j două vârfuri ale sale. Dacă drumul minim de la i la j trece prin vârful k, atunci subdrumurile de la i la k și de la k la j sunt minime. Demonstrația este imediată și folosește metoda reducerii la absurd. Putem concluziona că dacă un drum de la i la j este optim (are lungimea minimă) atunci orice subdrum al său este optim.

Considerăm că $D_1,D_2,\ldots,D_n$ este un șir de decizii care conduce sistemul în mod optim din starea inițială S_0 în starea finală S_n.

Principiul optimității (forme de exprimare):

1) $D_k,D_{k+1},\ldots,D_n$ este un șir de decizii ce conduce optim sistemul din starea S_{k-1} în starea S_n , $(\forall)$ k, $1\le k\le n$;

2) $D_1,D_2,\ldots,D_k$ este un șir de decizii care conduce optim sistemul din starea S_0 în starea S_k , $(\forall)$ k, $1\le k\le n$;

3) $D_1,\ldots,D_k$ și $D_{k+1},\ldots,D_n$ sunt șiruri de decizii care conduc optim sistemul din starea S_0 în starea S_k, respectiv din starea S_k în starea S_n, $(\forall)$ k, $1\le k\le n$;

Programarea dinamică se poate aplica problemelor la care optimul general implică optimul parțial, deci pentru care se verifică principiul de optimalitate. În funcție de forma 1, 2 sau 3 pentru care se verifică principiul optimalității se aplică programarea dinamică metoda înainte, metoda înapoi, respectiv metoda mixtă.

https://biblioteca-digitala.ro / https://unibuc.ro

Observația 1. Implicația „optim general $\Rightarrow$ optim parțial " nu ne asigură că afirmația reciprocă este valabilă. Mai exact, dacă optimul general determină optimul parțial, nu înseamnă că și optimul parțial determină optimul general. Pentru exemplul dat anterior, dacă există un drum de la i la k optim și un altul de la k la j optim, concatenând cele două drumuri nu se obține obligatoriu un drum de la i la j optim. Acest lucru se poate demonstra ușor printr-un contraexemplu.

Rezolvările folosind programarea dinamică sunt în general bazate pe realizarea unei optimizări, de aflare a unui șir optim de transformări conform unui criteriu dat de natura problemei.

Observația 2. Aplicarea principiului de optimalitate în forma 1 semnifică faptul că fiecare decizie D_k depinde de deciziile $D_{k+1},... D_n$ ce urmează să le luăm în continuare. În programarea dinamică metoda înapoi, fiecare decizie D_k depinde de deciziile luate până atunci $D_1,..., D_{k-1}$. Pentru metoda mixtă fiecare decizie D_k va depinde atât de deciziile luate până atunci cât și de cele ce urmează a se lua în continuare.

Etapele rezolvării unei probleme de programare dinamică:

E1. Verificarea principiului de optimalitate în una din cele 3 forme ale sale (caracterizarea structurii unei soluții optime);

E2. Scrierea unor relații (de obicei de recurență).care exprimă modul de obținere al optimului general din optime parțiale;

E3. Scrierea programului.

a) Se calculează bottom-up valoarea unei soluții optime

b) Dacă se dorește și soluția propriu-zisă, din informațiile calculate (de obicei stocate în memorie), se construiește top-down o soluție optimă. În această fază se aplică de obicei un algoritm recursiv prin care se efectuează o parcurgere în sens invers a secvenței optime de decizii calculate anterior.

Observația 3. Optimul general dorit a se calcula se caută între sau combinând optimele parțiale pe care le reținem la fiecare pas (fără a se calcula de mai multe ori, așa cum se procedează, de exemplu, la divide et impera sau recursivitate). Căutarea în acest mod este redusă considerabil deoarece se evită investigarea tuturor posibilităților și extragerea celei optime dintre ele. Relațiile de recurență ce apar se calculează în mod iterativ.

https://biblioteca-digitala.ro / https://unibuc.ro

Observația 4. Aplicarea programării dinamice conduce la o singură soluție, optimă pentru un anumit criteriu dar nu întotdeauna unica ce respectă condiția de optim.

3.4.5.2 Concluzii [53]

Programarea dinamică este o metodă de proiectare a algoritmilor care poate fi utilizată în problemele de optimizare în cazul în care *soluția poate fi privită ca rezultatul unei succesiuni de decizii ce respectă principiul optimalității.*

După cum s-a observat , algoritmii divide et impera împart problema în subprobleme independente, rezolvă (recursiv) subproblemele și apoi combină soluțiile lor pentru găsirea soluției problemei inițiale. Spre deosebire de această abordare, programarea dinamică este aplicabilă și eficientă atunci când subproblemele nu sunt independente, adică subproblemele au în comun sub-subprobleme. (Aceasta este exprimată de faptul că o decizie D_k luată la un moment dat este influențată de decizii luate anterior și / sau de decizii ce urmează să fie luate). În acest context, un algoritm de tipul divide et impera presupune mai multe calcule decât ar fi necesar dacă s-ar rezolva iterativ sub-subproblemele comune. *Un algoritm de programare dinamică rezolvă fiecare sub-subproblemă o singură dată* și, apoi, memorează soluția acesteia într-un tablou, prin aceasta evitând recalcularea soluției de fiecare dată când sub-subproblema apare din nou și reducând considerabil timpul de execuție pentru program.

Problemele de optimizare pot avea mai multe soluții posibile, caracterizate printr-o anumită valoare, valoare optimă (maximă / minimă) pentru un anumit criteriu dat de natura problemei.

Un *algoritm bazat pe programarea dinamică ce returnează toate soluțiile optime* este, schematic, următorul:

1. Caracterizarea structurii unei soluții optime.
2. Definirea recursivă a valorii unei soluții optime.
3. Calculul valorii unei soluții optime într-o manieră de tip "bottom-up".
4. Construirea soluțiilor optime din informația calculată.

Pasul 4 este necesar doar în cazul în care se doresc toate soluțiile optime, acestea fiind mai multe. Pentru aceasta deseori se păstrează informație suplimentară de la execuția pasului 3.

https://biblioteca-digitala.ro / https://unibuc.ro

Pentru ca programarea dinamică să fie adecvată unei probleme de optimizare aceasta trebuie să se caracterizeze printr-o **substructură optimă** şi să aibă **subprobleme suprapuse**. Există şi o variantă a metodei, numită **memorizare**, în care se utilizează proprietatea de suprapunere a subproblemelor.

3.4.5.2.1 Substructura optimă.

O problemă prezintă substructură optimă dacă o soluţie optimă a problemei include soluţiile optime ale subproblemelor (aceasta este un indiciu că pentru rezolvarea problemei ar putea fi adecvata programarea dinamică sau metoda GREEDY) .

Exemple de probleme cu această proprietate sunt înmulţirea optimă a unui şir de matrice (a se vedea problema 3 a paragrafului 2.14.5 din partea a doua) şi triangularea optimă pentru un poligon convex. Tehnica utilizată pentru a arăta că subproblemele au soluţii optime este clasică: reducerea la absurd. Presupunem că există o soluţie mai bună pentru o subproblemă şi arătăm că aceasta contrazice optimalitatea soluţiei problemei iniţiale.

Investigarea substructurii optime a unei probleme prin iterarea instanţelor subproblemelor acesteia este o bună metodă pentru a găsi un spaţiu potrivit de subprobleme pentru programarea dinamică. De exemplu, după examinarea structurii unei soluţii optime a problemei înmulţirii unui şir de matrice, putem itera şi analiza structura soluţiilor optime ale subproblemelor, a sub-subproblemelor ş.a.m.d. Se observă că toate sunt compuse din subsecvenţe ale şirului, $<A_1,A_2,\ldots,A_n>$. Deci, mulţimea şirurilor de formă $<A_i, A_{i+1}, \ldots, A_j>$ cu $1 \le i < j \le n$ este un spaţiu de subprobleme natural şi rezonabil de utilizat.

3.4.5.2.2 Suprapunerea subproblemelor

Pentru aplicarea metodei este important ca spaţiul subproblemelor să fie relativ mic. Un algoritm recursiv rezolvă mereu aceleaşi subprobleme, în loc să genereze subprobleme noi. De obicei, numărul total de subprobleme distincte este dependent polinomial de dimensiunea datelor de intrare. Când un algoritm recursiv arborează de mai multe ori o aceeaşi problemă, se spune că problema de optimizare are subprobleme suprapuse. Spre deosebire de aceasta, o problemă ce se rezolvă cu divide et impera generează, la fiecare nouă etapă, probleme noi. În general, algoritmii de programare dinamică folosesc suprapunerea subproblemelor,

https://biblioteca-digitala.ro / https://unibuc.ro

urmată de stocarea soluției într-un tablou unde poate fi găsită la nevoie, necesitând pentru regăsire un timp constant.

3.4.5.2.3 Memorizarea

Această variantă a programării dinamice mărește deseori eficiența abordării uzuale a metodei folosind o strategie de tip „top-down". Ideea este de a memoriza algoritmul recursiv natural care este însă în practică ineficient având timp de execuție exponențial în n (n=dimensiunea problemei inițiale).

Ca și în abordarea obișnuită, soluțiile subproblemelor se păstrează într-un tablou dar structura de control pentru completarea tabloului este mult mai asemănătoare algoritmului recursiv.

Un algoritm recursiv memorizat folosește un element al tabloului pentru fiecare soluție a fiecărei subprobleme. (Tabloul poate avea una, două sau mai multe dimensiuni în funcție de numărul de parametrii ce definesc o subproblemă). Fiecare element al tabloului se inițializează cu o valoare specială care indică faptul că respectivul element trebuie completat (nu a fost încă completat). Atunci când subproblema este întâlnită prima oară pe parcursul execuției algoritmului recursiv, soluția sa este calculată și apoi stocată în tablou. După aceasta, ori de câte ori este întâlnită subproblema respectivă, valoarea stocată în tablou este căutată și regăsită. Această abordare presupune că mulțimea parametrilor tuturor subproblemelor posibile este cunoscută și finită și că este stabilită relația dintre pozițiile din tablou și subprobleme.

Un algoritm recursiv memorizat are un timp de execuție polinomial.

În practică, dacă toate subproblemele trebuie rezolvate cel puțin o dată, un algoritm clasic de programare dinamică „bottom-up" îmbunătățește cu un factor constant un algoritm „top-down" cu memorizare, deoarece nu apare efortul suplimentar necesitat de recurență, iar pentru gestionarea tabloului efortul suplimentar este mai mic. În plus, în unele probleme, accesarea în mod regulat a tabloului în metoda programării dinamice poate fi folosită pentru reducerea suplimentară a timpului sau spațiului necesar. Reciproc, dacă unele subprobleme din spațiul subproblemelor nu trebuie rezolvate deloc, soluția cu memorizarea are avantajul de a rezolva numai subproblemele necesare.

Programarea dinamică permite obținerea unei soluții optime fără a considera toate cazurile posibile, algoritmii obținuți având o *complexitate*

https://biblioteca-digitala.ro / https://unibuc.ro

polinomială. Aceasta este o deosebire fundamentală față de backtracking care, deşi se poate aplica şi ea unei probleme de optimizare, este mult mai ineficientă.

Sesizăm şi o asemănare cu metoda greedy, ambele exploatând proprietatea de substructură optimă (soluția optimă a problemei conține soluțiile optime ale subproblemelor). Dacă în cazul metodei greedy fiecare decizie care se ia este independentă de celelalte, remarcabil este faptul că în programarea dinamică o decizie D_k depinde, de cele mai multe ori, de deciziile luate deja şi/sau de cele ce urmează a fi luate. O altă diferență între greedy şi programarea dinamică constă în faptul că metoda greedy generează o singură secvență de decizii în timp ce metoda programării dinamice poate genera mai multe secvențe de decizii (putând calcula mai multe soluții optime globale, eventual în combinație cu recursivitatea).

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL IV Complexitatea algoritmilor [18, 11, 53, 86]

4.1 Generalități

Elaborarea programelor este un proces complex ce necesită parcurgerea următoarelor etape:

1. *Analiza problemei* (definirea datelor de intrare și a celor de ieșire a relațiilor dintre ele și a cazurilor particulare representative).
2. *Elaborarea algoritmilor* (proiectarea structurilor de date necesare, descompunerea problemei în subprobleme relativ independente, determinarea algoritmilor de rezolvare a subproblemelor)
3. *Verificarea corectitudinii algoritmului de rezolvare.* La început se testează algoritmul pe diferite seturi de date de test. Aceste seturi trebuie să acopere toate variantele de execuție a algoritmului, inclusiv situațiile de excepție și să clarifice dacă fiecare subproblemă este corect rezolvată. Aceste testări pot evidenția omisiuni sau erori de concepere a algoritmilor, dar nu garantează corectitudinea lor. După eventuale corecții și/sau îmbunătățiri ale algoritmului se trece la demonstrația formală a corectitudinii algoritmului, fiind necesar un efort matematic complex.
4. *Analiza complexității algoritmului.* Aceasta are scopul de a determina eficiența algoritmilor care rezolvă o aceeași problemă și de a alege dintre ei pe cel optim. Criteriile care definesc eficiența unui algoritm sunt: complexitatea spațială și complexitatea temporală.

Prin ***complexitate spațială*** înțelegem dimensiunea spațiului de memorie folosit la execuția unui program. Un program necesită un spațiu de memorie constant, independent de datele de intrare, și unul variabil a cărui dimensiune depinde de datele de intrare.

În spațiul constant se reține: codul programului, constantele, variabilele simple și structurile de date de dimensiune constantă, alocate static.

Spațiul variabil este compus din spațiul necesar structurilor de date alocate dinamic și din spațiul necesar apelurilor de proceduri și funcții. Sunt situații în care un algoritm corect este practic inutilizabil pentru di-

https://biblioteca-digitala.ro / https://unibuc.ro

mensiuni mari ale datelor de intrare datorită includerii unor proceduri recursive. Algoritmul are deci o complexitate spațială de eficiență minimă.

Progresele tehnicii și ale calculatoarelor fac ca criteriul de eficiență a complexității spațiale să scadă simțitor, prioritar devenind criteriul de complexitate temporală.

Prin ***complexitate temporală*** înțelegem timpul necesar executării programului.

Calculatorul folosit influențează acest timp, pentru că, evident, un program rulează mai rapid pe un calculator de ultimul tip decât pe unul cu viteză de execuție mică.

Pe un calculator secvențial, în care la fiecare moment se execută o singură instrucțiune, timpul necesar programului depinde de numărul operațiilor elementare efectuate de algoritm.

Astfel, pentru analiza complexității temporale a unui algoritm, trebuie la început să determinăm numărul operațiilor elementare ale algoritmului și costul acestora exprimat în unități de timp.

Numim ***operație elementară*** o operație a cărui timp de execuție este independent de datele de intrare. Timpul de execuție a unei operații elementare diferă de la un tip de operație la altul, dar fiind fixat, putem considera că operațiile elementare au timpul mărginit de o constantă c.

Fără a restrânge generalitatea, putem considera că operațiile elementare au același timp de execuție, fiind astfel necesară doar evaluarea numărului de operații elementare și nu și a timpului lor de execuție. Analiza teoretică ignoră factorii ce depind de tipul de calculator folosit sau de limbajul de programare ales, axându-se doar pe determinarea ordinului de mărime a numărului de operații elementare,

Analiza teoretică urmărește determinarea unor funcții care să limiteze superior și inferior comportarea în timp a algoritmilor. Funcțiile depind de caracteristicile relevante ale datelor de intrare.

Complexitatea temporală a unei secvențe S se determină astfel:

1. Dacă S este v←expr, complexitatea temporală a lui S este dată de complexitatea evaluării expresiei din dreapta.
2. Dacă S este S_1; S_2, complexitatea temporală a lui S este maximul dintre complexitatea temporală a lui S_1 și cea a lui S_2.
3. Dacă S este: if c then S_1
   else S_2
   endif

https://biblioteca-digitala.ro / https://unibuc.ro

complexitatea temporală a lui S este suma dintre complexitatea temporală a evaluării condiției c și maximul complexității lui S_1 și a lui S_2.

4. Dacă S este una din formele:
   * while c_1 S_1 repeat
   * do S_1 until c
   * for $v=v_i, v_f, h$
     S_1
     repeat, unde $v_i \leq v_f$, $h>0$,

atunci complexitatea temporală a lui S este produsul dintre numărul maxim de execuții ale lui S_1 și ordinul de complexitate temporală a lui S_1.

Exemplu: Să se analizeze complexitatea temporală a algoritmului de rezolvare pentru următoarea problemă:

Fie a(n), $n \geq 2$, un vector cu n componente reale. Să se determine numărul de a-inversiuni ale vectorului a (a-inversiune este orice pereche $(i,j) \in \{1,...,n\} x \{1,...,n\}$, astfel încât $i<j$ și $a_i>a_j$).

Algoritmul de rezolvare reprezentat în pseudocod este:

```
procedure a_inversiune;
integer r,i,j,n; real a(n)
read n,a(n)
r←0
for i=1,n-1
        for j=i+1,n
                if a_i>a_j then r←r+1
                endif
        repeat
repeat
write"numărul de inversiuni este:" r
stop
end
```

Observații:

1. Timpul de calcul depinde în primul rând de volumul datelor de intrare, notat n. În exemplul de mai sus, n este dimensiunea vectorului a.
2. Nu totdeauna putem ști exact de câte ori se execută o instrucțiune, chiar dacă îl știm pe n.

https://biblioteca-digitala.ro / https://unibuc.ro

În exemplul de mai sus, nu putem şti de câte ori se execută atribuirea r←r+1. Putem însă considera o legătură directă între valoarea lui n şi numărul de execuţii. În funcţie de r, se poate calcula numărul maxim şi numărul minim de atribuiri r←r+1, pentru cazul cel mai defavorabil când a este sortat descrescător şi pentru cazul cel mai favorabil atunci când a este sortat crescător.

Pentru mulţi algoritmi, diferenţa dintre cazul cel mai defavorabil şi cel mediu este mică şi practic insesizabilă la rularea algoritmului pe un calculator performant.

3. Deoarece în practică nu se poate evalua exact timpul de execuţie al unui algoritm, se alege o operaţie reprezentativă pentru acesta, numită **operaţie de bază** şi se estimează de câte ori se execută aceasta în funcţie de n (volumul de intrari). Operaţia de bază este o secvenţă computaţională independentă de caracteristicile de intrare.

Pentru exemplul anterior, operaţia de bază poate fi comparaţia $a_i>a_j$ efectuată de cel mult $(n-1)+(n-2)+...+2+1=\frac{n(n-1)}{2}$ ori sau atribuirea r←r+1 efectuată de cel mult $\frac{n(n-1)}{2}$ ori.

Pentru un algoritm, alegerea unei operaţii de bază este foarte greu de realizat. Ea trebuie să fie reprezentativă pentru algoritmul respectiv şi să permită calculul complexităţii acestuia, pentru ca algoritmul să poată fi comparat cu alţii ce rezolvă problema prin alte metode. De exemplu, pentru generarea permutărilor, alegerea tipăririi unei permutări ca operaţie de bază este nepotrivită deoarece ar însemna că toţi algoritmii au acelaşi timp de execuţie. Aici, cea mai adecvată este alegerea comparaţiei ca operaţie de bază.

4. datele de intrare influenţează semnificativ timpul de calcul. De exemplu, pentru sortarea prin inserţie, dacă vectorul este sortat crescător, sunt necesare n-1 comparaţii, iar dacă de la început este sortat descrescător elementul de pe poziţia i, $i\in\{2,...,n\}$ se compară cu toate cele i-1 elemente din faţă, fiind necesare $1+2+...+n-1=\frac{n(n-1)}{2}$ comparaţii.

Prin acest exemplu concluzionăm că analiza complexităţii temporale a unui algoritm se poate face pentru timpul maxim de rulare, timpul minim de rulare şi uneori pentru timpul mediu de execuţie, corespunzător cazurilor cele mai defavorabile, favorabile, respectiv medii.

https://biblioteca-digitala.ro / https://unibuc.ro

4.2 Noțiuni de bază [18,53]

Noțiunea de timp de execuție poate avea una din următoarele semnificații:

- timp în cazul cel mai nefavorabil
- timp mediu de execuție
- timp în cazul cel mai favorabil

La unii algoritmi, diferența dintre acești timpi este nerelevantă.

Fie A un algoritm și D spațiul datelor de intrare ale sale. Notăm cu $T_A(d)$ timpul de execuție a algoritmului A pentru setul de date $d \in D$. Cu aceste notații avem:

- timpul în cazul cel mai nefavorabil: $\max_{d \in D} T_A(d)$
- timpul în cazul cel mai favorabil: $\min_{d \in D} T_A(d)$
- timpul mediu de execuție: $\sum_{d \in D} f(d) \cdot T_A(d)$, unde f(d) este probabilitatea de apariție la intrarea algoritmului a datei $d \in D$.

Se observă că timpul mediu necesită cunoașterea repartiției probabiliste a datelor de intrare, și deci analiza algoritmului este uneori dificil de realizat.

În cazul cel mai nefavorabil, timpul de execuție asigură o limită superioară a execuției pentru orice date de intrare. În cele ce urmează, ne vom referi doar la determinarea timpului de execuție în cazul cel mai nefavorabil; presupunem că operațiile elementare ce intervin în algoritm au același timp de execuție si de aceea , pentru analiza temporală a algoritmului vom evalua doar numărul acestor operații în funcție de n.

Analiza timpului vizează comportarea asimptotică a algoritmului, adică, comportarea sa pentru valori foarte mari ale lui n. Aceasta face ca în exprimarea numărului de operaţţii să ne intereseze doar termenul dominant, adică termenul care tinde cel mai repede la ∞.

Definiție: Fie $f:N \to (0,\infty)$ o funcție arbitrară. Spunem că algoritmul A este de ordinul lui f(n), notat $O(f(n)) \Leftrightarrow \exists c>0$ și $\exists n_0 \in N$ astfel încât $T_A(n) \leq c\ f(n)$, $\forall n \geq n_0$.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemple.

a. Dacă $T_A(n)=an+b$, $a>0 \Rightarrow T_A(n) \in O(n)$ pentru că $\exists c=a+1>0$ şi $n_0=max(0,[b])\in N$ astfel încât $an+b\leq c_0 n$, $\forall n\geq n_0$.

b. Dacă $T_A(n)= an^2+bn+c$, $a>0$, atunci $T_A(n)\in O(n^2)$ pentru că $\exists$ $c=a+1>O$ şi $n_0=max([b], [c], 0)+1$ astfel încât $an^2+bn+c\leq(a+1)n^2$, $\forall n\geq n_0$.

c. Dacă $T_A(n)=a\cdot b^n+P(n)$, $a>0$, $b>1$ şi P un polinom, atunci $T_A(n)\in O(b^n)$ pentru că $\exists c=a+1$, şi $n_0\in N$ astfel încât $P(n)\leq b^n$, $\forall n\geq n_0$. În acest caz, $a\cdot b^n+P(n)\leq(a+1)\cdot b^n$ şi deci $T_A(n)\leq c\cdot b^n$.

d. Dacă $T_A(n) = a_k n^k+a_{k-1}n^{k-1}+...+a_1 n+a_0$, atunci $T_A(n)\in O(n^k)$.

Demonstraţie: $T_A(n) \leq |T_A(k)| = |a_0+a_1 n+...+a_k n^k| \leq |a_0|+|a_1|n+...+|a_k|n^k \leq (|a_0|+|a_1|+...+ +|a_k|)n^k$, $\forall n\geq 1$.

Alegem $c=|a_0|+|a_1|+...+ +|a_k|>0$ şi $n_0=1$ şi obţinem $T_A(n)\in O(n^k)$.

e. Dacă un algoritm necesită un timp constant de execuţie, notăm cu $T_A(n)\in O(1)$.

Exemplu. Algortimul de rezolvare a ecuaţiei $ax^2+bx+c=0$ cu $a,b,c\in R$ daţi are ordinul de complexitate O(1).

Spunem că:

- A este liniar $\Leftrightarrow T_A(n)\in O(n)$
- A este pătratic $\Leftrightarrow T_A(n)\in O(n^2)$
- A este cubic $\Leftrightarrow T_A(n)\in O(n^3)$
- A este polinomial $\Leftrightarrow T_A(n)\in O(n^k)$, $k\in N^*$
- A este exponenţial $\Leftrightarrow T_A(n)\in O(a^n)$, $a>1$

Un algoritm ce nu este polinomial se va numi exponenţial aşa cum ar fi pentru cazurile $T_A(n)\in O(n!)$, $T_A(n)\in O(n^n)$, etc.

$n! = 1\cdot 2\cdot...\cdot n\geq 2\cdot 2\cdot...\cdot 2 = 2^{n-1}$.

Observaţie: Notaţia asimptotică O defineşte o relaţie de ordine parţială pe mulţimea algoritmilor.

De aici rezultă că putem compara din punctul de vedere al eficienţei, algoritmii ce rezolvă aceeaşi problemă.

Se demonstrează uşor că:

$O(1) \leq O(\lg n)\leq O(n)\leq O(n\cdot \log n)\leq O(n^2)\leq O(n^k)$ pentru $k\geq 2$.

https://biblioteca-digitala.ro / https://unibuc.ro

$O(a^n) \leq O(b^n)$ pentru $1<a<b$.

Următorul tabel ilustrează comportarea unor funcții de complexitate.

O(n)	$O(\log_2 n)$	$O(n \cdot \log_2 n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$	O(n!)
1	0	0	1	1	2	1
2	1	2	4	8	4	2
4	2	8	16	64	16	24
8	3	24	64	512	256	40326
16	4	64	256	4096	65536	20922789888000
32	5	1	1024	32768	4294967296	$26313 \cdot 10^{33}$

Observăm în tabel că în timp ce O(n) și $O(\log_2 n)$ cresc foarte lent, un algoritm exponențial $O(2^n)$,O(n!) se poate folosi în practică doar pentru valori foarte mici ale lui n.

Chiar și algoritmii A cu $T_A(n) \in O(n^k)$ pentru k relativ mare sunt destul de slabi în ceea ce privește eficiența exprimată în timp de calcul.

Exemple:

1. Dacă $T(n)=O(2^n)$ pentru:
   n=10 unui calculator cu 10^9 operații/secundă îi trebuie 18 secunde
   n=50 unui calculator cu 10^9 operații/secundă îi trebuie 13 zile
   n=60 unui calculator cu 10^9 operații/secundă îi trebuie 310 ani
   n=100 unui calculator cu 10^9 operații/secundă îi trebuie $4 \cdot 10^{13}$ ani
2. Dacă $T(n) = O(n^{10})$ pentru:
   n=10 unui calculator cu 10^9 operații/secundă îi trebuie 10 secunde
   n=100 unui calculator cu 10^9 operații/secundă îi trebuie 3 ani
   n=1000 unui calculator cu 10^9 operații/secundă îi trebuie $3 \cdot 10^{13}$ ani

În practică este de preferat să folosim un algoritm polinomial pe un calculator slab decât unul exponențial pe un calculator superperformant.

Apare întrebarea: pentru orice problemă putem găsi un algoritm de rezolvare în timp „polinomial?"

Din păcate nu se poate răspunde afirmativ la această întrebare. Există probleme pentru care nu s-au găsit algoritmi polinomiali de rezolvare cum ar fi: problema rucsacului, problema comis-voiajorului, problema colorării hărților, etc.

Pentru astfel de probleme, din dorința de a avea un timp polinomial, se renunță la optimalitatea soluției. Astfel de metode de

https://biblioteca-digitala.ro / https://unibuc.ro

rezolvare se numesc **euristice** şi sunt de mare actualitate în informatică teoretică.

4.3 Notaţii asimptotice:

Fie $g:N\rightarrow R_+^*$

θ: Pentru funcţia g(n) notăm:

$\theta(g(n))=\{f(n)|\exists c_1,\ c_2\ \in R_+,\ \exists n_0\in N$ a.î. $0\le c_1\cdot g(n)\le f(n)\le c_2\cdot g(n),\ \forall n\ge n_0\}$

Spunem că g(n) este o *margine asimptotică tare* pentru f(n) şi în mod abuziv dar sugestiv notăm $f(n)=\theta(g(n))$.

O: Pentru g(n) notăm:

$O(g(n))=\{f(n)|\exists c>0,\ \exists n_0\in N$ a.î. $0\le f(n)\le c\cdot g(n),\ \forall n\ge n_0\}$

Notăm şi $f(n) = O(g(n))$.

Observăm că θ delimitează o funcţie asimptotică inferior şi superior iar O, indică doar o margine asimptotică superioară pentru o funcţie.

Rezultă că dacă $f(n)=\theta(g(n)) \Rightarrow f(n)=O(g(n))$, echivalent cu $\theta(g(n))\subseteq O(g(n))$.

Ω: Pentru g(n) notăm:

$\Omega(g(n)) = \{f(n)|\exists c>0$ şi $\exists n_0\in N$ astfel încât $0\le c\cdot g(n)\le f(n),\ \forall n\ge n_0\}$

Ω furnizează o delimitare asimptotică inferioară.

Folosind aceste definiţii, se demonstrează următorul rezultat:

Teoremă. Pentru orice f(n), g(n) avem:

$f(n)=\theta(g(n)) \Leftrightarrow f(n)=O(g(n))$ şi $f(n)=\Omega(g(n))$.

o: Pentru g(n) notăm:

$o(g(n))=\{f(n)|\forall c>0,\ \exists n_0>0$ astfel încat $f(n)\le c\cdot g(n),\ \forall n\ge n_0\}$

o desemnează o delimitare superioară care nu este asimptotică tare.

Exemplu. $3n^2=O(n^2)$ este delimitare asimptotică tare, dar $17n = o(n^2)$ nu este asimptotică tare.

ω: Pentru g(n) definim:

$\omega(g(n)) = \{f(n)|\forall c>0,\ \exists n_0>0$ astfel încat $0\le c\cdot g(n)\le f(n),\ \forall n\ge n_0\}$.

ω desemnează o delimitare inferioară care nu este asimptotică tare.

Se observă o oarecare dualitate între O(f) şi Ω(f) în sensul că dacă $f,g:N\rightarrow R_+^*$ avem:

$f(n)\in O(g(n)) \Leftrightarrow g(n)\in\Omega(f(n))$.

https://biblioteca-digitala.ro / https://unibuc.ro

Observație. Din cele prezentate anterior, pentru algoritmul sortării prin inserție avem:

a) timpul minim de execuție corespunde cazului cel mai favorabil, când vectorul este ordonat crescător și avem T(n)=O(n).

b) timpul maxim de execuție corespunde cazului cel mai nefavorabil, când vectorul este sortat descrescător și avem: $T(n)=O(n^2)$; $T(n)=\frac{n(n-1)}{2} = ½\ n^2 - ½\ n < ½\ n^2$, iar c= ½ și $n_0=2$.

c) timpul mediu de execuție se obține astfel:

Presupunem că orice permutare a elementelor vectorului are aceeași probabilitate de apariție, adică orice ordine inițială a elementelor vectorului este egal probabilă. Astfel:

- probabilitatea ca elementul a_i să fie inserat printre elementele $a_1,...,a_{i-1}$ pe o poziție oarecare k=1,i este aceeași $\frac{1}{i}$.
- numărul mediu de deplasări de elemente ce se fac pentru ca a_i să ajungă pe poziția corectă k este $(i-k)\cdot \frac{1}{i}$ (i-k este numărul de deplasări spre dreapta a elementelor).
- numărul mediu total de operații elementare pentru un i fixat va fi:

$$\sum_{k=1}^{i} \frac{1}{i} \cdot (i-k) = \frac{1}{i}\left[i^2 - \frac{i(i+1)}{2} \right] = \frac{i-1}{2}$$

Astfel, pentru a sorta cele n elemente sunt necesare:

$$\sum_{i=2}^{n} \frac{i-1}{2} = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}$$ operații elemetare.

Deci, complexitatea temporală medie este $O(n^2)$ pentru că $T(n)=¼n^2-¼n<¼n^2$, c=¼, $n_0=2$.

https://biblioteca-digitala.ro / https://unibuc.ro

Probleme:

1. Fie T_n numărul de operații elementare necesare unui algoritm pentru a prelucra n date de intrare. Să se determine complexitatea temporală a algoritmului dacă:

$$T_0=T_1=0 \text{ şi} T_n=\frac{n-1}{n}+\frac{1}{n}\sum_{k=1}^{n}(T_{k-1}-T_{n-k})$$

2. Fie a(n) un vector ce conține toate naturalele de la 0 la n mai puțin unul dintre ele ($n>10^6$). Să se determine numărul care lipseşte din a folosind un algoritm de rezolvare care să aibă ordinul de complexitate temporală cât mai mic.
3. Fie a şi b două numere naturale nenule. Să se precizeze efectul următorului algoritm şi complexitatea sa temporală. Să se descrie apoi algoritmul în pseudocod:
   1) aşezăm a şi b pe o linie separate de 10 spații şi formăm sub fiecare câte o coloană conform regulilor:

   a) se împarte a şi apoi numărul de sub el la 2, luând partea întreagă a câtului.

   b) se înmulțeşte b şi apoi numărul de sub el cu 2
   2) repetăm regulile a) şi b) până ce se obține pe coloana lui a valoarea 1
   3) adunăm toate numerele din coloana lui b care corespund pe linie unor numere impare de pe coloana lui a.

Observație finală. Se poate ca timpul de execuție al unui algoritm să depindă simultan de mai mulți parametri, aşa cum ar fi algoritmii ce operează cu grafuri în care timpul depinde atât de numărul de vârfuri, cât şi de numărul de muchii. Notațiile asimptotice de mai sus se generalizează în mod natural şi pentru funcțiile cu mai multe variabile.

Exemplu.

Fie $g:NxN\rightarrow(0,\infty)$

$O(g(m,n))=\{f:NxN\rightarrow(0,\infty)|\exists c>0, \exists m_0,n_0\in N \text{ a.î. } \forall m\geq m_0, \forall n\geq n_0, 0\leq f(m,n)\leq c\cdot g(m,n)\}$

https://biblioteca-digitala.ro / https://unibuc.ro

4.4 Analiza complexitatii tehnicilor de elaborare a algoritmilor [18, 53, 86]

4.4.1 Metoda backtracking

Metoda backtracking se foloseşte în cazul problemelor a căror soluţie se prezintă sub forma $x_1,x_2,\ldots,x_i,\ldots,x_n$ unde $x_i \in A_i$, $i=\overline{1,n}$. Mulţimile A_i, $i=\overline{1,n}$ sunt finite şi ordonate. Dacă ar trebui să generăm toate n-uplele $(x_1,x_2,\ldots,x_n) \in A_1 \times A_2 \times \ldots \times A_n$ pentru a verifica, pe rând, care dintre ele satisfac cerinţele problemei (sunt soluţii) numărul acestora ar fi $|A_1| \cdot |A_2| \cdot \ldots \cdot |A_n|$. Considerând ca operaţie de bază generarea unui n-uplu (distinct de cele generate anterior), numărul $|A_1| \cdot |A_2| \cdot \ldots \cdot |A_n|$ ar constitui o limită superioară pentru determinarea complexităţii unui algoritm backtracking. În problema generării produsului cartezian a n mulţimi M_i cu câte ne_i elemente chiar se atinge această limită superioară, efectuându-se $\prod_{i=1}^{n} ne_i$ generări de n-uple. De asemenea, dacă problema nu are soluţie se explorează toate posibilităţile, timpul de calcul fiind maxim, (exponenţial).

În majoritatea problemelor $A_1 = A_2 = \ldots = A_n = A$. Notăm $|A| = k \in N^*$. Complexitatea unui algoritm backtracking în acest caz va fi $O(k^n)$ şi, dacă k=n, algoritmul va avea ordinul $O(n^n)$.

În concluzie *metoda backtracking necesită un timp de calcul exponenţial.*

Ştim că metoda determină soluţiile prin încercări succesive de extindere a unei soluţii parţiale. În fiecare etapă a căutării pentru care extinderea soluţiei parţiale curente cu siguranţă nu poate conduce la rezultat, se revine la o soluţie parţială anterioară şi se încearcă din nou. Faptul că se renunţă la generarea unui n-uplu inutil implică o reducere a timpului de rulare.

Observaţii:

1. De obicei revenirea la o soluţie parţială anterioară implică evitarea generării mai multor n-uplii.
2. Cu cât revenirea se face pe un nivel mai scăzut în stivă, cu atât numărul de n-uplii evitaţi de a fi generaţi, este mai mare.
   - Dacă soluţia problemei este un vector de n componente, revenirea în stivă de pe nivelul k+1 pe nivelul k este echivalentă cu evitarea generării a (maxim) (n-k+1)! n-uplii.

https://biblioteca-digitala.ro / https://unibuc.ro

- Dacă soluția problemei este un vector de lungime variabilă, maxim n, revenirea în stivă de pe nivelul k+1 pe nivelul k este echivalentă cu evitarea generării a (maxim) 1!+2!+...+(n-k+1)! vectori.
- Dacă soluția problemei este un vector de lungime variabilă, minim n, revenirea în stivă de pe nivelul k+1 pe nivelul k este echivalentă cu evitarea generării a minim 1!+2!+...+(n-k+1)! vectori.

În consecință se recomandă să se restricționeze cât mai la început generarea de n-uplii. Ordinea în care sunt testate condițiile de continuare şi locul în care se efectuează aceasta în cadrul procedurii VALID poate influența destul de mult timpul de rulare, reducându-l de 2, 3 sau chiar 10 ori. Se poate ține seama de probabilitatea ca anumite condiții să fie îndeplinite sau nu de diverse seturi de date de intrare.

3. Metoda backtracking recursiv are dezavantajul de a fi mai puțin eficientă considerând criteriu spațiu deoarece, prin autoapeluri repetate se încarcă inutil stiva calculatorului.

4.4.2 Recursivitatea

Unul din rezultatele remarcabile ale informaticii teoretice cu privire la recursivitate este faptul că pentru orice subprogram recursiv există o secvență iterativă echivalentă. Deci, în cazul cel mai general, se poate transforma programul recursiv în iterație şi estima timpul de execuție după stabilirea, operației de bază.

Se ştie de asemenea că *recursivitatea necesită un spațiu de memorie relativ mare datorită autoapelurilor repetate* fapt ce, în practică, ar putea chiar să conducă la decizia de a se renunța la o implementare recursivă în favoarea uneia iterative. În foarte multe situații (mai ales în cazurile în care subprogramul recursiv are mulți parametrii – în special de tip structurat, transmişi prin valoare – sau numărul de autoapeluri este foarte mare) este mai important să analizăm *complexitatea spațiu* a algoritmului respectiv, decât *complexitatea timp*.

Este remarcabil faptul că atunci când un algoritm conține (cel puțin) o apelare recursivă la el însuşi, timpul său de execuție poate fi descris adesea printr-o recurență. O recurență este o relație (de egalitate sau inegalitate: $\leq$, $<$, $\geq$, $>$) care descrie o funcție exprimând valoarea sa

https://biblioteca-digitala.ro / https://unibuc.ro

pentru argumente mai mici. De exemplu, timpul de execuție în cazul cel mai defavorabil T(n) al procedurii de sortare prin interclasarea a n valori reale poate fi descris prin recurența

$$T(n)=\begin{cases} \Theta(1) & \text{, dacă } n=1 \\ 2\cdot T(n/2)+\Theta(n) & \text{, dacă } n>1 \end{cases}$$

a cărei soluție este $T(n)=\Theta(n\cdot \lg n)$

4.4.3 Divide et impera

Algoritmul general al metodei a fost prezentat in capitolul precedent și, fiind de natură repetitivă, vom considera în continuare o abordare recursivă a sa. Când un algoritm conține un apel recursiv la el însuși, timpul său de execuție poate fi adesea descris printr-o relație de recurență ce estimează întregul timp de execuție al unei probleme de dimensiune n cu ajutorul timpilor de execuție pentru date de intrare de dimemsiuni mai mici, corespunzători rezolvării subproblemelor (de același tip cu problema inițială) problemei date.

O recurență pentru timpul de execuție al unui algoritm de tipul divide et impera se bazează pe cele trei etape definite în descrierea metodei. Notăm cu T(n) timpul de execuție al unei probleme de mărime n. Pentru situația când problema se rezolvă direct (fără descompunere în subprobleme), corespunzând cazului când $n \le c$ unde c este constant, timpul de execuție îl considerăm constant, anume $\theta(1)$. În general problema inițială se împarte în a subprobleme, unde $a \ge 2$ (dacă a=1 algoritmul se încadrează la recursivitate). În cele mai multe situații subproblemele au o dimensiune aproximativ egală dar nu neapărat n/a. Să presupunem deci că dimensiunea unei subprobleme este n/b unde b>1. Atunci putem scrie recurența

$$T(n)=\begin{cases} \theta(1) & \text{, dacă } n \le c \\ a\cdot T(n/b)+D(n)+C(n) & \text{, dacă } n > c \end{cases}$$

unde D(n) = timpul necesar pentru a divide problema în subprobleme
și C(n) = timpul necesar pentru a combina soluțiile subproblemelor în soluția problemei originale.

https://biblioteca-digitala.ro / https://unibuc.ro

Din punct de vedere tehnic, recurența nu este, de fapt, bine definită, deoarece n/b ar putea să nu fie întreg. Totuși, există teoreme ce demonstrează că înlocuirea fiecăruia dintre cei a termeni T(n/b) fie cu $T(\lfloor n/b \rfloor)$, fie cu $T(\lceil n/b \rceil)$, nu afectează comportamentul asimptotic al recurenței.

Vom exemplifica în continuare estimarea timpului de calcul pentru un algoritm cunoscut, care foloseste tehnica divide et impera .

Exemplu. *Sortarea prin interclasare a unui vector x dat, cu n elemente.* Algoritmul, ce urmează îndeaproape paradigma divide și stăpânește este:

Etapa 1. (divide) Se împarte vectorul x în doi subvectori cu câte n/2 elemente (prima jumătate constituie un subvector, a doua jumătate va reprezenta al doilea subvector).

Etapa 2. (stăpânește) Se sortează recursiv cei doi subvectori utilizând sortarea prin interclasare.

Etapa 3. (combină) Interclasează cele souă subșiruri sortate.

Se observă că recursivitatea se oprește când vectorul de sortat are lungimea 1. Operația principală a algoritmului este interclasarea a două șiruri sortate în etapa 3 pentru care în practică se implementează o procedură auxiliară , INTERCLASEAZA (x, p, q, r) unde p, q, r sunt indici ai vectorului x astfel încât $p \le q < r$. Procedura interclasează subvectorii x[p..q] și x[q..r] având ca rezultat subvectorul curent x[p..r].

Pentru determinarea timpului de execuție în cazul cel mai defavorabil se observă că:

- $T(1) = \Theta(1)$ - timp constant
- Pentru n>1, descompunem timpul T(n) după cele 3 etape:
  a) În etapa 1 se calculează mijlocul subvectorului (n DIV 2) deci $D(n) = \Theta(1)$
  b) În etapa 2 se rezolvă recursiv două subprobleme, fiecare de dimensiune n/2, deci timpul va fi $2 \cdot T(n/2)$ (de fapt, mai exact este $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$)
  c) Se apelează procedura INTERCLASEAZA (x, p, q, r) pentru un vector cu n elemente, aceasta consumând $\Theta(n)$ timp de execuție, deci $C(n) = \Theta(n)$

Astfel: $D(n) + C(n) = \Theta(n) + \Theta(1) = \Theta(n)$

https://biblioteca-digitala.ro / https://unibuc.ro

În concluzie

$$T(n) = \begin{cases} \Theta(1) & \text{, dacă } n = 1 \\ 2 \cdot T(n/2) + \Theta(n) & \text{, dacă } n > 1 \end{cases}$$

Vom considera în continuare că D(n) este neglijabil iar n este o putere a lui 2 (în caz contrar îl majorăm la cea mai mică putere a lui 2 mai care decât n, completând, de exemplu, pe ultimele poziții, elementele din x cu valoarea maximă din x, valoare ce se poate calcula cu un timp Θ (n), și în final, elementele adăugate vor fi ignorate). Fie, deci

$$T(n) = \begin{cases} \Theta(1) & \text{, dacă } n = 1 \\ 2 \cdot T(n/2) + a \cdot n & \text{, dacă } n > 1 \end{cases}$$

Se observă că, dacă $n=2^k$, atunci

$T(n) = T(2^k) = 2 \cdot T(2^k/2) + a \cdot 2^k = 2 \cdot T(2^{k-1}) + 2^k \cdot a =$

$= 2 \cdot (2 \cdot T(2^{k-1}/2) + a \cdot 2^{k-1}) + a \cdot 2^k = 2^2 \cdot T(2^{k-2}) + a \cdot 2^k + a \cdot 2^k =$

$= 2^2(2 \cdot T(2^{k-3}) + a \cdot 2^{k-2}) + 2 \cdot a \cdot 2^k = 2^3 \cdot T(2^{k-3}) + a \cdot 2^k + 2 \cdot a \cdot 2^k =$

$= 2^3(2 \cdot T(2^{k-4}) + a \cdot 2^{k-3}) + 3 \cdot a \cdot 2^k = 2^4 \cdot T(2^{k-4}) + 4 \cdot a \cdot 2^k = \ldots$

$= 2^i \cdot T(2^{k-i}) + i \cdot a \cdot 2^k = \ldots = 2^{k-1} \cdot T(2) + (k-1) \cdot a \cdot 2^k =$

$= 2^{k-1}(2 \cdot \Theta(1) + a \cdot 2) + (k-1) \cdot a \cdot 2^k = 2^k \cdot \Theta(1) + a \cdot 2^k \cdot (k-1+1) =$

$= 2^k \cdot (\Theta(1) + a \cdot k)$

Deoarece $k = \log_2 n$ concluzionăm că:

$$T(n) = n(\Theta(1) + a \cdot \log_2 n) = \Theta(n \cdot \log_2 n)$$

4.4.4 Metoda greedy

Un algoritm greedy va alege la fiecare moment de timp soluția ce pare cea mai bună, deci *se face o alegere de optim local în speranța că ea va conduce la un optim global.* În multe cazuri se obțin soluții optime dar, din păcate, nu se poate generaliza această afirmație.

Trebuie menționat faptul că, o dată ce un element a fost ales, el va rămâne în soluția optimă, în timp ce, dacă un element a fost exclus din soluție, el nu va mai fi reconsiderat pentru a fi inclus ulterior în soluția finală. Acest lucru este esențial pentru calculul complexității. Cei mai mulți algoritmi greedy au un *timp de calcul polinomial*, excepție făcând cazurile când prelucrarea unui element (includerea lui în soluție după

https://biblioteca-digitala.ro / https://unibuc.ro

eventuale teste) nu se face în timp constant ci necesită o complexitate oarecare. În general, presupunând că mulțimea din care se face alegerea are n elemente şi soluția are m elemente, $m \leq n$, se vor face m alegeri, la fiecare efectuându-se maxim n teste. Deoarece numărul maxim de teste este $m \cdot n \leq n^2$ rezultă că timpul maxim (pentru cazul cel mai defavorabil) are ordinul $O(n^2)$.

De multe ori este util ca elementele mulțimii A să fie sortate deoarece ordinea acestora în soluție determină optimalitatea soluției găsie. Sortarea se efectuaează la început şi necesită un timp mediu $O(n \cdot \log_2 n)$. Prin urmare, acest timp se adună şi nu influențează rezultatul complexității algoritmului, în sensul că algoritmul rămâne polinomial.

Din prezentarea teoretică a metodei folosind matroizi, reamintim ca pentru un matroid grafic $M_G=(S_G,I_G)$ definit în termenii unui graf neorientat G=(V,E) avem:

- S_G = E = mulțimea muchiilor lui G
- Dacă $A \subseteq E$ atunci $A \in I_G$ dacă şi numai dacă A nu conține cicluri.

Altfel spus, o mulțime de muchii din G este independentă dacă şi numai dacă formează o pădure.

Pe matroidul grafic M_G evidențiem câteva dintre cele mai importante aplicații ale metodei Greedy: parcurgerea grafurilor în lățime şi în adâncime, algoritmul lui Dijkstra pentru determinarea drumurilor minime de la un nod sursă la toate celelalte, algoritmul lui Prim şi algoritmul lui Kruskal pentru determinarea arborelui parțial de cost minim. În toți algoritmii mai sus amintiți strategia greedy conduce la obținerea unei soluții optime global.

4.4.5 Programarea dinamică

Programarea dinamică nu este o tehnică standardizată şi, pentru calculul complexității unui algoritm ce alpică această metodă este indicată o analiză specifică.

În general se poate încerca aplicarea programării dinamice ori de câte ori arborele de recurență corespunzător soluției recursive naturale conține, în mod repetat, o aceeaşi subproblemă iar numărul total de subprobleme este mic.

Prin folosirea memorizării, soluțiile subproblemelor se păstrează într-un tablou, adesea necompletat în totalitate, deoarece indicii tabloului

https://biblioteca-digitala.ro / https://unibuc.ro

au semnificații ce definesc o subproblemă și, între ei, există de cele mai multe ori relații de interdependență; Prin memorizare se reușește transformarea unui algoritm exponențial în unul polinomial.
Pe cazul cel mai general putem concluziona următoarele:

1. Problema admite o substructură optimă și are un număr finit (și relativ mic) de subprobleme suprapuse;
2. O subproblemă se caracterizează prin k parametrii, parametrul al i-lea putând lua n_i valori;
3. Algoritmul de rezolvare a unei subprobleme este de ordinul O(f(n));

Aplicând programarea dinamică (eventual metoda memorizării) vom obține un algoritm de ordinul $O(n_1 \cdot n_2 \cdot \ldots \cdot n_k \cdot f(n))$, care va fi polinomial dacă f(n) este polinomială. De multe ori $n_1=n_2=\ldots=n_k=n$ și deci ordinul algoritmului este $O(n^k \cdot f(n))$.

https://biblioteca-digitala.ro / https://unibuc.ro

https://biblioteca-digitala.ro / https://unibuc.ro

Partea a II-a

Limbaje de programare



- Programare în C şi C++
- Grafica pe calculator

https://biblioteca-digitala.ro / https://unibuc.ro

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL I Elemente de C şi C++

1.1 Introducere [12, 43, 59, 80]

Conceput de BJARNE STROUSTRUP de la laboratoarele BELL şi finalizat la sfârşitul anilor '80 pe baza limbajului C si completat cu elemente de programare orientată pe obiecte, C++ este unul din cele mai puternice şi populare produse software.

Un program în C++ este o **secvenţă de instrucţiuni** de limbaj introdusă printr-un editor de texte şi apoi salvată ca fişier sursă care conţine:

- **comentarii** – linii de program care încep cu //
- **directive de preprocesare** sau **comenzi pentru preprocesor** care includ în fişierul sursă *fişiere antet* necesare compilării condiţionate ale unor zone de program sursă, fişiere care sunt precedate de #.
- **declarări de variabile** şi **funcţii** (nu cer rezervare de spaţiu de memorie)
- **definiţii de variabile** şi **funcţii** – cer rezervare de spaţiu de memorie pentru stocarea datelor sau a codului sursă. Pe lângă funcţii definite de programator, fişierul sursă poate conţine şi funcţii din biblioteca limbajului. Fişierul sursă are extensia .CPP. Acesta este inspectat de compilatorul C++ care semnalează nerespectarea regulilor de programare C++ prin mesaje de eroare afişate pe ecran.

Pentru corectarea erorilor este necesară **editarea fişierului sursă** şi apoi o nouă compilare a programului sursă. După eliminarea erorilor de sintaxă compilatorul transformă instrucţiunile în *cod maşină* (succesiuni de biţi) obţinând *module obiect* asociate. Acestea sunt completate cu module din biblioteca limbajului prin *editorul de legături*, obţinând un fişier executabil cu extensia .EXE, fişier care este stocat pe disc. Programul se execută tastând numele fişierului executabil şi <ENTER>.

Elementele de bază ale limbajului sunt:

- *cuvinte cheie* – cuvinte rezervate pentru declararea tipurilor de date şi a instrucţiunilor,

https://biblioteca-digitala.ro / https://unibuc.ro

- *identificatori* – nume de constante, variabile, câmpuri ale unor structuri definite de utilizatori,
- *constante* – valori fixe reprezentând numere, caractere, şiruri de caractere,
- *operatori* – simboluri folosite pentru specificarea unor operaţii,
- *separatori* – caractere sau şiruri de caractere care separă diferite entităţi ca: blank-uri, tab-uri orizontale sau verticale, linie nouă, pagină nouă, comentarii, etc.

Exemplu.
```
#include <iostream.h>
void main()
{cout <<"Atenţie! Un program foarte simplu";}
```

unde:

- #include – directivă de preprocesare
- caracterele "{" şi "}" definesc blocurile unui program C++
- un program C++ trebuie să aibă o singură dată o linie ce conţine funcţia main()
- fiecare instrucţiune se încheie cu ";"
- şirurile de caractere se includ între ghilimele, iar cele individuale între apostrofuri
- pentru afişarea la ecran, se foloseşte funcţia de ieşire cout
- dirijarea spre ieşire se face cu operatorul de inserare în fluxul de ieşire:<<

1.2 Directive de preprocesare [12, 63]

Un program este alcătuit dintr-un ansamblu de module de dimensiuni mici numite funcţii.

Funcţia este un set de instrucţiuni proiectată pentru a efectua o anumită sarcină. Funcţiile au o organizare ierarhică (arborescentă) având în rădăcina arborelui (vârful ierarhiei) funcţia **main**, numită modul principal.

Preprocesarea este prima etapă a compilării unui fişier sursă şi este realizată de un program special numit preprocesor, care este diferit de compilator. El este apelat automat de compilator pentru a executa anumite comenzi din fişierul sursă, numite *directive de preprocesare* şi care sunt precedate de #.

https://biblioteca-digitala.ro / https://unibuc.ro

Directiva #include

Inserează în programul sursă curent conținutul fișierului indicat de directivă. Cu aceasta putem modulariza scrierea programelor. Această directivă are trei sintaxe:

#include<fișier>
#include "fișier" (se folosește când fișierul antet se află în directorul curent)
#include nume, (**nume** este numele unei macro-comenzi).

Fișierele antet sunt fișiere text ASCII stocate în subdirectorul INCLUDE al directorului ce conține compilatorul C++.

Exemplu. Fișierul antet **iostream.h** oferă funcția de bibliotecă pentru operații de intrare/ieșire în flux. Orice program C++ începe cu una sau mai multe directive #include.

Directiva #define

Permite definirea constantelor și a pseudofuncțiilor numite și macrodefiniții cu parametri. Are sintaxele:

#define macro text
#define macro valoare
#define nume(listă_parametri) expresie
unde macro=identificatorul
text=secvență de caractere
valoare=valoare numerică
nume=numele pseudofuncției
expresie=expresia pseudofuncției

La preprocesare, orice apariție a identificatorului macro se înlocuiește cu definiția sa.

Exemplu.

#define EPS 0.001
#define PI 3.1415

Observație. Pe o linie de cod nu pot exista mai multe directive #define.

Macrodefinițiile cu parametri permit definirea pseudofuncțiilor care sunt mai rapide decât funcțiile obișnuite, dar ocupă mai multă memorie. O pseudofuncție acceptă ca argument un parametru și înlocuiește orice apariție în program a acelui parametru prin valoarea furnizată la apelul pseudofuncției.

Exemplu.

#define PATRAT(x) ((x)*(x))
#define CUB(x) (PATRAT(x)*(x))

https://biblioteca-digitala.ro / https://unibuc.ro

Aceste două pseudofuncţii calculează x^2 respectiv x^3 pentru x furnizat la apel.
Directiva #define se mai poate folosi pentru:

- înlocuirea cuvintelor rezervate sau a simbolurilor cu alţi identificatori definiţi de utilizator
- crearea de identificatori pentru tipuri de date definite de utilizator cu ajutorul tipurilor standard
- prescurtarea unor comenzi.

Directiva #undef
Este opusa directivei #define, permiţând anularea definirii curente a unui identificator, dacă aceasta nu mai este necesară. Are sintaxa:

#undef macro

Cu ea eliberăm spaţiul de memorie ocupat de macrodefinirea anulată şi putem reutiliza numele identificatorului acestuia într-o altă directivă #define.
Exemplu.

```
#undef EPS
#undef PATRAT
```

1.3 Operaţii de intrare/ieşire [12, 43, 80]

Se realizează prin funcţiile *printf* şi *scanf* din biblioteca standard ANSI.C specifică limbajelor TC, BC, C. C++ include şi biblioteca IOSTREAM care realizează operaţiile de intrare/ieşire printr-o clasa de obiecte.

1.3.1 Fluxuri de intrare/ieşire

Fluxul este o secvenţă de octeţi de la o sursă către o destinaţie. Este un model abstract pentru dispozitivele de intrare/ieşire precum tastatură, ecran, fişiere pe disc şi buffer de memorie. Biblioteca IOSTREAM conţine fluxul de intrare (clasa ISTREAM), de ieşire (clasa OSTREAM) şi intrare/ieşire (clasa IOSTREAM).
Fluxurile predefinite din bibliotecă sunt:

- *cin* – flux de intrare conectat de regulă la intrarea standard, care este tastatura
- *cout* – flux de ieşire conectat la ieşirea standard – ecranul .

https://biblioteca-digitala.ro / https://unibuc.ro

Pentru folosirea bibliotecii IOSTREAM trebuie inclus în programul C++ fişierul antet IOSTREAM.h, echivalent lui STDIO.h din ANSI.C.

Clasa IOSTREAM defineşte operatorii de intrare >> şi ieşire <<. Primul (>>) preia caracterele din bufferul asociat fluxului de intrare *cin* şi le converteşte în biţi, iar al doilea (<<) converteşte reprezentarea internă din memorie a variabilelor în caractere pe care le trimite bufferului asociat funcţiei de ieşire *cout*.

1.3.1.1 Fluxul de ieşire cout

Implicit acesta este conectat la unitatea standard de ieşire care este ecranul. Prin operatorii de redirectare putem orienta ieşirea către un fişier, imprimantă sau ca intrare pentru un alt program.

Fluxul *cout* permite afişarea atât a numerelor întregi sau în virgulă mobilă cât şi a şirurilor de caractere. Operaţia de ieşire << converteşte automat reprezentarea internă a unei variabile într-o reprezentare de tip text.

Mai mulţi operatori de ieşire pot fi concatenaţi într-o singură linie de program fiind trimişi către un acelaşi flux de ieşire.

Numărul sau caracterul care urmează operatorului de ieşire se ataşează celor aflate curent în fluxul de ieşire.

Exemplu.

```
#include<iostream.h>
void main()
{cout<<100;
cout<<2.475;
cout<<1<<0<<0;
cout<<"Numărul "<<100<<" este par";
cout<<"Numerele "<<25<<" şi "<<31.4<<" sunt reale";}
```

Pentru a afişa pe mai multe rânduri avem două posibilităţi

- plasăm caracterul de linie nouă \n în fluxul de ieşire liber în cadrul unui şir de caractere sau, inclus între două apostrofuri dacă programul nu afişează şir de caractere.
- folosim manipulatorul endl (sfârşit de linie)

Exemplu.

```
#include<iostream.h>
void main()
{cout<<"Toamna plouă \n Iarna ninge";
cout<<1<<'\n'<<2<<'\n'<<3<<'\n'<<4;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout<<"Mesaj"<<endl<<"Program C++! \n";
cout<<"Universitatea Bucureşti"<<endl;
cout<<"Academiei 14"<<endl;
cout<<"Bucureşti, România"<<endl;
cout<<"Universitatea Bucureşti"<<endl<<"Academiei
        14"<<endl<<"Bucureşti, România"<<endl;}
```

Pentru a afişa un text caracter cu caracter putem folosi funcţia *cout.put()*.

Fluxul cout foloseşte următoarele **caractere**:

\b – caracterul backspace
\f – avans la pagină nouă
\n – avans la linie nouă
\r – retur de car (fără avans de linie)
\t – tabulator orizontal
\v – tabulator vertical
\? – caracterul "?"
\' – caracterul " ' "
\" – caracterul " " "
\0 – caracterul nul
\ooo – valoare octală
\xhh – valoare hexazecimală

Exemplu.

```
#include<iostream.h>
void main()
```

{cout<<"\a Semnal!\a\t Semnal!";} – generează sunete în difuzorul calculatorului şi afişează "Semnal! Semnal!" separate de un tab.

1.3.1.2 Fluxul de intrare cin

Operatorul de intrare >> este folosit pentru preluare de date de la intrarea standard (tastatura) prin fluxul cin. Datele se introduc prin tastatură, după care se tastează <ENTER> pentru a confirma inserarea lor în flux.

Exemplu.

```
#include<iostream.h>
void main()
{double x,y,z,w;
cout<<"Introduceţi două numere \n";
cin>>x>>y;
z=x+y; w=x*y;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout<<x<<"+"<<y<<"="<<z<<'\n'
cout<<x<<"*"<<y<<"="<<w<<'\n';
```

Datele pot fi citite caracter cu caracter prin funcția *cin.get()*, iar şirul de caractere se poate citi de la tastatură prin *cin.getline()* care are formatul:

cin.getline(şir,lungime_şir);

Lungimea şirului este dată de operatorul *sizeof* din C++.

1.3.2 Manipulatori [47, 59, 12]

Sunt funcții care modifică starea unui flux prin plasarea lor în lanțul de operatori. Ei sunt declarați în fişierul antet <iostream.h> şi în <iomanip.h> pentru manipulatorii cu argumente.

Printre aceştia avem:

- dec→pentru intrare/ieşire – inițializează baza zecimală
- hex→pentru intrare/ieşire – inițializează baza hexazecimală
- oct→pentru intrare/ieşire – inițializează baza octală
- ws→pentru intrare – elimină spațiile albe
- endl→pentru ieşire – inserează linie nouă
- ends→pentru ieşire – inserează octet nul
- flush→pentru ieşire – descarcă bufferul fluxului ostream
- setbase(int)→pentru intrare/ieşire – inițializează baza de conversie (2,8,10,16)
- setfill(char)→pentru ieşire – stabileşte caracterul de completare a câmpurilor
- setiosflags(long)→pentru intrare/ieşire – inițializează parametrii de formatare specificați printr-un argument de tip long
- resetiosflags(long)→pentru intrare/ieşire – reinițializează parametrii de formatare
- setprecision(int)→pentru ieşire – stabileşte precizia în virgulă mobilă la un număr specificat de zecimale
- setw(int)→pentru intrare/ieşire – stabileşte lungimea totală a unui camp

Manipulatorul setw poate fi inlocuit cu metoda **cout.width** cu acelasi efect.

De asemenea manipulatorul setfill poate fi inlocuit cu metoda **cout.fill** .

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu.

```
#include<iostream.h>
#include<iomanip.h>
void main()
{int i=45;j=200;k=7;
cout<<"numărul"; cout.width(7); cout<<i<<endl;
cout<<"numărul"; cout.width(7); cout<<j<<endl;
cout<<"numărul"; cout.width(7); cout<<k<<endl;
cout.fill('.');
cout<<"numărul"<<setw(7)<<i<<endl;
cout<<"numărul"<<setw(7)<<j<<endl;
cout<<"numărul"<<setw(7)<<k<<endl;}
```

Programul afişează:

```
numărul 45
numărul 200
numărul 7
numărul.....45
numărul....200
numărul......7
```

Pentru valori în virgulă mobilă afişate de cout, numărul cifrelor de după virgulă este controlat cu modificatorul **setprecision**. Precizia stabilită este valabilă până la o nouă utilizare a acestui modificator.

Exemplu.

```
#include<iostream.h>
#include<iomanip.h>
void main()
{float x=100.12345;
cout<<setprecision(5)<<x<<endl;
cout<<setprecision(2)<<x<<endl;
cout<<setfill('0')<<setw(8)<<setprecision(2)<<x<<endl;
cout<<x<<endl;
}
```

Programul afişează:

```
100.12345
100.12
00100.12
```

Manipulatorul **setiosflags** permite specificarea unui anumit format pentru afişare. Pentru a reveni la formatul anterior, folosim **resetiosflags**. Aceşti

https://biblioteca-digitala.ro / https://unibuc.ro

manipulatori au argumente care conţin: numele clasei **ios** în care sunt definite, urmat de operatorul de rezoluţie globală :: şi de numele parametrilor care poate fi:

- skipws: elimină spaţiile albe la intrare
- left: aliniere la stânga în câmpul de ieşire
- right: aliniere la dreapta în câmpul de ieşire
- scientific: foloseşte notaţia ştiintifică pentru numere reale în virgulă mobilă)exemplu -2.45e+03
- fixed: foloseşte notaţia zecimală (-31.67)
- dec: foloseşte notaţia zecimală pentru întregi
- hex: foloseşte notaţia hexazecimală pentru întregi
- oct: foloseşte notaţia octală pentru întregi
- uppercase: foloseşte litere mari la ieşire: (exemplu -2.45E+03)
- showbase: indică baza sistemului de numeraţie la ieşire:
  - 0x pentru hexazecimal, ca prefix
  - 0 pentru octal, ca prefix
- showpoint: introduce un punct la ieşirea în virgulă mobilă (exemplu .368)
- showpos: introduce "+" la afişarea valorii pozitive
- unitbuf: pentru golirea tuturor fluxurilor după inserarea caracterelor într-un flux

Exemplu. [12]

```
#include<iomanip.h>
#include<iostream.h>
#include<stdio.h>
void main( )
{int i=300;
cout<<setfill('.');
cout<<setiosflags(ios::left);
cout<<setw(15)<<"zecimal"
cout<<setiosflags(ios::left);
cout<<setiosflags(ios::showpos);
cout<<setw(4)<<dec<<i<<endl;
cout<<setiosflags(ios::left);
cout<<setw(15)<<"Hexazecimal"
cout<<setiosflags(ios::left);
cout<<setiosflags(ios::uppercase);
cout<<setw(4)<<hex<<i<<endl;}
```

https://biblioteca-digitala.ro / https://unibuc.ro

Afişează la ecran:

zecimal.....+300
Hexazecimal.....12C

1.4 Variabile şi constante [12, 47, 63]

Variabila este un obiect al limbajului dat prin tip, adresă şi valoare.

Tipurile predefinite în C++ sunt: int, long, float, double, char.

Adresa este un număr care precizează zona de memorie ocupată de valoarea variabilei. Caracterul & pus în faţa variabilei semnifică adresa variabilei respective.

Valoarea este furnizată de programator şi este un atribut modificabil. Astfel unei variabile îi corespund două valori:

- adresa zonei de memorie rezervată variabilei
- valoarea memorată în zona rezervată.

Pentru a putea folosi în program o variabilă mai întâi aceasta trebuie declarată. Compilatorul va rezerva spaţiu de memorie şi o va reţine într-o poziţie fixă. Se declară prin:

tip nume; sau **tip nume=valoare_iniţială**;

Numele este unic şi este o combinaţie de litere , cifre şi "_" ce nu poate începe cu o cifră (se disting literele mari de cele mici).

După tip pot apărea mai multe nume de variabile separate prin ",".

Unei variabile declarate i se poate atribui o valoare care poate fi o constantă sau o expresie ce are ca operanzi variabile anterior definite. Această valoare poate fi folosită în program prin simpla specificare a numelui variabilei.

Exemplu.

```
int i,j=1,k=5;
float a=4.58
float x=7.2, y=3*a; z=2*a;
```

Dacă unei variabile i se atribuie o valoare din afara intervalului corespunzător tipului variabilei, se produce o eroare de depăşire a capacităţii de stocare a variabilei respective .

Exemplu. int a=35000; In acest caz a primeşte o valoare arbitrară – 30536, deoarece valoarea maximă admisă este 32768 pentru tipul int.

Atributul valoare al unei variabile poate conţine nu numai numere, ci şi adrese, ca în cazul pointerilor.

https://biblioteca-digitala.ro / https://unibuc.ro

Variabilele pot primi valori şi prin citire cu operatorul de intrare >> aplicat dispozitivului standard de intrare (tastatura) prin fluxul cin.

O altă caracteristică a variabilelor o constituie *durata de viaţă*, care reprezintă timpul cât variabilei respective îi este alocat spaţiul de memorie şi îi este determinată *clasa de memorie* asociată. Clasele de memorie posibile sunt:

- auto – datele se află pe stivă
- register – datele se află într-un registru al calculatorului
- static – datele sunt rezidente în modul de definiţie
- extern – datele sunt definite într-un modul extern

Implicit, variabilele aparţin clasei de memorare auto.
Există trei posibilităţi de alocare a memoriei:

- variabilele statice, plasate în segmentul de dată al programului, au o durată de viaţă egală cu timpul de execuţie al programului,
- variabilele depuse pe stiva modulului de definiţie sau într-un registru al calculatorului, au durata de viaţă egală cu cea a apelului funcţiei care le defineşte, memoria fiind alocată automat,
- variabilele din memoria *heap* (rezervată obiectelor dinamice), au durata de viaţă controlată de program, memoria fiind alocată dinamic.

Constantele sunt indentificatori de un tip anume care au o valoare fixată. Pot fi de tip: întreg, real, caracter sau şir de caractere.

Constanta de tip caracter este compusă dintr-un caracter cuprins între apostrofuri. Constantele se folosesc pentru definirea altor constante sau pentru dimensionarea tablourilor.

Prezenţa modificatorului *const* în sintaxa declarării unei variabile face ca valoarea acesteia să nu se poată modifica în timpul rulării programului şi astfel variabila devine o constantă.

Constantele se declară prin: **const tip nume = valoare;**

Dacă *tip* lipseşte, compilatorul atribuie constantei automat tipul *int*.

Constantele se definesc şi prin directiva de preprocesare **#define** care este moştenită de la limbajul C. Se definesc la începutul programului *nume de constante simbolice* la care se asociază *şiruri de caractere* particulare. Fiecare apariţie în program a numelui unei constante simbolice se înlocuie de către compilator cu şirul de caractere asociat. Numele simbolice se scriu cu majuscule şi definiţia constantei simbolice nu se încheie cu ";".

https://biblioteca-digitala.ro / https://unibuc.ro

Se recomandă definirea constantelor prin *modificatorul const,* care precizează şi tipul constantei şi mai puţin prin constante simbolice.

Exemplu.

```
#include<iostream.h>
#define MAX 2
const int x=5;
const char y='$';
cout<<"Constantele sunt:"<<endl;
cout<<"x="<<x<<"şi"<<"y="<<y<<endl;
cout<<"Sunt afisate"<<MAX<<"valori"<<endl;}
```

După compilare şi rulare vor fi afişate următoarele:

```
Constantele sunt:
x=5 şi y=$
```

Sunt afisate 2 valori.

1.5 Tipuri şi structuri de date [43, 59, 47]

1.5.1 Tipuri de date fundamentale

Tipurile numerice fundamentale în C++ sunt *int* şi *long* pentru întregi şi *float* şi *double* pentru reale în virgulă mobilă.

Tipurile *int* şi *long* au diverse variante prin folosirea prefixelor short, signed, unsigned.

Variantele acestor tipuri, spaţiul de memorie ocupat şi domeniul de valori sunt:

Variante	Spaţiul de memorie ocupat (în biţi)	Domeniul de valori
Short, short int	16	-32768÷32767
Signed short	16	-32768÷32767
Unsigned short	16	0÷65535
Unsigned short int	16	0÷65535
Int, signed int	16	-32768÷32767
Unsigned int	16	0÷65535
Long, long int	32	$-2147483648 \div 2^{31}-1$
Signed long	32	$-2147483648 \div 2^{31}-1$
Signed long int	32	$-2147483648 \div 2^{31}-1$

https://biblioteca-digitala.ro / https://unibuc.ro

Unsigned long	32	0÷4294567195
Unsigned long int	32	0÷4294567195
Float	32	$[3.4 \cdot 10^{-38}, 3.4 \cdot 10^{38}]$
Double	64	$1.7 \cdot 10^{-308}, 1.7 \cdot 10^{308}]$
Lond double	80	$3.4 \cdot 10^{-4932}, 1.1 \cdot 10^{4932}]$

Limbajul C++ face conversii între tipurile de date atât implicit de calculator cât şi explicit.

Dacă o expresie conţine operanzi de tipuri diferite, înaintea executării operaţiilor prevăzute în expresie, *tipul inferior* este convertit la *tipul superior.*

Pentru orice *operator aritmetic* regulile de conversie sunt:

- *short* se converteşte la *int*, iar *float* la *double*
- dacă un operand este *double*, atunci şi al doilea este convertit la double şi rezultatul este double
- dacă un operand este *long*, atunci şi al doilea este convertit la long şi rezultatul este long
- dacă un operator este *unsigned* atunci şi al doilea este convertit la unsigned şi rezultatul este unsigned.

Conversia implicită se poate face şi prin *operaţia de atribuire.*

În expresia x=y, valoarea lui y este automat convertită la tipul lui x care este şi tipul rezultatului. Conversia de la *int* la *float* nu modifică valoarea afişată, ci doar reprezentarea internă a acesteia.

Conversia de la *float* la *int* trunchează partea fracţionară.

Conversia de la *double* la *float* se face prin rotunjire.

Conversia de la *long* la *short* duce la pierderea biţilor de ordin superior care sunt în plus.

Conversia explicită se face cu ajutorul unor funcţii speciale.

Conversia de la *float* la *int* se face cu funcţia **(int)** care are sintaxa:

(int)var, unde var=variabilă reală.

Conversia de la *int* la *float* se face cu funcţia **(float)** care are sintaxa:

(float)var, unde var=variabilă întreagă.

Exemplu.

```
#include<iostream.h>
void main( )
{float x=15.34, y=7.6;
int i=15, j=-24;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout<<(int)x<<endl;
cout<<(int)y<<endl;
cout<<"Valorile înainte de conversie:"<<endl;
cout<<i<<endl<<j<<endl;
cout<<"Valorile după conversie:"<<endl;
cout<<(float)i<<endl;
cout<<(float)j<<endl;}
```

După execuţie se afişează:

```
15
-7
Valorile înainte de conversie:
15
-24
Valorile după conversie:
15
-24
```

Se observă că în conversia de la *int* la *float* s-a produs doar modificarea reprezentării interne a valorilor lui i şi j şi nu s-au modificat valorile afişate 15 şi –24.

Conversia se face şi la transmiterea argumentelor către o funcţie. Prin aceasta, *short* devine *int*, iar *float* devine *double*. Astfel, argumentul unei funcţii poate fi declarat *int* sau *double*, chiar dacă funcţia este apelată cu *short* sau *float*.

Conversia explicită se poate face şi prin construcţia *typecast* care are sintaxa:

```
(tip)expresie;
tip(expresie);
```

Exemplu.

```
int x=3;
double y,z;
y=double(x);
z=(double)x;
```

Aceasta are ca efect conversia tipului int al lui x într-un număr double stocat în variabilele y şi z.

Tipul caracter (char)

Se referă la date α-numerice. Variantele tipului char, spaţiul de memorie ocupat şi domeniul de valori sunt:

https://biblioteca-digitala.ro / https://unibuc.ro

Varianta	Spațiul de memorie ocupat	Domeniul de valori
Signed char	8 biți	-128÷127
Unsigned char	8 biți	0÷255
Char	8 biți	-128÷127

Unele caractere au corespondent grafic.

Variabilele de tip char pot primi valori prin atribuire astfel:

var='c'; c=caracter grafic (litere, cifre, operatori aritmetici, semne de punctuație)

var='xXY, X,Y=cifre în baza 16

var=0xXY, unde X, Y sunt cifre sexazecimale

var='abc'; abc=număr în baza 8

Exemplu. v_1='\x74'; v_2='t'. Aici, v_1 şi v_2 au ca valoare caracterul t, deoarece

$74_{16}=116_{10}\rightarrow$t.

Între datele de tip *int* şi *char* acționează conversia implicită: *unsigned char* este convertit la tipul *int* ca un întreg pozitiv reprezentat pe un octet; *signed char* este convertit la *int* ca un întreg pozitiv sau negativ reprezentat pe un octet în care primul bit este rezervat pentru semn.

Astfel, ordinea între caractere este dată de ordinea numerelor corespunzătoare. Conversia *char-int-char* nu duce la pierderea de informații.

Exemplu.

```
char a,b; int i;
a='c'; i=a; b=i;
cout<<"i are valoarea:"<<i<<endl;
cout<<"b are valoarea:"<<b<<endl;
```

Se va afişa:

i are valoarea:99

b are valoarea:c.

Datorită diferenței de lungime dintre zonele ocupate de tipurile char (8 biți) şi int (16 biți), la conversia int→char se pierd informații. Conversia întreg fără semn→char→întreg are ca rezultat restul împărțirii numărului inițial la 256.

Exemplu.

```
int i,j; char x;
i=800; x=i; j=x;
cout<<"j are valoarea"<<j<<endl;
```

https://biblioteca-digitala.ro / https://unibuc.ro

Se va afişa:
j are valoarea 32.

1.5.2 Tablouri [12, 47, 43, 80]

Tablourile sunt tipuri de date care reţin mai multe valori de acelaşi tip. Ele pot fi unidimensionale (vectori), bidimensionale (matrice), tridimensionale, etc.

1.5.2.1 Vectori

Vectorul este un obiect caracterizat de tip, adresă, valoare şi care conţine unul sau mai multe elemente de acelaşi tip.

Valoarea unui vector coincide cu adresa primului element al său.

Tipul vectorului este tipul comun al elementelor sale care poate fi: int, char şi float.

Elementele unui vector se memorează unul după altul. Accesul la elemente se face printr-o valoare *index* care indică elementul accesat şi care se include între paranteze drepte. Adresa primului element v[0] coincide cu adresa vectorului.

Orice vector are un nume unic şi o dimensiune care dă numărul elementelor sale.

Se declară prin:

tip nume[dim];

După declarare, compilatorul alocă memoria necesară păstrării numărului de elemente indicat prin *dim*.

Primul element al vectorului este nume[0], iar ultimul nume[dim-1].

Exemplu.

```
int i; int vect[4];
vect[0]=10; vect[1]=20; vect[2]=30; vect[3]=40;
cout<<"Vectorul vect are elementele:"<<endl;
for (i=0;i<=3;i++) {cout<<vect[i]<< " ";}
```

După compilare şi execuţie se afişează:

```
Vectorul vect are elemetele:
10 20 30 40.
```

La declarare un vector se poate şi iniţializa sub forma:

tip nume[dim]={exp_1,exp_2,...,exp_n}; unde n=dim.

Exemplu.

```
int vect[4]={10, 20, 30, 40}; sau
int vect[ ]= {10, 20, 30, 40};
```

https://biblioteca-digitala.ro / https://unibuc.ro

Dacă la declararea cu iniţializare nu se specifică dimensiunea, compilatorul alocă suficient spaţiu pentru memorarea numărului de elemente care corespund valorilor de iniţializare.

Vectorii de tip char reţin şiruri de caractere prin atribuirea de caractere elementelor componente. C++ foloseşte caracterul NULL reprezentat de caracterul special '\0' şi având codul ASCII 0 pentru a marca ultimul caracter al unui şir de caractere.

Pentru a declara un şir de caractere, se declară un vector de tip char cu o dimensiune suficient de mare pentru a reţine caracterele şirului. Ultimul element al vectorului este caracterul NULL.

Constantele şir de caractere, plasate între ghilimele au ataşat în mod automat de compilator caracterul NULL.

La declararea vectorului de tip char este posibilă şi iniţializarea şirului de caractere prin precizarea caracterelor incluse între ghilimele.

Exemplu.

char vect[40]="Acesta este un exemplu de şir";

1.5.2.2 Matrice

Matricea este un obiect caracterizat prin: tip, adresă şi valoare care coincide cu adresa primului său element.

Matricea conţine unul sau mai multe elemente de acelaşi tip, care este şi tipul matricei.

Matricea are un nume unic şi două dimensiuni care dau numărul liniilor şi numărul coloanelor. Se memorează pe linii.

Matricea se consideră ca fiind tablou de tablouri şi deci se declară prin:

tip nume[dim1][dim2]; dim1=numărul liniilor; dim2=numărul coloanelor

Numerotarea elementelor începe cu 0. Astfel, nume[0] indică prima linie, nume[1] a doua linie, etc. Adresa primei linii coincide cu adresa matricei. Ca şi la vectori, matricele pot fi iniţializate la declarare prin:

tip nume[dim1][dim2]={exp_{11},...,exp_{1dim2},...,exp_{dim11}, ...,$exp_{dim1dim2}$}.

Valorile iniţiale se introduc pe linii.

Exemplu.

```
#include<iostream.h>
void main( )
{int i,j;
int mat[3][3]={1, 2, 3, 4, 5, 6, 7, 8, 9};
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout<<"Matricea mat conține elementele:"<<endl<<endl;
for (i=0;i<3;i++){
  cout<< " ";
for (j=0;j<3;j++)
  cout<<mat[i][j]<<" ";
cout<<endl;
}}
```

Se afişează:

Matricea mat conține elemetele:
1 2 3
4 5 6
7 8 9

1.5.3 Definirea tipurilor de date [12, 47, 59]

Putem crea noi tipuri de date în C++ prin *typedef* urmat de o declarație de tip şi de un identificator sub forma:

typedef tip identificator;

unde tip este un tip de date existent, iar identificator este numele noului tip definit aici.

Exemplu.

```
typedef struct coor
        {float x;
        float y;};          →declară tipul coor
```

Astfel, typedef permite:

- crearea de noi tipuri de date
- acordarea de nume explicative tipurilor folosite in program
- crearea unor nume care prescurtează numele tipurilor existente
- definirea numelui unui tip de tablou prin:

typedef tip nume_tip [dim];

Tipul enumerare

Acest tip defineşte o listă de identificatori cărora li se asociază valori întregi.

Tipul enumerare se declară prin:

enum tip{id_1,id_2...};

Compilatorul atribuie 0 lui id_1, 1 lui id_2, etc/

Identificatorii din listă pot fi iniţializaţi cu întregi în mod explicit, iar valorile lor pot să şi coincidă.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplul 1.

enum valori{adevărat,fals,indiferent}
enum zile{luni,marți,miercuri,joi,vineri}
enum calif{FB,B,S,NS}
Astfel, FB=0,B=1,S=2,NS=3.

Exemplul 2.

enum numerotare{Mihai=3,Andrei=12,Vlad=7};

Exemplul 3.

enum anotimp{iarna,primăvara=1,vara,toamna}←se atribuie valori unui singur membru

În acest exemplu, compilatorul atribuie celorlalți membri valorile 2,3,respectiv 0 pentru iarna, în funcție de ordinea lor în listă.

Se pot defini tipuri *enumerare anonime* (fără nume), care trebuie să includă şi declararea variabilelor de tip enumerare anonimă sub forma:

enum{iarna,primăvara,vara,toamna}anotimp;
enum{nord,vest,sud,est}puncte_cardinale;
Variabilele de tip enumerare se declară:

- prin includerea declarării lor în declararea tipului enumerare
- prin neincluderea declarării lor în declararea tipului enumerare

adică sub formele echivalente:

enum tipen{id_1,id_2,...}var_1,var_2,...;
enum tipen{id_1,id_2,...};
tipen var_1,var_2,...;

Exemple.

enum sporturi{fotbal,baschet,handbal,volei}tenis,polo; sau
enum sporturi{fotbal,baschet,handbal,volei};
sporturi tenis,polo;

1.5.4 Structuri

Structurile sunt colecții de date, eventual de tipuri diferite, care pot fi referite atât separat la nivelul membrilor colecției, cât şi împreună la nivel de grup. Membrii colectiei pot fi tipuri predefinite sau alte structuri. Se declară prin:

struct nume{
tip_1 $membru_1$;
tip_2 $membru_2$;

https://biblioteca-digitala.ro / https://unibuc.ro

```
};
```

Dacă declararea nu este urmată de o listă de variabile, ea nu produce alocare de memorie ci descrie doar organizarea structurii.

Exemplu.

```
struct persoana {
        char nume[10];
        char pren[10];
        char prof[10];
        unsigned vârsta;
        char localitatea[15];
        char adresa[40];};
persoana elev={"Popescu","Ion","Arhitect",40,"Bucureşti","Ion Manolescu 2"};
```

Variabilele de tip structură se declară prin:

- includerea declarării variabilelor la definirea structurii
- neincluderea declarării variabilelor la definirea structurii (ca în exemplul de mai sus).

Exemplu.

```
struct complex {
        float modul;
        float argument;}n1,n2;
```

Variabilele n1 şi n2 sunt de tip complex, având fiecare două componente: modul şi argument.

Se pot declara şi structuri anonime, care trebuie să includă şi declararea variabilelor de tip structură anonimă.

Exemplu.

```
struct{
        float modul;
        float argument;}n1,n2;
```

Variabilele de tip structură se pot iniţializa chiar la declararea loar ca în exemplul de mai sus pentru variabila elev.

Atribuirea de valori unui membru al unei structuri sau accesarea unui membru dintr-o structură se face cu operatorul "." numit *operator de apartenenţă* sub forma:

nume-variabila-structura • membru;

Exemplu.

Pentru structurile de mai sus avem:

```
elev.vârsta=55;
```

https://biblioteca-digitala.ro / https://unibuc.ro

n1.modul=8.3;
n2.argument=1.57;

1.5.5 Uniuni

Uniunile sunt structuri speciale, în care membrii, nu neapărat de acelaşi tip, folosesc în comun aceeaşi zonă de memorie. Lungimea unei uniuni este dimensiunea maximă a componentelor sale.

Spre deosebire de structuri, o singură componentă a uniunii este accesibilă la un moment dat.

Se declară prin:

```
union nume{
        tip1 nume1;
        tip2 nume2;
        .....};
```

Variabilele se declară ca şi la structuri în cele două moduri precizate(definim variabilele la declararea structurii imediat după acolada de închidere sau separat).

Când se atribuie valori unui mambru dintr-o uniune, are loc o suprascriere peste orice atribuire anterioară (fiind vorba de aceeaşi zonă de memorie).

Accesarea membrilor unei uniuni se face tot cu *operatorul apartenenţă* "."

Exemplu.

```
union valoare{
      int a; long b;}var;
var.a=700;
var.b=100000
```

Se pot declara *uniuni anonime*. In acest caz referirea se face direct prin numele variabilei, fără operatorul "." .

Exemplu.

```
union {
        int a; long b;}var;
a=700;
b=100000;
```

https://biblioteca-digitala.ro / https://unibuc.ro

1.6 Operatori [12, 47, 43, 63]

Prin *expresie* înțelegem o combinație logică între operanzi (date) şi operatori care pot fi: unari, binari şi ternari.
Operatorii pot fi:

- aritmetici
- de incrementare şi decrementare
- de atribuire
- relaționali şi logici
- la nivel de bit
- sizeof,virgulă,condițional
- operatori specifici limbajului C++
    - o de alocare a memoriei
    - o de eliberare a memoriei
    - o de rezoluție

Mai mulți operatori într-o expresie se evaluează de la stânga la dreapta sau de la dreapta la stânga, conform ordinului de prioritate stabilit în funcție de nivelul de prioritate al operatorilor.
Operatorii aritmetici sunt:

- semn +,- (unari)
- +,-,*,/ (binari)
- %(modulo)←dă restul împățirii a doi operanzi întregi

Cu aceştia se construiesc expresii artimetice care pot conține şi apeluri de funcții. În acest caz evaluarea funcțiilor are prioritate față de celelalte operații ale expresiei.
Operatorii de incrementare şi decrementare sunt: ++ respectiv --
Se aplică doar variabilelor şi le măresc valoarea cu 1, respectiv le scad valoarea cu 1.

Au sintaxa: var++; respectiv var--;
++var; --var;

Se observă că ei pot fi prefixați când modifică valoarea variabilei înainte de a fi folosită într-o expresie sau postfixați când modifică valoarea după folosirea variabilei în expresie.

Dacă operatorul ++, respectiv - - este singur într-o expresie, atunci ambele forme (prefixate sau postfixate) au acelaşi efect.
Operatorul de atribuire este =.

Operatorul "=" separă doi operanzi cu care formează expresia de atribuire:

https://biblioteca-digitala.ro / https://unibuc.ro

var=expresie;

Aici var primeşte valoarea expresiei din dreapta. Expresia de atribuire **var=expresie** are ca tip tipul variabilei din stânga (tipul lui var).

C++ pune la dispoziţie şi operatori aritmetici de atribuire care combină operatorii aritmetici cu cel de atribuire şi permit scrierea prescurtată a unor expresii aritmetice. Această formă scurtă a operatorilor aritmetici de atribuire permit ca în anumite instrucţini de atribuire numele unei variabile să apară o singură dată. Operatorii aritmetici de atribuire sunt +=,-=,*=,/=,%=.

Exemplu. In loc de x=x%y scriem x%=y; lungime=lungime+5 devine lungime+=5;

Operatorii logici şi relaţionali sunt:

- &&, ||, ! – operatori logici (şi, sau, non)
- <,<=,>,>=,==,!= - operatori relaţionali

Cu operatorii logici se construiesc expresii logice.

Cu operatorii relaţionali se construiesc expresii relaţionale, care se evaluează de la dreapta la stânga. Rezultatul evaluării este 1 dacă expresia este adevărată şi 0 dacă este falsă.

C++ nu dispune de un tip boolean predefinit. El consideră valoarea "0" ca "fals" şi orice valoare diferită de "0" ca "adevărat".

Expresiile logice se evaluează de la stânga la dreapta şi evaluarea se opreşte atunci când se cunoaşte valoarea de adevăr a rezultatului.

Expresiile mixte sunt cele care conţin atât operatori relaţionali, cât şi logici. Operatorii relaţionali au prioritate faţă de cei logici. Dacă expresia conţine şi operatori aritmetici aceştia au prioritate la execuţie.

Operatorii la nivel de bit se folosesc pentru operatii de manipulare a biţilor precum: comutare, iniţializare, filtrare şi deplasare.

Aceştia sunt: &,|,~ (şi, sau, non bit cu bit), ^ (sau exclusiv bit cu bit), <<,>> deplasare stânga, respectiv deplasare dreapta bit cu bit.

Exemplu.

x=x|ANT pune în x pe 1 biţii care sunt 1 în ANT

^ sau exclusiv operează astfel:

	1	0
1	0	1
0	1	0

https://biblioteca-digitala.ro / https://unibuc.ro

Operatorii << şi >> deplasează la stânga, respectiv la dreapta biţii reprezentând valoarea binară a operatorului din stânga cu un număr de poziţii dat de operandul din dreapta.

a<<3 deplasează la stânga pe a cu trei poziţii iar biţii vacanţi îi umple cu 0.

C++ are şi operatori de atribuire la nivel de bit, care dau forme prescurtate pentru instrucţiuni simple la nivel de bit. Aceştia sunt: &=, |=, ^=, <<=, >>=.

Exemplu. x=x>>y se scrie x>>=y, etc.

Operatorul sizeof

Acest operator întoarce lungimea în octeţi a memoriei operandului, care poate fi variabilă (scalar, tablou, înregistrare, etc.) sau un tip de date, pe baza declarării elementelor acestuia. Are formatul: **sizeof(variabilă);** sau **sizeof(tip_dată);**

Exemplu.

cout<<"Lungime tip long double="<<sizeof(long double)<<endl;

Afişează: Lungime tip long double=10

Operatorul virgulă

Are sintaxa: **$expr_1$,$expr_2$,...,$expr_n$;**

Secvenţa din sintaxă este ea însăşi o expresie, în care primele n-1 expresii trebuie să fie atribuiri, incrementări sau decrementări, iar ultima poate să fie orice. Se evaluează de la stânga la dreapta. Rezultatul şi tipul acestei expresii (secvenţă de expresii) sunt date de valoarea şi tipul ultimei expresii.

Exemplu.

int x=13,y=8,z,t;

t=(z=x-5,x=y+4,y=7-x,x/3).

La sfârşit t are valoarea 4.

Operatorul condiţional ?: este unicul ternar în C++ şi are sintaxa:

$expr_1$?$expr_2$:$expr_3$;

Operează astfel: dacă $expr_1$ atunci $expr_2$ altfel $expr_3$.

Se evaluează $expr_1$ şi dacă aceasta este diferită de 0, operatorul întoarce valoarea lui $expr_2$ altfel, întoarce valoarea lui $expr_3$.

Operatorul de alocare a memoriei: new, are sintaxa

pointer=new nume[val_init];

Acesta creează obiectul **nume** prin alocarea unui număr de octeţi egal cu sizeof(nume) în zona specială din memoria liberă numita *heap* şi apoi

https://biblioteca-digitala.ro / https://unibuc.ro

întoarce adresa obiectului sau valoarea NULL dacă alocarea nu a avut succes. Obiectul creat se poate initialize cu val_init.

Operatorul de eliberare a memoriei: **delete**, eliberează o zonă de memorie alocată cu new când aceasta nu mai este necesară. Are sintaxa: **delete pointer**.

Eliberarea zonei de memorie începe de la adresa conținută de pointer.

Operatorul de rezoluție (scop) **::** este folosit pentru definirea funcțiilor membre unei clase pentru a preciza clasa cu care este asociată o anumită funcție. Dacă operatorul nu are nume de clasă, el indică o funcție sau o variabilă definită global.

Reguli de prioritate a operatorilor [12]

Nivelurile de prioritate ale operatorilor C++ şi ordinea de evaluare(←=dreapta-stânga, →=stânga-dreapta) când operatori consecutivi au aceeaşi prioritate sunt:

- rezoluție :: 1 →
- postincrementare ++ 2←
- postdecrementare - - 2←
- preincrementare ++ 2←
- predecrementare - - 2←
- operator adresă & 2←
- redirectare * 2←
- plus + 2 ←
- minus – 2 ←
- negație pe biți ~ 2 ←
- negație logică ! 2 ←
- sizeof 2 ←
- conversia tiptype 2 ←
- înmulțire * 3 →
- împărțire / 3 →
- modulo % 3 →
- adunare + 4 →
- scădere – 4 →
- deplasare stânga << 5 →
- deplasare dreapta >> 5 →
- mai mare strict > 6 →
- mai mare sau egal >= 6 →

https://biblioteca-digitala.ro / https://unibuc.ro

- mai mic strict < 6 →
- mai mic sau egal <= 6 →
- egal == 7→
- diferit != 7 →
- and pe biți & 8 →
- xor pe biți ^ 9 →
- or pe biți | 10 →
- and logic && 11 →
- or logic || 12 →
- condițional ?: 13 →
- atribuire = 14 ←
- atribuire înmulțire *= 14 ←
- atribuire împărțire /= 14 ←
- atribuire rest %= 14 ←
- atribuire adunare += 14 ←
- atribuire scădere -= 14 ←
- atribuire deplasare stânga <<= 14←
- atribuire deplasare dreapta >>=14←
- atribuire and pe biți &= 14 ←
- atribuire xor pe biți ^=14 ←
- atribuire or pe biți |= 14 ←
- virgulă , 15 →

1.7 Instructiuni [47, 59, 63]

Instrucțiunile descriu acțiunile pe care le relizează funcțiile unui program. O instrucțiune poate conține cuvinte cheie, expresii şi alte instrucțiuni. Se încheie cu ";" .
O linie de program poate conține mai multe instrucțiuni şi o instrucțiune se poate scrie pe mai multe linii program, spațiile nesemnificative fiind ignorate. O instrucțiune poate fi **simplă** (conține o singură operație) sau **compusă** (mai multe instrucțiuni incluse între acolade).
Instrucțiunile limbajului C++ sunt :

- instrucțiunea expresie
- instrucțiuni de decizie
    - if cu o alternativă

https://biblioteca-digitala.ro / https://unibuc.ro

- if _ else cu două alternative
- if _ else cu n ≥ 3 alternative
- instrucțiunea switch

- instrucțiuni de ciclare
  - instrucțiunea for
  - instrucțiunea while
  - instrucțiunea do _ while
- instrucțiuni de salt
  - instrucțiunea break
  - instrucțiunea continue
  - instrucțiunea goto
  - instrucțiunea return
- instrucțiunea compusă

Instrucțiunea expresie are sintaxa: **expresie;**
Aceasta atribuie valori variabilelor, incrementează/decrementează variabile, etc.
Exemplu. Apelarea unei funcții urmată de ; este o instrucțiune expresie.
Un caz particular de instrucțiune expresie este instrucțiunea vidă compusă din ; (aceasta evită plasarea unei etichete în fața acoladei de închidere a unui bloc) .
Un alt caz particular este instrucțiunea **expresie condițională** cu sintaxa :

$expr_1$? $expr_2$: $expr_3$;

Dacă $expr_1$ este adevărată (≠0) se evaluează $expr_2$ şi expresia condițională ia valoarea lui $expr_2$ iar dacă $expr_1$ este falsă (= 0) se evaluează $expr_3$ şi expresia condițională ia valoarea lui $expr_3$ (aceasta poate substitui instrucțiunea de decizie if_else cu două alternative).
Exemplu .

```
#include <iostream.h>
void main()
{
        double a, b, max;
        cout <<"introduceti numerele a şi b : "<<endl;
        cin >>a >> b ;
        max = (a>b) ? a : b ;
        cout << "Maximul dintre " << a << " şi " <<b<<"
este " <<max<<endl ;
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

Instrucţiunea if cu o alternativă are sintaxa :

if (condiţie) instr ;|{secvenţa de instrucţiuni}

unde **condiţie** este orice expresie corectă.

Se evaluează condiţie şi dacă este adevărată($\neq 0$) se execută instrucţiunea respectiv secvenţa de instrucţiuni, din sintaxa lui if.

Exemplu .

```
#include <iostream.h>
#include <conio.h>
 void main ()
{
int k;
clrscr();
cout << "Tasteaza o cifra " << endl;
cin >> k;
if (k>0) {
        cout << "Ati tastat nr. " << k << "." << endl;
        cout << "Sfarsit." << endl;
          }
}
```

Instrucţiunea if cu două alternative are sintaxa :

if (condiţie) instr$_1$; |{secvenţa$_1$ de instrucţiuni}

else instr$_2$; |{secvenţa$_2$ de instrucţiuni}

Se evaluează **condiţie** şi dacă aceasta este adevărată ($\neq 0$) se execută instr1 respectiv secvenţa$_1$ de instrucţiuni, iar dacă este falsă (se obţine o valoare = 0) se execută instr$_2$ respectiv secvenţa$_2$ de instrucţiuni .

Dacă nu putem reduce condiţia la o singură expresie(există mai multe ramuri de decizie) folosim instrucţiuni if_else imbricate :

```
 if (cond_1)
        if (cond_2) instr_1;
         else instr_2 ;
```

Aici if_else cu o singură alternativă include o instrucţiune if cu două alternative , deoarece Compilatorul asociază **else** cu ultima instrucţiune **if** întâlnită (aici cu instrucţiunea if imbricată).

Un alt mod de asociere se obţine folosind delimitatorii de bloc (acoladele) pentru instrucţiunea if imbricată cum ar fi în cazul :

```
if (cond_1)
{
        if (cond_2) instr_1;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
}
 else instr_2 ;
```

Exemplu .

```
#include <iostream.h>
#include <conio.h>
void main ()
     {
        int i;
        clrscr();
        cout << "Introduceti un nr natural  "<<endl ;
        cin >> i;
        if (( i % 2 !=  0 ) || ( i % 3 != 0))
                        cout << "Nr. " << i << " nu se divide cu 6 " ;
        else
                        cout << "Nr." << i << "se divide cu 6" ;
        }
```

Instrucțiunea if cu mai multe alternative este generată de două sau mai multe instrusțiuni if_else în cascadă imbricate. Are sintaxa :

```
if (cond1)
instr1; |{ secvența1 de instrucțiuni}
else if (cond2)
        instr2 ; | {secvența2 de instr}
...
else if (condn)
        instrn ;|{ secvențan de instr.}
[else
        instrn+1;|{ secventan+1 de instr.}  ]
```

Testarea în cascadă a condițiilor continuă până când una din condiții este adevarată şi se execută instrucțiunea sau secvența de instrucțiuni corespunzătoare, sau se execută instrucțiunea de după **else** dacă aceasta există, sau se iese, în caz contrar .

Pentru a spori lizibilitatea programului se recomandă ca instrucțiunile ce urmează un if , un else sau { să fie indentate.

Instrucțiunea switch

Aceasta permite selectarea şi execuția unei singure alternative din mai multe posibile în funcție de diversele valori ale unei expresii compatibile cu un întreg . Ea substituie intrucțiuni if-else în cascadă prea lungi .

https://biblioteca-digitala.ro / https://unibuc.ro

Are sintaxa : switch (expr)

```
{
    case cst1 :
        instr1 ;|set1 de instr;
        break;
    case cst2 :
        instr2 ;|set2 de instr;
        break;
    ....
    case cstn :
        instrn ;|setn de instr;
        break;
    [default :
        instrn+1 ; | setn+1 de instr;]
}
```

unde:

- **expr** este o expresie cu valoare compatibilă cu un întreg (o constantă, o variabilă, apel de funcţie, etc); nu poate fi double sau float
- cst_1, cst_2,...,cst_n sunt constante de selecţie cu valori diferite, convertibile la tipul lui expr. Fiecare etichetă **case** specifică o singură constantă,
- **break** încheie instrucţuinea **switch** prin salt la sfârşitul blocului {}. In absenţa sa execuţia programului continuă cu următoarea etichetă **case**.
- **default** , dacă apare , corespunde valorilor lui **expr** ce nu apar în lista constantelor de selecţie menţionate explicit în instrucţiune.

Setul de instrucţiuni corespunzător fiecarei etichete **case** nu trebuie inclus intre { }.

Pentru execuţie se evaluează expresia **expr** şi se compară valoarea obţinută cu constantele specificate de etichetele **case**. Dacă apare printre acestea se executa instrucţiunea corespunzătoare etichetei **case** în care a apărut. Dacă nu apare printre aceste constante se execută instrucţiunea corespunzătoare lui **default**, dacă aceasta există sau se trece la prima instrucţiune de după **switch** în caz contrar.

Instrucţiunea for

Permite repetarea uneia sau mai multor instrucţiuni de un număr fix de ori. Are sintaxa :

https://biblioteca-digitala.ro / https://unibuc.ro

for ([expr$_1$] ; [expr$_2$] ; [expr$_3$])
instr ; | {secvenţa de instr}

- Parantezele drepte indică opţionalitatea conţinutului lor.
- expr$_1$ se evaluează o singură dată inaintea primei iteraţii. Ea iniţializează variabila sau variabilele de control (contoare).
- expr$_2$ se evaluează şi testează înaintea fiecărei iteraţii. Reprezintă condiţia de ieşire din ciclu .
- expr$_3$ este evaluată la sfârşitul fiecărei iteraţii. Ea incrementează sau decrementează variabilele de control ale ciclului .

Se execută astfel: La început se iniţializează contoarele şi apoi se testează condiţia de ieşire. Dacă aceasta este adevarată se execută instrucţiunile ciclului şi se incrementează sau decrementează contoarele, repetând testarea condiţiei, altfel ciclul **for** se încheie .

Exemplu .

```
#include <iostream.h>
void main()
{
char c;
for (c = 'z' ; c >= 'a' ; c--)
        cout << c;
}
```

Acest program afişează:zyxw...dcba

Spre deosebire de Pascal în C++ limitele unui ciclu for pot fi modificate în interiorul ciclului iar variabila de control îşi păstrează valoarea şi după ieşirea din ciclu.

Exemplu.

```
#include <iostream.h>
void main()
{
        int i,s;
        for ( i=1,s=0 ; i <=10; s +=i, i++)
        cout <<"Suma primelor  "<<i-1<<" nr naturale este :"<<s;
}
```

Ciclul for poate avea două sau mai multe variabile de control .

Exemplu. Pentru inversarea unui şir de caractere :

```
#include <iostream.h>
void main()
{
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        int i, j;
        char v[5] = { "abcde" } , ch;
        cout << endl << "Sirul initial: ";
        for (i=0;i<=4;i++)
                cout << v[i] << " " ;
        for (i=0, j=4 ; i<j ; i++, j--)
        {
                ch=v[i] ; v[i] = v[j] ; v[j] = ch;
        }
        cout << endl << "Sirul inversat : ";
        for (j=0;j<=4;j++)
                cout << v[j] << " ";
}
```

Instrucțiunea while

Permite programarea ciclurilor cu test inițial pentru execuția repetată a unor instrucțiuni cât timp o anumită condiție este îndeplinită.

Are sintaxa :

```
        while (condiție)
                instr ; | {secvența de instructiuni}
```

Exemplu.

```
#include <iostream.h>
        void main()
        {
                char ch='A';
                cout << endl << "Literele alfabetului sunt : " << endl;
                while (ch<='Z')
                {
                 cout << ch << " " ;
                 ch++;
                }
        }
```

Intrucțiunea do_while

Permite programarea ciclurilor cu test final pentru execuția repetată a unor instrucțiuni până ce o anumită condiție devine falsă.

Are sintaxa :

```
                do
                   instr ; | {secventa de instr.}
                 while (conditie) ;
```

https://biblioteca-digitala.ro / https://unibuc.ro

Această instrucțiune se folosește în special pentru operațiile cu meniuri. Opțiunile meniului sunt afişate cel puțin o dată şi dacă se alege orice altă opțiune diferită de quit se execută comanda selectată , altfel se iese.

Exemplu . #include <iostream.h>

```
void main()
{
        char ch='a' ;
        cout << "Literele alfabetului :" << endl ;
        do
        {
          cout << ch << " ";
          ch++;
        }
        while (ch<='z');
}
```

Instrucțiuni de salt

Permit întreruperea necondiționată a unei secvențe de program şi continuă execuția din alt punct al programului.

Instrucțiunea break

Se foloseşte în:

- instrucțiunea switch , unde încheie secvența de instrucțiuni a etichetelor
- instrucțiunile for, while, do_while pentru a forța ieşirea din ciclu la prima instrucțiune de după cel mai interior ciclu care o conține.

Are sintaxa : **break;**

Instrucțiunea continue

Se foloseşte doar în cadrul blocurilor instrucțiunilor de ciclare. Intrerupe execuția iterației curente prin salt la sfârşitul secvenței de instrucțiuni ce formează corpul ciclului. Se continuă cu:

- testarea condiției de ciclare în **while** şi **do_while**
- actualizarea variabilelor de control şi testarea condiției de ciclare în **for**

Sintaxa este: **continue ;**

Intructiunea goto

Are sintaxa : **goto etichetă** ;

Aceasta întrerupe secvența curentă şi continuă execuția de la instrucțiunea precedată de eticheta din sintaxa instrucțiunii goto şi care se află în

https://biblioteca-digitala.ro / https://unibuc.ro

cadrul aceleiaşi funcţii . **Eticheta** precede instrucţiunea referită şi este separată de aceasta prin caracterul ":". Ea este un identificator.

Instrucţiunea return

Incheie execuţia unei funcţii şi determină revenirea la funcţia apelantă .
Are sintaxa :

return ;
return(expresie);

A doua formă transmite o valoare funcţiei apelante şi anume valoarea **expresiei** din sintaxă.

Instrucţiunea compusă(blocul)

Blocul este un grup de declaraţii urmat de instrucţiuni, toate incluse între acolade. El nu este urmat de ";".
Exemplu de instrucţiuni compuse:

- corpul unei funcţii,
- blocul ce urmează după instrucţiunea if.

Are sintaxa:

{lista de declaraţii;
lista de instrucţiuni;
}

1.8 Pointeri [47, 43, 63, 80]

Pointerul este un tip de date care stochează adresa unei zone de memorie. Diferenţa dintre un pointer şi o variabilă care are atribuită o valoare este că variabila indică totdeauna aceiaşi zonă de memorie (locul unde se află valoarea variabilei) pe când un pointer poate fi modificat încât să adreseze zone de momerie diferite.

Pointerul permite adresarea a două valori diferite

- valoarea stocată în interiorul său (adresa unei zone)
- valoarea elementului indicat de pointer (valoarea conţinută în zona indicată)

Prima valoare, ce se obţine prin **afişarea pointerului**, este adresa de memorie stocată în pointerul respectiv.
A doua, este valoarea stocată în locaţia precizată, accesul la ea fiind **referirea pointerului** prin operatorului unar * numit **referinţă** .
Privind conţinutul zonei de memorie adresate distingem :

- pointer de date, care conţine adresa unui tip de date ca: variabilă, constantă, tablou, structură, clasă

https://biblioteca-digitala.ro / https://unibuc.ro

- pointer de funcţii care conţine adresa codului executabil al unei funcţii.

Tipul de date asociat pointerilor poate fi:

- predefinit
- definit de utilizator

Adresele obiectelor (datelor şi funcţiilor) dintr-un program sunt nenule. Pointerul cu valoarea 0 nu conţine adresa nici unui obiect sau funcţie. El reprezintă pointerul NULL.
Pointerii permit alocarea dinamică a memoriei stocând adrese de memorie ale unor funcţii fără nume sau a unor date a căror dimensiune nu este dinainte cunoscută.

1.8.1 Pointeri la variabile simple

Pointerii se declară ca şi variabilele obişnuite punând "*" în faţa numelui lor sub forma : **tip *nume_pointer ;**
unde **tip** este tipul datei a cărei adresă o va memora pointerul respectiv.
La definire un pointer nu are valoare, el adresează un spaţiu aleator.
Compilatorul interpretează zona adresată de pointer ca un obiect de tipul specificat prin declaraţie cu atributele, dimensiunea şi semnificaţia informaţiei conţinute în zona respectivă.
Exemplu . int *a ; double *b ; float *c , y ;
Deoarece o variabilă pointer este un obiect putem declara pointer la pointer
Exemplu . int **a;
Aici **a** poate conţine adresa unui pointer de obiecte de tipul **int.**
Inainte de folosire o variabilă pointer trebuie iniţializată cu 0 sau cu adresa unui obiect, altfel pot apărea erori grave care pot chiar bloca programul.
Pentru a atribui adresa unei variabile unui pointer compatibil, folosim operatorul unar "&" pentru determinarea adresei, sub forma:

pointer = &variabila ;

Exemplu . int i,*p;
p=&i;

Aici se crează pointerul **p** care apoi memorează adresa variabilei i.
Pentru a accesa conţinutul locaţiei indicate de un pointer (referirea unui pointer) prin operatorul referinţă * folosim sintaxa :

variabila = *pointer;

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu.

```
# include <iostream.h>
void main()
{
        int i,j,*p;
        cout << "dati un intreg: ";
        cin >> i;
        cout << "variabila i are valoarea " << i << endl ;
        p=&i;
        cout << adresa variabilei i este : " << p << endl;
        j=*p;
        cout << "variabila j are valoarea " << j << endl;
}
```

Aici pointerului p i se atribuie adresa variabilei i iar lui j i se atribuie conţinutul locaţiei indicată de p. Deci i şi j au aceiaşi valoare .

Putem modifica valoarea stocată într-o locaţie indicată de un pointer prin sintaxa : ***pointer = valoare;**

unde **valoare** este noua valoare stocată în locaţia indicată de **pointer**.

Pentru a crea variabile dinamice folosim operatorii **new** şi **delete** cu sintaxa :

pointer = new tip ; ← dă adresa zonei de memorie alocată
delete pointer; ←eliberează zona de memorie

Dacă alocarea nu reuşeste **new** întoarce pointerul NULL .

Prin ştergerea unei zone de memorie alocată dinamic, pointerul care o indică nu se modifică. Pentru a evita apariţia unor erori ştergem şi pointerul prin atribuirea valorii zero.

Exemplu .

```
int *p;          ←crează pointer la un întreg
p=new int;       ←crează un întreg, adică alocă memorie
                   pentru un întreg care apoi primeşte valoarea 5
*p=5;
delete p;        ←şterge întregul (eliberează zona indicată de p)
p=0;             ←şterge pointerul
```

1.8.2 Pointeri la tablouri

Numele unui tablou este un pointer constant de tipul elementelor tabloului.Valoarea sa este adresa primului element al tabloului. Astfel:

- pentru vectorul **vect** , expresiile vect şi &vect[0] sunt echivalente

https://biblioteca-digitala.ro / https://unibuc.ro

- pentru matricea **mat** , expresiile mat şi &mat[0][0] sunt echivalente

Pointerii permit accesarea elementelor tabloului prin operaţia de adunare dintre pointeri şi întregi. În acest caz adresa de bază a unui tablou **tab** este locaţia primului său element, dată să spunem de pointerul p. Adresa elementului tab[i] din tablou va fi dată de expresia p+i .

Trebuie făcută diferenţa între pointeri la tablouri şi tablouri de pointeri.

Exemplu.

```
int t[n][n] ;     ← tablou cu nxn elemente
int *p[n]    ← tablou de pointeri  la care se alocă n pointeri
```

Fiecare din cei n pointeri ,indică (pointează) către un tablou de intregi , cu lungimi care pot fi diferite. Accesarea unui element se face indirect printr-un pointer fără a fi necesară indexarea.

Alocarea şi dealocarea (eliberarea) dinamică a unui tablou foloseşte sintaxa:

```
pointer-la-tablou = new tip-date-tablou[dim-tablou] ;
delete pointer-la-tablou ;
```

Exemplul 1.

```
#include <iostream.h>
void main()
{
        int v[10] = {1,2,3,4,5,6,7,8,9,10};
        int *pi, *pf, s=0, *ps;
        pi  =  &v[0];pf = &v[9];ps = &s;
        for( ; pi<=pf ; pi++)
                *ps+=*pi;
        cout << "suma componentelor lui v este: " << *ps;
}
```

Acest program calculează suma elementelor unui vector iniţializat la definirea sa. Pointerii pi şi pf memorează adresele de început respectiv de sfârşit ale elementelor tabloului.

Exemplul 2.

```
#include <iostream.h>
void main()
{
        int k;double *p;
        p=new double[4];
        cout << "tabloul creat are elementele: " << endl;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        for (k=0;k<4;k++)
        {
                p[k]=(double) k*k;
                cout << "elem(" << k << ")=" << *(p+k) << endl;
        }
        delete []p;
}
```

Aici se crează un tablou cu elementele 0,1,4,9 şi se folosesc pointeri pentru accesarea elementelor tabloului creat.

1.8.3 Pointeri la şiruri de caractere [12, 47]

Şirurile de caractere sunt considerate tablouri cu elemente de tip char care se încheie cu caracterul "\0" . Funcţiile pentru şiruri de caractere, incluse în fişierul antet **string.h,** manipulează şirurile de caractere prin pointeri la tipul de date **char**. Acestea sunt :

- **strcat()**-adaugă şirul sursă la sfârşitul şirului destinaţie
- **strchr()**-indică prima apariţie a unui caracter în şirul destinaţie
- **strcmp()**-compară două şiruri
- **strcpy()**-copiază un şir în altul
- **strcspn()**-parcurge un şir şi dă lungimea primului subşir care nu are nici un caracter într-un al doilea şir
- **strdup()**-creează un duplicat al unui şir prin copiere într-o zonă special alocată
- **stricmp()**-compară două şiruri fără a face diferenţa între majuscule şi minuscule
- **strlen()**-dă lungimea unui şir
- **strlwr()**-converteşte majusculele în minuscule
- **strncat()**-adaugă un număr specificat de caractere ale unui şir sursă la sfârşitul unui şir destinaţie
- **strncmp()**-compară un număr specificat de caractere de la începutul a două şiruri
- **strncpy()**-copiază un număr specificat de caractere din fişierul sursă în cel destinaţie
- **strnicmp()**-compară un număr specificat de caractere de la începutul a două şiruri ignorând diferenţele dintre majuscule şi minuscule

https://biblioteca-digitala.ro / https://unibuc.ro

- **strnset()**-suprascrie un număr specificat de caractere dintr-un şir cu copii ale unui singur caracter
- **strrchr()**-caută ultima apariţie a unui caracter într-un şir
- **strrev()**-inversează ordinea caracterelor unui şir
- **strset()**-înlocuie toate caracterele unui şir, excepţie terminatorul de şir, cu un caracter
- **strspn()**-dă numărul caracterelor din prima parte a unui şir, similare cu unul din caracterele unui şir şablon
- **strstr()**-caută prima apariţie a unui subşir într-un şir dat
- **strtod()**-converteşte un şir de caractere în double
- **strtok()**-caută anumite simboluri sau subşiruri dintr-un şir; delimitatorii sunt daşi printr-un al doilea şir de caractere
- **strtol ()**-converteşte un şir de caractere într-un întreg de tip long zecimal, octal sau hexazecimal
- **strupr()**-converteşte minusculele în majuscule

Exemplu. Se defineşte şi iniţializează vectorul de pointeri **ps** prin care se ordonează alfabetic un şir de caractere ce conţine numele unor persoane. Interschimbarea adreselor în vectorul de pointeri ps se face prin funcţia predefinită **strcmp()** .

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
void main()
{
        clrscr();
        int i,j;
        char *temp;
        char *ps[]={"Paul", "Vlad", "Andrei", "Iancu", "Gheorghe" ,
        "Ion", "Bogdan","Dumitru","Radu","Dan" };
        cout << "Numele initiale sunt: " << endl;
        for(i=0;i<10;i++)
                cout << ps[i] << ", ";
        for(j=0;j<10;j++)
                for(i=j;i<10;i++)
                        if (strcmp(ps[j], ps[i])>0)
                                {
                                temp = ps[j];ps[j]=ps[i];ps[i]=temp;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                                    }
        cout << endl;
        cout << endl << "Numele ordonate sunt:" << endl << "....." << endl;
        for(i=0;i<10;i++)
                cout << ps[i] << ", ";
}
```

1.8.4 Pointeri la structuri

In C++ putem transmite unei funcţii o structură atât prin valoare cât şi prin adresa (pointer). Se ştie că structurile pot fi iniţializate chiar în momentul declarării lor.
Atribuirea adresei unei variabile de tip structură către un pointer se face ca la variabilele simple. Pentru a accesa un membru al unei structuri putem folosi una din formele :

pointer-structura -> membru-structura ;
(*pointer-structura) . membru-structura ;

Exemplu: Se calculează şi se afişează coordonatele mijlocului segmentului dat de punctele a şi b .

```
#include <iostream.h>
void main()
{
        struct coord {
                double x;
                double y;};

        coord a = {1.0, 4.0}, b = {2.0, 7.0};
        coord *p1, *p2;
        p1 = &a;p2=&b;
        cout << "Mijlocul segmentului ab este punctul : (" << (p1→x+
        p2→x)/2 << "," << ((*p1).y+(*p2).y)/2 << ")." << endl;
}
```

Apare pe ecran mesajul:
Mijlocul segmentului ab este punctul : (1.5,5.5).

https://biblioteca-digitala.ro / https://unibuc.ro

1.9 Funcţii [12, 47, 59, 63]

Un program în C++ este o succesiune de funcţii, succesiune în care fiecare funcţie îndeplineşte o sarcină bine definită. Funcţia este o colecţie de declaraţii şi instrucţiuni care execută o acţiune.
Funcţiile :

- sunt esenţiale în construcţia blocurilor
- permit folosirea facilităţilor oferite de programarea structurată
- permit elaborarea unor aplicaţii proprii utilizatorului
- permit obţinerea unor funcţii mai complexe prin combinarea unora mai simple, care scriu, citesc, testează şi depanează programe

Un program C++ conţine cel putin o funcţie **main()** care este apelată la lansarea în execuţie a programului, funcţii predefinite (de bibliotecă) şi funcţii construite de programator.
Inainte de utilizarea unei funcţii ea trebuie fie definită, fie declarată.

1.9.1 Definirea funcţiilor

Funcţiile se definesc prin sintaxa :

```
tip-rez nume(lista)
{
decl-locale
  instr
}
```

unde:
tip-rez = tipul reultatului întors de funcţie. Poate fi orice tip diferit de tablou. Implicit se consideră tipul int. Dacă funcţia nu întoarce rezultate, se foloseşte cuvântul **void**.
lista = o listă de declaraţii ale parametrilor separaţi prin ",". Ea are forma: $tip_1\ np_1$, $tip_2\ np_2$,...., np_i=parametru formal , i=1,2,...
Daca lista este vidă se pune ( ) sau (void).
Nu este permisă definirea unei funcţii în blocul { } altei funcţii. Nu este permisă ieşirea dintr-o funcţie în afara ei prin instrucţiunea goto. O funcţie se execută la apelarea ei prin una din formele:

```
nume (lista_argumente);
nume ( );
```

https://biblioteca-digitala.ro / https://unibuc.ro

unde **lista_argumente** conține valorile parametrilor din lista parametrilor funcției. O funcție se poate apela ori de cite ori este nevoie. După executarea instrucțiunilor unei funcții se continuă executarea programului cu prima instructiune de după cea de apel a funcției. Incheierea execuției unei funcții şi revenirea în funcția apelantă se face prin instructiunea **return** care are una din sintaxele:

return(expr);
return expr;
return;

Valoarea expresiei **expr** este rezultatul funcției. Ea este convertită la tipul funcției înainte de a se transmite în funcția apelantă. Funcțiile de tip **void** nu întorc valoare şi deci return are aici doar forma **return;** .

Exemplul 1.

```
#include<iostream.h>.
void  desen(void)
{
cout<<"*********"<<endl;
  for(int k=1; k<=9; k++)
      cont<<"*          *"<<endl;
       cont<<"*********"<<endl;
return;
}
void main(void)
{
 desen();
 }
```

Acest program desenează un dreptunghi din *-uri.

Exemplul 2.

```
#include<iostream.h>
float zece(float x)
 {
 float y=1;
 for(int k=1; k<=10; k++)
       y*=x;
 return y;
 }
 void main()
 {
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
    cout<<endl<<zece(3.5);
     }
```

Acest program calculează 3.5^{10} cu ajutorul funcției zece.

1.9.2 Prototipul unei funcții

Dacă o funcție este apelată dintr-o bibliotecă standard sau definită de utilizator sub forma de fişier obiect sau dacă se află într-un alt fişier sursă, pentru a verifica validitatea apelurilor de către compilator este obligatorie folosirea declarației acelei funcții, declarație numită **prototip**. Declarația are forma :

tip-rez nume(lista);

unde **lista** poate conține doar tipul parametrilor şi numărul lor fără a fi neapărat necesară şi specificarea numelui lor (care este însă recomandată).

Prototipul unei funcții trebuie pus înaintea primului apel al său, de obicei chiar la începutul programului.

Exemplu .

```
#include<iostream.h>
const n=3;
void citire(int mat[n][n]);
void tiparire(int mat[n][n]);
int  urma (int mat[n][n]);
 int sum, mat[n][n];
 void main( )
 {
 int k,nr;
 cout<<"Introduceti numarul matricelor:";
 cin>>nr;
 for(k=1;k<=nr;k++)
{
 cout<<endl<<"Tastam matricea"<<k<<":"<<endl;
citire(mat);
cout<<endl<<"Matricea"<<k<<"este:"<<endl;
tiparire(mat);
urma(mat);
cout<<endl<<"Urma matricei "<<k<<" este: "<<sum;
   }
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        }
        void citire(int mat[n][n])
        {
        for(int i=0;i<n;i++)
        {
        cout<<"Introdu elementele liniei "<<i+1<<":"<<endl;
        for(int j=0;j<n;j++)
        cin>>mat[i][j];
        }
       return;
       }
void tiparire (int mat[n][n])
{
for (int i=0; i<n; i++)
        {
           cout <<endl;
           for (int j=0; j<n; j++)
                cout <<"  "  << mat [i][j] <<"  "   ;
        }
        cout<<endl;
        return;
}
int urma (int mat [n][n])
{ sum=0;
  for (int i=0; i<n; i++)
        sum+=mat[i][i];
  return sum;
}
```

1.9.3 Funcţii recursive

Dacă o funcţie se autoapelează direct sau indirect se numeşte **recursivă**. Astfel apelul unei funcţii recursive poate apărea şi în definiţia sa.
O funcţie recursivă nu are sintaxă specială. Ea implică existenţa unei **stive** care la fiecare apel al funcţiei recursive se încarcă cu valorile parametrilor, variabilelor locale şi rezultatului. La fiecare revenire, această stivă este descarcată.

https://biblioteca-digitala.ro / https://unibuc.ro

Recursivitatea se aplică în special la prelucrarea structurilor de date de tip recursiv cum ar fi liste, arbori, etc.

Exemplu. Calculul nerecursiv şi recursiv al factorialului unui număr natural.

Calcul nerecursiv

```
#include < iostream.h>
int factorial (int n)
{
  int rez=1;
  for (int i=1; i<=n; i++)
     rez*=i;
  return rez;
}
void main ()
{
  int num;
  cout << "Introduceti un numar natural:";
  cin >>num;
  cout << endl << " Rezultat:"<< num <<"!="<<
        factorial (num) << endl;
}
```

Calcul recursiv

```
 #include < iostream.h>
int fact (int n)
 {
if (n==0)
     return 1;
 else
    return n*fact(n-1);
 }
 void main ()
 {
 int num;
cout << "Introduceti un numar natural:";
  cin >>num;
  cout << endl << " Rezultat:"<< num <<
  "!="<<fact (num) << endl;
 }
```

https://biblioteca-digitala.ro / https://unibuc.ro

1.9.4 Funcţii cu număr variabil de parametrii

Pentru astfel de funcţii la declarare se recomandă a se specifica totdeauna numărul parametrilor fie prin menţionarea parametrilor formali fie prin ... sau void.

Exemplu. void fun (int a, char b,...);

Aici funcţia **fun** conţine 2 parametrii a şi b dar mai pot apărea şi alti parametrii.

Dacă nu se cunoaşte numărul şi tipul parametrilor funcţiei se foloseşte fişierul *stdarg.h* care declară un pointer către lista de parametrii *va_list* şi macrodefiniţiile *va_arg*, *va_end* şi *va_start* care permit un acces portabil la lista de parametrii ai funcţiei.

La definirea funcţiei se declară o variabilă *ap* de tip *va_list* care permite adresarea parametrilor.

va_start () iniţializează variabila *ap* cu adresa primului parametru din lista cu parametrii ai funcţiei.

va_arg () conţine parametrii variabili într-o singură expresie din care se pot extrage unul câte unul. La fiecare apelare *va_arg()* întoarce valoarea parametrului curent din listă, indicat de variabila *ap*. Aceasta este dereferita şi incrementată pentru a indica următorul parametru.

va_end () nu întoarce nici o valoare ci finalizează operaţia de extragere a parametrilor şi determină un retur normal din funcţia apelată. Trebuie ca *va_start()* să fie apelată înaintea folosirii celorlalte două macrodefiniţii *va_arg* () şi *va_end* ().

Exemplu.

Funcţia *prod* are un număr variabil de parametrii, primul fiind de tip char. Ea întoarce valoarea produsului parametrilor efectivi indicaţi în apelul ei, care sunt numerele listate până la întâlnirea numărului "0".

```
# include < iostream.h>
# include < stdarg.h>
void prod (char*mes,...)
{
int rez=1, par;
va_list ap;
va_start (ap,mes);
while ((par=va_arg(ap,int))!=0)
                {rez*=par;}
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout <<mes<<rez<<endl;
va_end (ap);
 }
 void main ()
 {
  prod ("Prod. nr. 3,5,2, este",3,5,2,0);
  prod ("Prod. nr. 2,1,4,7 este",2,1,4,7,0);
 }
```

După compilare şi rulare programul afişează:
Prod. nr. 3,5,2 este 30
Prod. nr. 2,1,4,7 este 56

1.9.5 Parametrii cu valori implicite

C + + permite declararea funcţiilor cu valori prestabilite ale parametrilor. Dacă la apelul unor astfel de funcţii se omite valoarea argumentului pentru parametrii formali cu valorii prestabilite, acestora le sunt automat transferate valorile implicite.
Pentru folosirea parametrilor cu valori implicite trebuie respectate regulile următoare:

- Valorile implicite se specifică o singură dată în prototip sau în definiţie;
- Parametrii cu valori implicite se trec în lista după cei fără valori implicite;
- La apel se specifică un argument pentru fiecare parametru fără valoare prestabilită; pentru ceilalţi argumentul se poate omite;
- Parametrii care urmează unui parametru cu valoare implicită al cărui argument a fost omis, vor avea argumentele omise;
- Parametrii cu valori implicite se trec în listă în ordinea descrescătoare a probabilităţilor de folosire a argumentelor lor.

Exemplu.

```
#include < iostream.h>
void f(int, int=5);
void main ()
{
cout <<"Apelul f(20,10) implica:";
  f(20,10);
cout <<"Apelul f(20) implica:";
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
  f(20);
}
void f(int i, int j)
{
cout <<i<<"+" <<j<< "=" <<i+j<<endl;
}
```

Programul afişează: Apelul f(20,10) implica: 20+10=30
Apelul f(20) implica 20+5=25

1.9.6 Supradefinirea funcţiilor

C ++ permite definirea mai multor funcţii cu acelaşi **nume** dar cu liste de parametrii (semnături) diferite ca tip sau ca număr al parametrilor.
Supradefinirea funcţiilor care au parametrii cu valori implicite poate genera duplicarea semnăturii funcţiilor respective, ambiguitate pe care compilatorul o semnlează ca eroare.
Această facilitate permite obţinerea unui rezultat chiar dacă tipurile parametrilor diferă. Ea contribuie la simplificarea programării pentru că permite folosirea unui singur nume de funcţie când programul trebuie să îndeplinească o anumită sarcină.

Exemplu:

```
#include < iostrem.h>
int adun (int x, int y)
{
return (x+y);
}
int adun (int x, int y, int z)
{
return (x+y+z)
}
void main ()
{
cout << "Rezultate:"<<endl;
cout << "l0+20="<<adun (10,20)<<endl;
cout << "15+25+35"=<<adun(15,25,35)<<endl;
}
```

După compilare şi rulare apare pe ecran:
Rezultate:

https://biblioteca-digitala.ro / https://unibuc.ro

10+20=30
15+25+35=75

1.9.7 Tipuri speciale de parametrii

1.9.7.1 Parametrul tablou

Funcţiile care transmit ca parametru un tablou necesită declararea acelui parametru ca pointer la tipul de bază al tabloului.
Prototipul funcţiei are sintaxa:

tip_baza nume_tablou;
tip_rez nume_fct (tip_baza*, alte_tipuri);

Definirea unei funcţii cu parametru tablou are sintaxa:

tip_rez nume_fct (tip_baza*tab, alti_parametrii)
declara tab printr-un pointer
tip_rez nume_fct (tip_baza tab[], alti parametrii)
declara tab prin []

Exemplu.

```
#include < iostream.h>
void main ()
{
int suma (int *, int);
int vector[] = {4,2,1,5,3,6};
int dim = sizeof (vector) / sizeof (int);
cout << "Suma elementelor vectorului este:" << suma (vector,
dim)<<endl;
}
int suma (int*p, int n)
{
for (int i=0, s=0; i<n; i++)
  s + = p[i];
  return s;
}
```

Funcţia calculează suma elementelor lui vector. Parametrul *int** este tipul de bază al parametrului vector.

1.9.7.2 Parametrul şir de caractere

Şirurile de caractere în C + + sunt considerate tablouri de caractere. Dacă un şir de caractere este parametrul unei funcţii, pentru el sunt valabile

https://biblioteca-digitala.ro / https://unibuc.ro

aceleaşi reguli ca la parametrul tablou.

Exemplu. Programul următor calculează numărul de caractere litere mici dintr-un şir de caractere prin funcţia *contor* care primeşte ca parametru respectivul şir. Introducerea şirului se face cu funcşia de bibliotecă predefinită *gets*() din fişierul antet *stdio.h*. Funcţia *contor* atribuie adresa parametrului şir unui pointer local *psir*. Acesta este folosit de funcţie pentru procesarea caracterelor din şir.

```
#include < iostream.h>
#include < stdio.h>
int contor (char *sir)
{
int k=0;
char *psir = sir;
while (*psir! = '\0') {
                        if (* psir >='a' && * psir<='z')
                            k + = 1;
                        psir + + ;}
return k;
}
void main ()
{
char sir[20];
cout << "Introduceti sirul:";
gets (sir);
cout << "Sirul introdus" << sir << "contine" << contor (sir) <<
"caractere mici"<<endl;
}
```

După compilare şi rulare programul afişează:

Sirul introdus 'Asd$%90bc' contine 4 caractere mici.
Sirul introdus 'UN907.12CD' contine 0 caractere mici.

1.9.7.3 Parametrul structură [12]

Acest parametru se poate transfera funcţiilor C ++ prin valoare sau prin referinţă. Tipul structurii apare în prototip şi în antetul funcţiei ca la tipurile predefinite.

Exemplu.

```
#include < iostream.h>
struct complex{float x;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                    float y;
                    };
complex operatie (complex, complex);
void main ()
{
complex c1,c2;
cout << "Introduceti componenetele lui c1:" <<endl;
cin >> c1.x >> c1.y;
cout << "Introduceti componenetele lui c2:" <<endl;
cin >> c2.x >> c2.y;
cout << "Parametrii transmisi sunt: c1=("<<c1.x<<
","<<c1.y<<") şi c2 = (" <<c2.x <<","<<c2.y<<").";
complex c = operatie (c1, c2);
cout <<endl<< "Rezultatul operatiei este: c = ("<<c.x<<","
<<c.y<<")." <<endl;
}
complex operatie (complex a, complex b)
{
complex rez;
rez.x=a.x*b.x - a.y*b.y;
rez.y=a.x*b.y + a.y*b.x;
return rez;
}
```

Un exemplu de sesiune de lucru cu acest program ar fi:

```
Intodduceți componentele lui c1: 1
                                 2
Intoduceți componentele lui c2:  -1
                                 3
Parametrii transmişi sunt: c1=(1,2) şi c2=(-1,3)
  Rezultatul operației este: c=(-7,1)
```

1.9.8 Transferul parametrilor prin referință

Functiile C + + transferă în mod normal parametrii prin valoare. Ele primesc valorile efective ale parametrilor sub forma de copii ale datelor originale. Pot modifica în interiorul lor valorile parametrilor prin pointeri fără a afecta datele originale. Modificarea valorilor parametrilor se poate

https://biblioteca-digitala.ro / https://unibuc.ro

face mai simplu prin transferul parametrilor nu prin valoare, ci prin referință, eliminând instrucțiunile care combină variabilele pointer cu cele normale.

O **referința** crează un nume alternativ (**alias**) pentru o variabilă. Se declară prin sintaxă:

tip& nume_alias = variabila;

unde: & (ampersand) se pune imediat dupa tip.

variabila = este cea pentru care referința este un alias

Exemplu: int& intreg = n;
double& precizie = eps;
float& media_adm = media;

După declararea unei **referințe** în program putem folosi atat variabila cât şi referința.

Referința nu este o variabilă. Ea nu mai poate fi modificată după ce a fost asociată unei variabile. Folosirea unei referințe ca identificator este utilă când aceasta apare ca parametru formal al unei funcții. Spre deosebire de pointeri, referințele nu permit operațiile:

- atribuirea unui pointer o referință;
- obținerea adresei unei referințe cu operatorul *adresa*;
- compararea valorilor referințelor prin operatorii relaționali;
- operații aritmetice cum ar fi adunarea unui deplasament(a unei adrese).

Transferul parametrilor unei funcții prin referință este făcut de compilator, ceea ce simplifică scrierea funcției şi apelul ei.

Transferul parametrilor de tip structură prin referință formale ca şi cel prin pointeri este mai eficient decât transferul prin valoare, deoarece elimină copierea structurii pe stivă, conducând astfel la creşterea vitezei de execuție. Rezultatul unei funcții poate fi transferat prin valoare, pointer sau referință.

Exemplu. Funcția operație() are 3 parametrii referință, al treilea obține informații de la funcția apelată şi întoarace rezultatul funcției.

```
#include < iostream.h>
#include < conio.h>
struct comp {
      float x;
      float y;};
void operatie (comp& a, comp& b, comp& r)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
{
r.x = a.x * b.x - a.y * b.y;
r.y = a.x * b.y + a.y * b.x;
}
void main ()
comp c1, c2, c;
cout << "Introdu componentele lui c1:" <<endl;
cin >>c1.x>>c1.y;
cout << "Introdu componentele lui c2:" <<endl;
cin >>c2.x>>c2.y;
cout << "Au fost introduse: c1=("<< c1.x<<", "<< c1.y<<") şi
        c2=("<< c2.x<<", "<< c2.y<<") ." endl;
operatie (c1, c2, c);
cout << "Rezultatul operatiei este: c=("<< c.x<<", "<< c.y<<")
."<<endl;
```

1.9.9 Pointeri la funcţii [12, 47]

În C + + funcţia nu este o variabilă. Compilatorul transformă numele funcţiei într-o adresă de cod executabil. Ştim că pointerii la adrese de memorie pot accesa adresele.

Putem defini un pointer sau un tablou de pointeri la o funcţie ceea ce ne permite să extindem strategia manipulării variabilelor prin pointeri şi la funcţii.

Un pointer la o funcţie se declară prin:

tip_rez (*pointer_funcţie) (lista_parametrii);

unde:

tip_rez este tipul rezultatului care se returnează

pointer_funcţie este numele pointerului la funcţie

lista_parametrii este de forma: tip_par_1, tip_par_2,....

Exemplu.

int(*pf) (int i); ← *pf* este pointer la o funcţie care intoarce un rezultat întreg şi are un singur parametru de tip întreg

double (*h) (double x, double y); ← *h* este un pointer la o funcţie care întoarce un rezultat de tip double şi are 2 parametrii de tip double

void (*sort) (char *); ← *sort* este un pointer la o funcţie care nu întoarce rezultate şi are ca parametru un şir de caractere

https://biblioteca-digitala.ro / https://unibuc.ro

Inaintea folosirii unui pointer la o funcție acesta trebuie inițializat sub forma: **pointer_funcție = nume_funcție;**
unde *nume_funcție* este funcția atribuită pointerului la funcție, care trebuie să îndeplinească următoarele condiții:

1. tipul rezultatului întors de funcție să fie acelaşi cu cel întors de pointer;
2. lista parametrilor funcției să fie aceeaşi cu cea a pointerului.

Apelul pointerului la o funcție are sintaxa:

(*pointer_funcție) (lista_argumente);

Exemplu. Pointerului *pf* îi este atribuită adresa funcției f () care calculează suma a 2 întregi.

```
#include < iostream.h>
int f(int i, int j)
{
return i+j;
}
void main ()
{
int x,y;
int (*pf) (int, int);
cout << " Introduceti intregii x şi y:" <<endl;
cin >>x>>y;
cout<<"Numerele introduse sunt:<<endl;
cout << " x="<<x<<" şi y="<<y<<" endl;
pf = f;
cout <<"Rezultat:";
cout <<x<<"+"<<y<<"="<<(*pf) (x,y)<<endl;
}
```

Putem declara un tablou de pointeri către o funcție sub forma:

tip_rez (*pointer_funcție [dim_tablou]) (lista_param.);

Exemplu.

int (*pf [3]) (int i); ←declară un tablou *pf* de 3 pointeri la o funcție. Fiecare element al tabloului pointează(indică) spre o funcție ce întoarce un rezultat de tip int

double (*pg [2]) (double x, double y); ← declară un tablou *pg* de 2 pointeri la o funcție. Fiecare element al

https://biblioteca-digitala.ro / https://unibuc.ro

tabloului pointează (indică) către o funcție ce întoarce un rezultat de tip double şi care are 2 parametrii de tip double

Pentru a atribui o funcție unui element dintr-un tablou de pointeri la o funcție, folosim sintaxa:

pointer_fct [indice] = nume_funcție;

Fiecare pointer trebuie să întoarcă acelaşi tip de rezultat ca şi funcția accesată şi să aibă aceiaşi listă de parametrii.

Apelul unui tablou de pointeri la o funcție are sintaxa:

(* pointer_fct [indice]) (lista_argumente);

Exemplu. [12] Un program ce conține tabloul de pointeri pf la o funcție, tablou cu două elemente, fiecare indicând către o funcție min () respectiv max () care întorc un rezultat de tip int şi au 2 parametrii de tip int, primul fiind un tablou unidimensional. Cu acest program calculăm cel mai mic şi cel mai mare element al unui vector.

```
#include < iostream.h>
#include < conio.h>
int min (int w[], int n)
{
int minw = w[0], i;
for (i=1; i<n; i++)
        if (minw > w[i]) minw = w[i];
cout <<"Elementul minim este:";
return minw;
}
int max (int w[], int n)
{
int maxw = w[0], i;
for (i=1; i<n; i++)
        if (maxw < w[i]) maxw = w[i];
cout <<"Elementul maxim este:";
return maxw;
}
void main ()
{
int n, i, v[10];
clrscr();
cout <<"Introduceti numarul  elementelor (max 10):";
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        cin >>n;
        cout <<"Introduceti elementele vectorului (intregi):"<<endl;
        for (i=0, i<n, i+ +)
                cin >> v[i];
        cout <<" Ati introdus vectorul :"<<endl;
        cout <<"v["<<n<<"]={";
        for (i=0, i<n-1, i+ +)
                cout <<v[i]<<",";
        cout <<v[n-1]<<"}"<<endl;
        int (*pf [2]) (int [], int);
        pf [0] = min; pf [1] = max;
        cout <<"Rezultate:"<<endl;
        for (i=0, i<2, i+ +)
                cout <<(*pf [i])(v,n) <endl;
}
```

Observaţie. Pointerii la funcţii permit crearea de biblioteci de funcţii care pot fi folosite de către funcţii nedefinite în momentul apelului.

1.9.10 Funcţii de biblioteca [12, 47, 80]

Compilatorul C + + pune la dispoziţia utilizatorului o mulţime de funcţii predefinite în fişiere antet din biblioteca cu fisiere executabile (biblioteca de execuţie), care permit elaboarrea programelor de dimensiuni mari.
Pentru a folosi aceste funcţii trebuie inclus în program fişierul antet respectiv prin directiva #include şi trebuie specificat prototipul acestor funcţii, prototip care se află în fişierul antet inclus în program.
Printre fişierele antet, enumerăm:

- **assert.h** – diagnostichează programul
- **complex.h** – declară clasa *complex* pentru operaţii cu numere complexe
- **conio.h** – defineşte I/ E de la consolă în MS DOS
- **ctype.h** – declară funcţii pentru clasificare şi convertire de caractere
- **graphics.h** – defineşte pachetul de funcţii GRAPHICS
- **float.h** – defineşte un domeniu de valori care pot fi stocate în tipurile virgulă mobilă
- **fstream.h** – defineşte o clasă ce reprezintă fluxul de ieşire ofstream

https://biblioteca-digitala.ro / https://unibuc.ro

- **iostream.h** – defineşte fluxul de intrare *cin* şi de ieşire *cout*
- **limits.h** – defineşte valorile limită pentru tipurile de date întregi
- **math.h** – defineşte funcţii matematice pentru operaţii cu numere reale
- **search.h** – declară funcţii de căutare
- **signal.h** – defineşte simbolurile şi rutinele pentru tratarea condiţiilor excepţionale
- **stdarg.h** – defineşte macrocomenzi care furnizează acces la argumente fără nume dintr-o funcţie cu număr variabil de parametrii
- **stdlib.h**—declară rutinele de conversie de şiruri, de alocare a memoriei, de control a proceselor şi generatorul de numere aleatoare
- **stdio.h** – declară funcţii şi tipuri de date necesare operaţiilor de I/E
- **string.h** – declară rutinele de manipulare a şirurilor
- **time.h**—defineşte tipuri de date şi declară funcţii care manipulează timpul.

Printre funcţiile matematice cu argumente reale din fişierul antet '**math.h**' enumerăm:

- **abs** () cu prototipul: **int abs (int x);** – dă valoarea absolută a unui întreg
- **acos** () cu prototipul: **double acos (double x);** – calculează arccos x, x real
- **asin** () cu prototipul: **double asin (double x);** – calculează arcsin x, x real
- **atan** () cu prototipul: **double atan (double x);** – calculează arctg x ,x real
- **atan2** () cu prototipul: **double atan2 (double y, double x);** – calculează arctg y/ x ,x, y reali
- **ceil** () cu prototipul: **double ceil (double x);** – dă cel mai mic întreg $\geq x$
- **cos** () cu prototipul: **double cos (double x);** – calculează cos x, cu x real în radiani
- **cosh** () cu prototipul:**double cosh (double x);** – dă ch x, cu x real
- **exp** () cu prototipul: **double exp (double x);** – calculează e^x , x real

https://biblioteca-digitala.ro / https://unibuc.ro

- **fabs()** cu prototipul : **double fabs(double x);**—dă |x|, pentru x real
- **floor** () cu prototipul: **double floor(double x);** – dă [x] , x real
- **fmod()** cu prototipul: **double fmod(double x,double y)**;--dă restul împărţirii lui x la y cu cât întreg
- **frexp** () cu prototipul: **double frexp(double x, int *expptr);** – separă x în mantisă şi exponentul puterii lui 2
- **ldexp** () cu prototipul**: double ldexp(double x, int exp)**;-- reconstruieşte x din mantisă şi exponentul puterii lui 2 (calculează $x*2^{exp}$)
- **log** () cu prototipul**: double log(double x)**; – calculează lnx ,x>0
- **log10** () cu prototipul**: double log10(double x)**; – calculează lgx ,x>0
- **modf()** cu prototipul: **double modf(double x, double *intptr)**;-- separă x în [x] şi {x}
- **pow()** cu prototipul: **double pow(double x, double y)**;-- calculează x^y
- **sin()** cu prototipul: **double sin(double x)**;-- calculează sin x , x real în radiani
- **sinh** () cu prototipul: **double sinh(double x)**; – calculează sh x, x real
- **sqrt** () cu prototipul: **double sqrt(double x)**; – calculează $\sqrt{x}$,x>=0
- **tan** () cu prototipul: **double tan(double x)**; – calculează tg x, cu x în radiani
- **tanh** () cu prototipul: **double tanh(double x)** ;– calculează th x, x real

Exemplu.

```
#include <iostream.h>
#include <math.h>
void main ()
{
double x, y;
cout << "Introdu numerele x şi y strict pozitive:";cin >>x>>y;
cout << "Radical din"<<x<<"este"<<sqrt (x)<<endl;
cout << "Logaritm natural din"<<x<<"este"<<log (x)<<endl;
cout << "Logaritm zecimal din"<<y<<"este"<<log10 (y)<<endl;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout << "Exponentiala lui"<<x<<"este"<<exp (x)<<endl;
cout<<x<<" ridicat la puterea "<<y<<" este "<<pow(x,y)<<endl;
}
```

1.9.11 Funcţii inline

Folosirea funcţiilor în programe creste durata de execuţie a acestora deoarece la apelul unei funcţii compilatorul C++ plasează parametrii acesteia pe stivă, ramifică execuţia programului la instrucţiunile funcţiei şi apoi revine la prima instrucţiune din program de după aceea de apel a funcţiei.

Pentru reducerea apelurilor de functii, C++ oferă funcţi inline, care sunt combinaţii între expresii macro şi funcţii. Ca şi macrodefiniţiile, la apelul unei funcţii inline se inserează la poziţia apelului instrucţiuni echivalente celor din cadrul funcţiilor respective. Spre deosebire de macrodefiniţii aici se verifică tipurile parametrilor care se transmit ca şi la funcţiile normale. Funcţiile inline se declară prin:

inline tip_rez nume_fct (lista_parametrii)

Corpul unei funcţii inline se defineţte în momentul declarării funcţiei ca fiind inline.

Exemplu: Un program care defineşte funcţiile **sum** şi **prod** ca fiind inline.

```
#include <iostream.h>
inline double sum(double x, double y)
{
return (x+y);
}
inline double prod(double x, double y)
{
return (x*y);
}
void main ()
{
double a, b;
cout << "Introdu doua numere reale:" <<endl; cin >>a>>b;
cout << "Rezultate:" <<endl;
cout << "Suma numerelor " <<a<<" şi "<<b<<" este "<<sum
(a,b)<<endl;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout << "Produsul numerelor " <<a<<" şi "<<b<<" este "<<prod
(a,b)<<endl;
}
```

Observaţie. Prin folosirea funcţiilor **inline** creşte performanţa programului datorită eliminării consumului de timp implicat de apelurile funcţiilor. Corpul unei astfel de funcţii este însă duplicat pentru fiecare apel şi de aceea se recomandă folosirea funcţiilor inline numai pentru funcţii de dimensiuni mici.

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL II Elemente de programare orientată obiect (POO)

2.1 Concepte de bază [12, 58]

Conceptele fundamentale ale POO sunt noţiunile de *clasa* şi *obiect* precum şi *relaţiile* dintre clase, obiecte şi dintre clase şi obiecte.
Obiectul este o entitate dinamică, caracterizată prin: stare, comportament şi identitate.
Starea unui obiect semnifică temporalitatea lui. El poate fi creat şi distrus în timp, de alt obiect sau de utilizator. Starea obiectului este caracterizată de atribute care au nume asociat şi valori asociate.
Comportarea unui obiect exprimă metoda (modalitatea) de răspuns a sa la mesaje (acţiuni) provenite din exterior, care determină schimbarea stării sale.
Identitatea este proprietatea care distinge un obiect de alte obiecte. Ea se păstrează şi după ce obiectul a fost supus unor transformări sau şi-a modificat starea iniţială.
Obiectele cu stări şi comportări asemănătoare se grupează în *clase*.
O *clasă* abstractizează proprietăţile comune ale unui set de obiecte, asfel că fiecare obiect este o **instanţă** (reprezentant sau individ) a unei clase. O clasă poate avea nici unul, unul sau mai multe obiecte instanţă. Clasa este o entitate *statică* şi nu dinamică precum obiectul, deoarece după specificare şi implementare se modifică foarte greu.
Relaţiile stabilesc conexiunile logice dintre diferite entităţi.
Relaţiile obiect-obiect sunt:

- de utilizare-între obiecte ce schimbă mesaje între ele
- de conţinere-obiecte complexe care conţin alte obiecte

Relaţia obiect –clasă este cea de instanţiere.
Relaţiile clasă –clasă pot fi :

- moştenite
- de asociere în utilizări
- de conţinere (agregare)

Paradigmele POO sunt:

- abstractizarea
- încapsularea
- moştenirea

https://biblioteca-digitala.ro / https://unibuc.ro

- polimorfismul

Abstractizarea reprezintă procesul de ignorare intenționată a detaliilor nesemnificative şi de reținere a proprietăților definitorii ale unei entități. Ea poate fi: procedurală şi a datelor

Abstractizarea procedurală ignoră detaliile privitoare la desfăşurarea proceselor. Este specifică programării procedurale în care datele de I/E şi funcțiile sunt distincte şi folosite separat (exemplu C++ permite programare procedurală).

Abstractizarea datelor ignoră detalii privind reprezentarea unui tip de date .

În POO pentru entități de nivel înalt cele două abstractizări sunt combinate în cadrul claselor.

Incapsularea (ascunderea informațiilor) implementează abstracțiile prin clase. Ea separă informațiile de manipulare a unei entități de informațiile de implementare.

Incapsularea informației la nivel de tip se face prin împărțirea unei clase în două secțiuni distincte: interfață şi implementare.

Interfața unei clase, transmisa tuturor obiectelor instanță ale clasei respective, permite accesul la clasă din exterior, al potențialilor utilizatori.

Implementarea clasei, care cuprinde reprezentarea abstracției şi algoritmii metodelor de interfață, nu este cunoscută şi accesibilă utilizatorilor.

Moştenirea permite clasificarea tipurilor prin organizarea lor în *ierarhii.* În funcție de poziția ocupată în ierarhie, un tip poate fi: generalizarea altor tipuri, sau specificarea altor tipuri. Astfel, o clasă este *clasă de bază* sau *superclasă* sau *supertip* pentru toate clasele care o specializează şi *clasa derivată* sau *subclasa* sau *subtip* pentru toate clasele pe care le specializează. O clasă derivată preia de la clasa de bază toate caracteristicile acesteia, la care se mai pot adăuga şi altele noi.

Polimorfismul permite unei valori de a avea mai multe tipuri. În C + + polimorfismul este: parametric, ad-hoc şi de moştenire

Polimorfismul parametric apare când aplicăm o funcție pe argumente de tipuri diferite.

Polimorfismul ad-hoc permite supraîncărcarea funcțiilor (implementări diferite), şi redefinirea metodelor unei clase de bază prin redefinirea şi supraîncărcarea lor în clase derivate.

https://biblioteca-digitala.ro / https://unibuc.ro

Polimorfismul de moştenire permite manipularea obiectelor de un anumit tip în situaţii ce necesită tipuri diferite de cele ale obiectelor.

2.2 Declararea claselor [12, 47]

Clasa este un tip de date abstracte definit de utilizator, care se comportă ca un tip predefinit. O clasă C + + încapsulează **datele membre**, ce reprezintă atributele clasei precum şi **funcţii membre** (sau **metodele**) care efectuează operaţii asupra datelor clasei.
Clasa are un *nume* ce poate fi orice identificator, unic în domeniul respectiv de existenţă. Declararea unei clase foloseşte cuvântul cheie *class* şi are sintaxa:

```
class nume_clasa
{
[private:
      <date membre private>
      <constructori privati>
      <funcţii membre private>]
[protected:
       <date membre protejate>
       <constructori protejati>
       <funcţii membre protejate>]
 [public:
        <date membre publice>
        <constructori publici>
        <destructor public>
        <funcţii membre publice>]
   };
```

C + + asigură o limitare a accesului la membrii unei clase şi de aceea în declararea unei clase folosim *specificatori de control ai accesului*: private, protected şi public.
Specificatorul private precizează membrii proprii ai clasei. Ei pot fi accesaţi de orice funcţie, membră a clasei. Instanţele clasei nu au acces la membrii privaţi.
Specificatorul protected precizează membrii clasei care pot fi accesaţi nu numai de funcţii membre ale clasei, ci şi de funcţii membre ale claselor derivate din clasa respectivă.

https://biblioteca-digitala.ro / https://unibuc.ro

Specificatorul public precizează membrii publici care pot fi accesaţi de orice funcţie din domeniul declaraţiei clasei(funcţii membre şi instanţe).
Implicit membrii unei clase au atributul de acces **privat**. Dacă sunt prezenţi, specificatorii de acces pot fi în orice ordine şi se pot chiar repeta.
O funcţie membră a unei clase are acces la toate datele membre ale oricărui obiect din clasa respectivă, indiferent de specificatorul de acces. Se recomandă plasarea datelor membre în secţiunea protected pentru a facilita accesul la ele de către toate funcţiile membre ale claselor derivate.
În declaraţia unei clase pot apare numai prototipurile funcţiilor membre. Definirea lor poate fi inclusă în acelaşi fişier sau în fişiere separate din cadrul bibliotecilor de programe. Funcţiile definite în cadrul clasei fac implicit parte din categoria **inline**.
Pentru a defini o funcţie membră aflată în afara declaraţiei clasei de care aparţine, folosim operatorul de rezoluţie :: conform sintaxei:

nume_clasa :: nume_funcţie

unde nume_clasa este numele clasei din care face parte funcţia.
Operatorul de rezoluţie :: spune ca funcţia are acelaşi domeniu cu declararea clasei şi este membră a clasei.
După declararea unei clase, numele ei poate fi folosit ca identificator de tip pentru a declara instanţe ale clasei, cu o sintaxă asemănătoare cu cea de la declararea variabilelor.
Exemplu. Clasa **coordonate** încapsulează două date membre x,y şi doua funcţii membre (translatare şi tipărire).Cele două date membre sunt coordonatele unui punct din plan. Funcţia *translatare* ( ) translatează coordonatele punctului considerat cu a respectiv b, iar funcţia *tipărire*() afişează coordonatele punctului translatat.

```
#include<iostream.h>
class coordonate
{
private:
        float x,y;
public :
        void translatare (float a, float b);
        void tiparire ();
};
void coordonate :: translatare(float a, float b)
{
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
x + = a;
y + = b;
}
void coordonate :: tiparire ()
{
cout << "Coordonatele punctului translatat sunt: x  = " << x << "
şi y = " << y << endl ;
}
```

coordonate M , N ; ⟵ aici se declară două obiecte de tip coordonate.

Definirea funcțiilor membre translatare () şi tipărire () este făcută în afara declarării clasei *coordonate* căreia îi aparțin.

Putem opera cu obiectele dintr-o clasă ca şi cu orice tip predefinit de date.

Exemplu.

```
coordonate pct1, pct2;   ⟵   declară două obiecte
coordonate *p;           ⟵   declară un pointer la obiect
coordonate sir [10] ;    ⟵   declară un şir de 10 obiecte
pct1 = pct2 ;
*p =sir ;                ⟵   atribuiri
```

Putem accesa orice membru public prin operatorul ".". Membrii privați nu pot fi accesați decât de funcții membre ale clasei. Putem atribui o valoare unei date membre private numai printr-o funcție membră a clasei, care poate testa validitatea respectivei valori.

Exemplu [12]. Program ce declară clasa *angajat* care include două funcții membre publice şi 4 date membre private. Funcția *valori ()* inițializează datele private ale clasei şi foloseşte instrucțiunea *if* pentru a asigura datei membre *salar* o valoare corectă. Funcția *afisare ()* , afişează datele membre private. Funcțiile sunt accesate prin operatorul ".''.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class angajat
{
public:
        int valori (char*, int, long, float);
        void afisare (void) ;
private :
        char nume [50] ;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                int varsta ;
                long marca ;
                float salar ;
        } ;
        int angajat :: valori (char*ang_nume, int ang_varsta, long
        ang_marca, float ang_salar)
        {
        strcpy(nume,ang_nume);
        varsta = ang_varsta;
        marca = ang_marca;
        if ( ang_salar < 3000000)
                        {salar =ang_salar;
                        return (0);
                        }
        else return (1);
        }
        void angajat :: afisare (void)
        {
        cout <<"Nume:" <<nume<<endl;
        cout << ''Varsta : ''<<varsta<<endl ;
        cout << ''Marca :''<<marca<<endl ;
        cout <<''Salariu :''<<salar<<endl ;
        }
        void main (void)
        {
        clrsc() ;
        angajat lucrator ;
        if (lucrator.valori("Popescu '',55,1313,2000000) = = 0)
        {
        cout <<''Date angajat :''<<endl ;
        lucrator.afisare () ;
        }else cout <<''Valoarea salariului este incorecta'' ;
        }
```

După compilare şi rulare apare pe ecran :

```
        Date angajat :
        Nume :Popescu
        Varsta :55
        Marca :1313
```

https://biblioteca-digitala.ro / https://unibuc.ro

Salariu :2000000

Observaţie. Este posibil ca anumite clase să conţină funcţii membre private. Ele nu pot fi apelate de program prin operatorul ''.'' , ci doar de alţi membri ai claselor de care aparţin.

2.3 Funcţii membre inline [12, 58, 47]

Definirea unei funcţii membră în interiorul unei clase este considerată inline.

Exemplu. Aici se declară clasa *coordonate* de mai sus în care funcţia *translatare* nu este doar declarată ci şi definită ; ea devine implicit funcţie *inline*. Ca urmare, la fiecare apel, ea nu este doar apelată ci şi expandată. Definirea unei funcţii într-o clasă se numeşte şi *declarare inline implicită*. Funcţia membră *tipărire* este doar declarată în cadrul clasei. Definirea ei în afara declaraţiei clasei este prefixată de cuvântul cheie *inline*, ceea ce este echivalent cu definirea funcţiei în cadrul declaraţiei clasei a cărei membră este. Aceasta este o *decalaraţie inline explicită.*

```
#include <iostream.h>
class coordonate
{
private :
float x,y;
public :
void translatare (float a, float b)
        {
        x  + = a;
        y + = b;
        }
void tiparire () ;
} ;
inline void coordonate :: tiparire ()
{
cout <<'' Coordonatele punctului translatat sunt : x = "<<x<< "
şi y = "<<y<< endl ;
}
```

Obesrvaţie : Este interzisă folosirea în funcţiile **inline** a instrucţiunilor repetitive for, while şi do-while.

https://biblioteca-digitala.ro / https://unibuc.ro

Deoarece compilatorul are nevoie de întreaga definiţie a unei funcţii **inline**, funcţia nu poate fi folosită decât în fişierul în care a fost definită.

2.4 Constructori [43, 58]

Constructorul este o funcţie membră specială a unei clase, care are acelaşi nume cu clasa căreia îi aparţine. El serveşte la crearea şi iniţializarea obiectelor şi de aceea este apelat implicit de fiecare dată când se crează noi instanţieri ale clasei.

2.4.1 Declarare şi caracteristici

Intr-o clasă pot exista nici unul, unul sau mai mulţi constructori. Dacă nu este nici un constructor, compilatorul generează automat unul care este public, fără parametri şi cu o listă vidă de instrucţiuni.
O funcţie constructor nu întoarce un tip de date şi deci în corpul ei nu trebuie folosit *return*. Se declară un constructor prin :

```
class nume_clasa
{
public:
        nume_clasa ();                          constructor prestabilit, fără parametri
        nume_clasa (nume_clasa & c);            constructor de copiere
        nume_clasa ( parametri);                constructor cu parametri
} ;
```

La declararea şi la definirea unei funcţii constructor tipul void nu poate fi specificat ca tip returnat.
Constructorii pot avea parametri (inclusiv din cei impliciţi) şi pot fi supradefiniţi. Pointerii către constructori nu se pot folosi şi deci adresa lor nu ne este accesibilă.
Intr-un constructor putem folosi operatorii **new** şi **delete**.
Un constructor prestabilit nu are parametri sau are o listă de parametri în care toţi parametrii folosesc argumente prestabilite.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplul 1.

```
clasa coordonate
{
private:
        float x,y;
public:
coordonate( );
coordonate(coordonate&c);
coordonate(float a, float b);
..... alte funcţii.....
};
```

Exemplul 2 [12]. Program care crează clasa **angajat** ce conţine funcţia constructor *angajat*. Aceasta se defineşte ca orice funcţie membră a clasei şi atribuie valori iniţiale la trei obiecte lucr1, lucr2 şi lucr3 fără a întoarce vreo valoare.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class angajat
{
public:  declararea functiei constructor „angajat"
        angajat(char*, int, long, float);
void afisare (void);
private :
        char nume [50];
        int varsta;
        long marca;
        float salar;
};      definirea functiei constructor „angajat"
angajat :: angajat(char*nume, int varsta, long marca, float salar)
{
strcpy (angajat::nume,nume);
angajat :: varsta = varsta;
angajat :: marca = marca;
if (salar <3200000.0)
        angajat :: salar =salar;
else
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
angajat :: salar = 0.0;

}

                    definirea functiei membru „afisare"
void angajat :: afisare (void)
{
cout << endl;
cout << "Nume:" << nume <<endl;
cout << "Varsta:" << varsta <<endl;
cout << "Marca" << marca <<endl;
cout << "Salariu:" <<salar <<endl;
}
void  main  (void )
{
clrscr( );
angajat lucr1 ("M. Pop",50, 1313, 2000000.0);
angajat lucr2 ("A. Pop",17, 1713, 3000000.0);
angajat lucr3 ("Marian P.", 43, 1317, 3200000.0);
cout <<"Date angajati:" << endl;
lucr1.afisare( );
lucr2.afisare( );
lucr3.afisare( );
}
```

După compilare şi rulare pe ecran se vor afişa datele:

```
Date angajati:
Nume: M.Pop
Varsta: 50
Marca: 1313
Salariu: 2000000.0
Nume: A. Pop
Varsta: 17
Marca:1713
Salariu: 3000000.0
Nume : Marian P.
Varsta: 43
Marca 1317
Salariu 0.0
```

https://biblioteca-digitala.ro / https://unibuc.ro

Dacă nu precizăm valori pentru toţi parametrii, funcţia constructor ca orice funcţie foloseste valori implicite.

2.4.2 Supradefinirea constructorilor

O funcţie constructor se supradefineşte ca orice altă funcţie prin specificarea unor funcţii alternative care corespund unor tipuri diferite de parametri.

Spre exemplu în programul de mai sus putem da o a doua definire a funcţiei constructor **angajat** înaintea definirii funcţiei membră **afişare** care solicită utilizatorului să introducă o valoare a argumentului **salar** dacă aceasta nu este precizată în program, în timp ce prima definire implică specificarea în program a valorilor celor 4 argumente: nume, vârstă, marcă, salar.

A doua definire are forma:

```
angajat :: angajat (char* nume, int varsta, long marca)
{
strcpy (angajat :: nume,nume)
angajat :: varsta = varsta;
angajat :: marca=marca;
do{
        cout <<"Introdu un salar pentru" << nume << "mai mic
                                        decat 3200000.0";
        cin >> angajat :: salar;
        }
while (salar>= 3200000.0)
}
```

În programul principal totul este identic ca mai sus cu deosebire pentru lucr3 unde instrucţiunea este:

```
angajat lucr3 ("Marian P.", 43, 1317);
```

La compilare şi rulare se cere introducerea unei valori pentru salariul persoanei lucr3 după care execuţia continuă cu afişarea datelor pentru cei trei angajaţi.

2.4.3 Constructor de copiere

Acest constructor se foloseşte în C + + pentru a declara un obiect de tip clasă şi pentru a transfera un obiect ca parametru sau ca rezultat al unei

https://biblioteca-digitala.ro / https://unibuc.ro

funcţii însotit de o atribuire respectiv de iniţializarea obiectului cu datele altui obiect de acelaşi tip.
Dacă un astfel de constructor de copiere nu este declarat într-o anumită clasă, compilatorul generează automat unul care iniţializează datele unui nou obiect cu valorile corespunzătoare dintr-un obiect specificat prin copiere membru cu membru ca la atribuirea standard.
Pentru a declara un constructor de copiere pentru clasă trebuie să specificăm un parametru unic de tip referinţă la un obiect de acelaşi tip **nume_clasa** sub forma:
nume_clasa (const nume_clasa& nume_obiect);
unde modificatorul **const** asigură că obiectul de intrare nu poate fi modificat. El poate lipsi din sintaxa constructorului, dar se recomandă folosirea lui.
Constructorul de copiere acţionează ca orice apel de funcţie obişnuită.
Exemplu: Aici clasa dublet conţine declaraţiile unui constructor prestabilit şi a unuia de copiere care este apelat de trei ori:

```
Class dublet
{
        protected:
                int i; int j;
        public :
                dublet ();              ← constructor prestabilit
                dublet (dublet& p);     ← constructor de copiere
                .....                   ← alţi membrii

};
.....
dublet f(dublet arg)        ← apelarea constructorului de copiere pentru a
{                             iniţializa argumentul arg.
.....
dublet obiect(arg);         ←apelarea constructorului de copiere pentru a
                              iniţializa obiectul obiect
.....
return obiect;              ← apelarea constructorului de copiere pentru a
}                             iniţializa valoarea de retur
```

Un constructor de copiere poate promova mai multe argumente, dar cele suplimentare trebuie să fie cu valoare implicită pentru a permite copierea automata a obiectelor la apelul funcţiei sau la returul din funcţie.

https://biblioteca-digitala.ro / https://unibuc.ro

Constructorul de copiere generat implicit de compilator copiază doar datele membre în cazul unor structuri dinamice şi de aceea se recomandă declararea de utilizator a constructorului de copiere pentru clasele care se referă la acest tip de structură de date.

2.5 Destructori [12, 47, 58]

Destructorii sunt funcţii speciale ale unei clase cu efect complementar constructorului. Ele sunt automat apelate pentru distrugerea obiectelor unei clase la sfârşitul programului.
Destructorii au acelaşi nume cu cel al clasei de care aparţin, precedaţi însă de caracterul "~". Fiecare clasă are cel mult un destructor. Dacă acesta lipseşte, compilatorul generează automat unul cu efect nul.
Funcţiile destructor nu primesc parametri. Se declară sub forma:

```
class nume_clasa
{
public:
nume_clasa( )                 ←constructor prestabilit
. . . . . .                   ←alti constructori
~ nume_clasa( )               ←destructor
. . . . . .                   ←alte funcţii membre
};
```

Funcţia destructor nu ţntoarce nici un tip de date, nici chiar tipul void. Destructorii sunt utili mai ales pentru obiectele create prin alocarea dinamică a memoriei.

Exemplu [12]. Un program ce are clasa **angajat** în care este definit destructorul **~angajat**.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class angajat
{
public:        declararea functiei constructor
  angajat(char*, int, long, float)
  ~angajat( );      declararea functiei destructor
  void afisare (void);          ← fct. membru
private:
  char nume [50];
  int varsta;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
   long marca;
   float salar;     definirea functiei constructor
};
angajat :: angajat (char* nume, int varsta,
                        long marca, float salar)
{
strcpy(angajat :: nume,nume);
angajat :: varsta = varsta;
angajat :: marca = marca;
angajat :: salar =salar;
}
angajat :: ~angajat( )       definire functiei
 {                                destructor
cout<<"Se distruge obiectul pentru angajatul"
                                  <<nume<<endl;
}
void angajat :: afisare(void)    definirea functiei
{                                  membru
cout<<"Nume:"<<nume<<endl;
cout<<"Varsta:"<<varsta<<endl;
cout<<"Marca:"<<marca<<endl;
cout<<"Salariu"<<salar<<endl;
}
void main ( )
{
clrscr( );
angajat lucr("B.P.",90,2051,2000000.0);
cout<<"Date angajat:"<<endl;
lucr.afisare( );
}
```

```
Date angajat:
Nume: B. P.
Varsta: 90
Marca: 2051
Salariu: 2000000.0
Se distruge obiectul pentru angajatul B.P.
```

https://biblioteca-digitala.ro / https://unibuc.ro

2.6 Membri statici ai unei clase [12, 47]

Datele şi funcţiile unei clase declarate prin cuvântul cheie **static** se numesc **membri statici**.

2.6.1 Date membre statice

Fiecare obiect dintr-o clasă are o mulţime de date membre. Cele nestatice există în copii distincte în fiecare obiect ce le conţine. Cele statice există intr-o singură copie comună tuturor obiectelor ce le conţine şi deci unei **date membre statice** i se rezervă o singură zonă de memorie comună tuturor instanţelor clasei. Acestea se declară în cadrul clasei sub forma:

```
{private ;}
{public;  }
 static tip nume_data;
```

După declararea clasei orice dată membră statică trebuie declarată ca variabilă globală în exteriorul clasei sub forma:

tip_data nume_clasa::nume_data_membru;

Obligatoriu datele membre statice trebuie iniţializate în afara declaraţiei clasei.

Datele membre statice există separat de obiectele clasei, dar orice modificare a unei date membre este sesizată de toate obiectele clasei ce o conţin.

Datele membere statice pot fi accesate înainte de crearea obiectelor clasei. Accesarea lor din interiorul funcţiilor membre ale clasei respective se face la fel ca şi pentru datele membre nestatistice.

Exemplu [12]. Program ce are clasa **stoc** în care membrul **preţ** este declarat static. El este acelaşi pentru cele două obiecte ale clasei **prim** şi **secund**. Modificarea valorii membrului **preţ** afectează instanţele celor două obiecte. Valorile celorlalte date membre ale obiectelor nu se modifică.

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class stoc
{
private:  declararea membrului static pret
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        static int pret;
        char denumire[50];
        char producator[50];
        int cantitate;
public:
        stoc(char*, char*, int);
        void afisare ( );
void val (int);
};      ↙ declararea membrului static pret ca
int stoc :: pret;              variabila globala
void stoc :: val (int prt)
{
pret = prt;            ← initializarea membrului static
}
stoc :: stoc (char* den, char* prod,  int cant);
{
strcpy (stoc :: denumire, den);
strcpy (stoc :: producator, prod);
stoc :: cantitate = cant;
}
void stoc :: afisare ( )
{
cout<<"Denumire:"<<denumire<<endl;
cout<<"Producator:"<<producator<<endl;
cout<<"Cantitate:"<<cantitate<<"buc"<<endl;
cout<<"Pret unitar:"<<pret<<"u.m."<<endl;
}
void main ( )
{
clrscr ( );
stoc prim ("Produs1", "Firma A", 200);
stoc secund ("Produs2", "Firma B", 300);
prim.val(100);
cout<<"Oferta initiala:"<<endl;
prim.afisare( );
secund.afisare( );
cout<<endl<<"Oferta finala:"<<endl;
prim.val(90);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
prim.afisare( );
secund.afisare( );
}
```

După rulare pe ecran apar afişate:
Oferta initiala:
Denumire: Produs1
Producator: Firma A
Cantitate: 200 buc.
Pret unitar: 100 u.m.
Denumire: Produs2
Producator: Firma B
Cantitate: 300 buc.
Pret unitar: 100 u.m.
Oferta finala:
Denumire: Produs1
Producator: Firma A
Cantitate: 200 buc.
Pret unitar: 90 u.m.
Denumire: Produs2
Producator: Firma B
Cantitate: 300 buc.
Pret unitar: 90 u.m.
Orice dată membră declarată ca **publică** şi **statică** se poate folosi în program chiar dacă în program nu există nici un obiect al clasei.

Pentru accesarea unui astfel de membru se foloseşte operatorul de rezoluţie globală sub forma: **nume_clasa :: nume_membru**

Exemplu.

```
#include<iostream.h>
#include<conio.h>
class stoc
{
public:
        static int pret;
private:
        char denumire [50];
        char producator [50];
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
int cantitate;
};
int stoc :: pret; <------ declararea membrului static pret ca
void main ( )             variabila globala
{
clrscr ( );
stoc :: pret = 100;
cout<<"Pretul unitar initial a fost:"<<stoc :: pret <<"u.m."<<endl;
stoc :: pret = 90;
cout<<"Pret unitar final este:"<<stoc :: pret<<"u.m";
}
```

Aici se accesează membrul **preţ** al clasei **stoc** fără să existe un obiect al clasei stoc.

Observaţie. Prin folosirea datelor membre statice:

- reducem numărul variabilelor globale din program;
- urmărim numărul instanţelor unei clase;
- controlăm accesul;
- alocăm un bloc de memorie special pentru diferite obiecte ale unei clase.

2.6.2 Funcţii membre statice [12, 47, 58]

Putem defini funcţii membre statice ca şi date membre statice. Ele fac operaţii care nu sunt asociate unor obiecte individuale, ci asociate întregii clase care le conţin.

O funcţie membră *statică* şi *publică* poate fi apelată şi când nu există nici un obiect al clasei respective.

Pentru apelare folosim operatorul de rezoluţie globală "::".

Aceste funcţii nu primesc implicit adresa unui obiect.

Spre deosebire de funcţiile membre nestatice ale unei clase, cele statice nu au pointerul **this** care să indice spre obiectul curent pentru care este apelată funcţia membră şi deci ele nu pot accesa direct decât *alte elemente statice* ale clasei respective (nu pot accesa elementele nestatice ale clasei!).

Se definesc ca funcţii membre statice într-o clasă acele funcţii de uz general care sunt folosite de mai multe clase, evitând definirea lor ca funcţii de sine stătătoare.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplul 1.

```
class clasa
{
private:
    static int i;
    . . . . . .
public:
    static void f(int);          declararea functiei membre statice
    . . . . . .
};
int clasa :: i=5;                definirea şi iniţializarea datei membre
statice
void clasa :: f(int i)           definirea functiei membre statice
{
i=i*i;                           accesarea directa a unui membru static
. . . . . .
}
```

Exemplul 2.

```
class cls
{
public:
    int vect [15];
    static int prim ( );
};
int cls :: prim ( )
{
return vect [0];
}
```

În acest caz compilatorul va semnala eroare la incercarea functiei membră statica **prim** de **a accesa** membrul nestatic **vect**.

2.7 Funcţii prietene [12, 58]

C + + permite accesul la datele membre private ale unei clase nu numai funcţiilor membre ale clasei respective ci şi altor funcţii nemembre clasei numite **funcţii prietene**. Ele pot fi independente sau membre altei clase.

https://biblioteca-digitala.ro / https://unibuc.ro

Declararea unei *funcţii prietene* independente se face în interiorul clasei precedând prototipul funcţiei de cuvântul cheie *friend* astfel:

```
class nume_clasa
        {
        public:
                friend tip_rez  nume_fct (parametri);
         };
```

Indiferent de poziţia prototipului, funcţiile prietene sunt publice . O astfel de funcţie nu poate întoarce o referinţă la clasa prietenă.

O funcţie prietenă poate accesa membri privaţi ai clasei direct prin operatorul " .".

Exemplu [12]. Programul următor declară clasa **comp** şi în ea *funcţia prietenă independentă*: **coincid**(), care întoarce o valoare nenulă dacă punctele primite ca parametri sunt identice şi nulă în caz contrar.

```
#include <iostream.h>
#include <conio.h>
class comp
{
int x,y;
public:          declararea constructorlui
   comp (int,int);
   ~ comp ( );                ← declararea destructorului
   void afisare ( );          ← declararea functiei membră
   friend int coincid (comp&, comp&);
};  declararea functiei prietene independente
comp::comp(int a,int b)       ← definirea constructorului
{x=a; y=b;
afisare( );
.....
}
comp::~comp( )                ← definirea destructorului
{.....}
void comp::afisare ( )        ← definirea functiei membră
{
cout <<"x="<<x<<";y="<<y<<endl;
}              definirea functiei prietene
int coincid (comp& p1,comp& p2)
{
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
if (p1.x == p2.x && p1.y == p2.y)
        return 1;
        else  return 0;
}
void main ( )
{
clrscr ( );
cout <<"Coordonatele punctului p1 sunt:";
comp p1(2,3);
cout <<"Coordonatele punctului p2 sunt:";
comp p2(2,4);
cout <<"Rezultat:"<<endl;
if (coincid(p1,p2))
   cout <<"Punctele p1 şi p2 coincid:"<<endl;
   else cout<<"Punctele p1 şi p2 sunt diferite";
}
```

După rulare pe ecran apare:

Coordonatele punctului p1 sunt: x=2; y=3
Coordonatele punctului p2 sunt: x=2; y=4

Rezultat:

Punctele p_1 şi p_2 sunt diferite.

O funcție **prietenă** poate opera asupra unui obiect transferat ca parametru. Ea foloseşte numele obiectului şi nu operatorul de rezoluție pentru a referi date sau funcții membre ale clasei.

Dacă un program conține o funcție membră unei clase care este prietenă altei clase se impun conditiițe:

- declararea corespunzatoare a funcțiilor prietene
- anumită ordine în definiția claselor.

Declararea funcțiilor prietene ce aparțin unei clase se face sub forma:

```
class nume_cls2
{
public:
. . . . . .
friend nume_cls1 :: nume_fct (parametri);
}
```

unde nume_cls1= numele clasei în care este funcția
nume_cls2 = numele clasei cu care funcția este prietena.

https://biblioteca-digitala.ro / https://unibuc.ro

Definiţiile celor două clase trebuie să respecte o ordine prestabilită. Pentru aceasta trebuie specificat un identificator **nume_cls1** sub forma: **class nume_cls1;**. Se defineşte apoi clasa **nume_cls2**. Cu acestea se indică compilatorului o revenire ulterioară la definirea clasei **nume clas1**. Altfel definiţia clasei **nume_cls2** ar face referire la clasa **nume_cls1**, necunoscută pentru compilator şi aceasta ar da eroare.

Exemplu.

Aici funcţia f( ) ∈ clasei A şi este prietena clasei B

```
class A;          definirea incompleta a clasei A
class B           definirea clasei B
{
. . . . . .
void f(A, . . . . )
. . . . . .
};
class A
{          ←definirea clasei A
. . . . . .
friend void B :: f(A, . . . . );
};
void B :: f(A obiect-A, . . . . )
{. . . . . .          definirea functiei prietene
}
```

Folosirea funcţiilor prietene:

- reprezintă o metodă de control a accesului la membrii unei clase
- oferă o metodă simplă de acces limitat din anumite funcţii la datele unei clase
- funcţia care în lista de parametri are cel puţin doi parametrii din aceeaşi clasă şi care accesează parametri privaţi nu poate fi decât prietenă.

2.8 Clase prietene [12, 47, 46]

Clasa B este prietena clasei A dacă toate funcţiile membre clasei B pot accesa direct membri privaţi ai clasei A.

Pentru a defini clasa B prietena clasei A se introduce între membri publici ai clasei A o instrucţiune care să conţină cuvântul **friend** urmat de numele clasei B sub forma:

https://biblioteca-digitala.ro / https://unibuc.ro

```
class B;
class A
  {
      public:
      . . . . . .
      friend class B;

  };
class B
  {. . . . . };
```

Relația **friend** nu este comutativă şi nici tranzitivă.

De regulă clasele prietene în C + + sunt independente, în sensul că nici o clasă nu moşteneste membrii celeilalte clase.

Exemplu [12]. Aici clasa *specificare* este prietena clasei *material.* Programul modifică una din caracteristicile unui material şi anume funcția *modif( )* a clasei *specificare* modifică, cantitatea disponibilă a materialului. Obiectul *material* este transmis funcției *modif( )* prin adresă. Pentru a accesa membrul respectiv funcția foloseşte un pointer.

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class specificare;
class material
{
public:
  material(char*, char*, char*);
  void afisare(void);
 friend specificare;
private:
  char den [30];
  char prod [30];
  char cant [20];
};
material ::material (char* den, char* prod, char*cant)
{
strcpy(material :: den, den);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
strcpy(material :: prod, prod);
strcpy(material :: cant, cant);
}
void material :: afisare(void)
{
cout<<"Denumire:"<<den<<endl;
cout<<"Producator:"<<prod<<endl;
cout<<"Cantitate:"<<cant<<endl;
}
class specificare
{
public:
  void modif(material*, char*);
  char* atrib(material);
};
void specificare ::modif(material*  material1, char* cant1)
{
strcpy(material1 → cant, cant1);
}
char* specificare :: atrib(material material1)
{
static char cant [20];
strcpy (cant, material1.cant);
return (cant);
}
void main ( )
{
clrscr( );
cout<<"Situatia initiala:"<<endl;
material car("Inox","Cos Pitesti","5t");
specificare actual;
car.afisare ( );
actual.modif(&car,"3t");
cout<<endl<<"Situatia actuala:"<<endl;
car. afisare ( );
}
```

După compilare şi rulare pe ecran apare:

```
Situatia initiala:
```

https://biblioteca-digitala.ro / https://unibuc.ro

Denumire: Inox
Producator: Cos Pitesti
Cantitate: 5t
Situatia actuala:
Denumire: Inox
Producator: Cos Pitesti
Cantitate: 3t

2.9 Supradefinirea operatorilor [12, 46]

Definirea unei clase într-un program reprezintă definirea unui nou tip de date. Folosind operatori standard putem defini diferite operaţii cu obiectele clasei. Aceste operaţii se obţin prin **supradefinirea** operatorilor standard adică prin modificarea semnificaţiei lor.
Rolul supradefinirii operatorilor este creşterea lizibilităţii programului exprimând operaţii cu obiectele claselor definite de utilizator sub o formă cât mai sugestivă.
Supradefinirea unui operator într-o clasă se face prin specificarea unei funcţii care implementează operaţia corespunzătoare operatorului respectiv. Această funcţie este ulterior apelată atunci când clasa foloseşte operatorul supradefinit.
Declararea funcţiei foloseşte cuvântul cheie *operator* sub forma:

tip_rez operator simbol_operator (parametri);

La supradefinirea operatorilor trebuie respectate regulile:

- simbol_operator este orice simbol de operator din C + + mai puţin următorii patru:
  . (operator de membru al clasei)
  * (operator de pointer la membru)
  :: (operator de rezoluţie globală)
  ?: (operator de expresie condiţională)
- nu se poate extinde setul de operatori C + + prin desemnarea de noi simboluri (caractere sau seturi de caractere).
- aritatea unui operator nu se poate modifica.
- prioritatea şi asociativitatea operatorilor supradefiniţi se păstrează; se poate modifica doar prin folosirea parantezelor.

La supradefinirea unui operator pentru o clasă, semnificaţia lui este modificată doar în clasa respectivă, pentru restul variabilelor din program, operatorul păstrându-şi semnificaţia cunoscută.

https://biblioteca-digitala.ro / https://unibuc.ro

Funcţiile operator care supradefinesc operatorii unei clase se definesc:
- sau ca funcţii prietene
- sau ca funcţii membre ale clasei respective.

2.10 Funcţii operator prietene [12, 46, 47]

Prototipul unei funcţii operator prietene apare în cadrul declaraţiei clasei fiind precedat de cuvântul cheie *friend*. Această funcţie are acces la toate datele private ale clasei.

Exemplu [12]. Aici implementăm operatorul unar minus, operatorul binar plus şi înmulţirea cu un scalar întreg, ca funcţii prietene clasei **coord**, care se referă la coordonatele punctelor din plan.

Prototipurile celor trei funcţii operator sunt incluse în declaraţia clasei iar definiţiile lor sunt plasate în afara clasei.

```
#include <iostream.h>
#include <conio.h>
class coord
{
private:
  double x; double y;
public:                    constructor
  coord (double x0=0, double y0=0)
  {x=x0; y=y0;}
  void afisare ( )  ←  fct. membru
    {
    cout <<"x="<<x;
    cout <<", y="<<y<<endl;
    }  declararea funcţiilor operator
friend coord operator –(coord);
friend coord operator +(coord, coord);
friend coord operator *(int, coord);
};
coord operator -(coord a)
{                          def. fct.
coord c;                   operator minus
c.x = -a.x;
c.y = -a.y;
return c;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
}
coord operator +(coord a, coord b)
{                          def. fct.
coord c;            operator plus
c.x = a.x + b.x;
c.y = a.y + b.y;
return c;
}
coord operator *(int k, coord a)
{                          def. fct. operator
coord c;            inmultire cu scalar întreg
c.x = k * a.x ;
c.y = k * a.y ;
return c;
}
void main ( )
{
clrscr ( );
cout <<"Date de intrare:"<<endl;
coord p1(1,1); p1.afisare( );
coord p2(2,4); p2.afisare( );
coord p3(3,0); p3.afisare( );
coord r1, r2, r3, r4;
cout <<"Rezultate:"<<endl;
r1 =- p1;
r1.afisare ( );
r2 = p1 + p2;
r2.afisare ( );
r3 = p1 + p3 +r2;
r3.afisare ( );
r4 = - 5 * p1;
r4.afisare ( );
}
```

După compilare şi rulare pe ecran apar informaţiile:
Date de intrare:
x = 1, y = 1
x = 2, y = 4
x = 3, y = 0

https://biblioteca-digitala.ro / https://unibuc.ro

Rezultate:
x = - 1, y = - 1
x = 3, y = 5
x = 7, y = 6
x = -5, y = - 5

2.11 Funcţii operator membre [46, 47]

O funcţie operator membră unei clase are unul din operanzi **this** (adică pointer implicit care indică spre obiectul curent pentru care este apelată funcţia) iar ceilalţi operanzi, dacă există sunt transmişi ca parametri. Astfel implicit funcţia primeşte adresa obiectului pentru care este apelată.

Exemplu. Reluăm exemplul de mai sus pentru implementarea celor trei operatori ca funcţii operator membre ale clasei **coord**. Programul este acelaşi cu următoarele modificări:

- Se înlocuie cele trei instrucţiuni **friend** care declară funcţiile operator prietene respectiv cu urmatoarele instrucţiuni care declară funcţiile operator membre:
  coord operator –( );
  coord operator +(coord);
  coord operator *(int);
- În definiţiile celor trei funcţii se înlocuie instrucţiunile subliniate mai sus respectiv cu câte una din următoarele:
  coord coord :: operator –( )
  coord coord :: operator +(coord b)
  coord coord :: operator *(int k)
- În rest totul este nemodificat, excepţie referirea la primul argument al funcţiilor care acum se face prin referirea directă a acestora aşa cum ar fi spre exemplu c•x=x+b•x în loc de c·x=a·x+b·x . Astfel programul va afişa aceleaşi rezultate ca şi cel de mai sus.

2.12 Directive condiţionale de preprocesare [12]

Preprocesarea este prima etapă a compilării unui fişier sursă şi este realizată de un program special numit **preprocesor** care este diferit de compilator. El este automat apelat de compilator pentru a executa anumite

https://biblioteca-digitala.ro / https://unibuc.ro

comenzi ce există în fişierul sursă, comenzi numite **directive de preprocesare** şi precedate de #.

Pe lângă directivele simple de preprocesare, prezentate la paragraful 1.2 al acestei părţi a cărţii există în C++ şi directive condiţionale.

Acestea sunt #if, #else, #elif, #endif, #ifdef, #ifndef, în care primele 4 directive controlează compilarea unor instrucţiuni în anumite condiţii şi au sintaxele:

```
#if cond1
        secv1
[#elif cond2
        secv2
        . . . . . .
[#elif condn
secvn]
        . . . . . ]
[#else
        secv0]
#endif
```

unde:

- **secv$_i$** este o secvenţă de directive de compilare care se execută când cond$_i$ este îndeplinită , i=1...n,
- **secv$_0$** este secvenţă de directive de compilare care se execută când nici una din condiţiile cond$_1$, ... , cond$_n$ nu este îndeplinită.

Exemplu.

```
#if VAR = = 1
        # define ALPHA char
#elif VAR = = 2
        #define BETA double
#else
        #define GAMA float
#endif
```

Directiva #ifdef compilează o linie de cod dacă un anumit identificator a fost definit anterior iar **#ifndef** dacă identificatorul nu a fost definit anterior.

Acestea au sintaxele:

```
#ifdef identificator
```

https://biblioteca-digitala.ro / https://unibuc.ro

$secv_1$
#endif
#ifndef identificator
$secv_2$
#endif

unde

- $secv_1$ = secvență de directive de compilare care se execută dacă **identificator** a fost definit anterior.
- $secv_2$ = secvență de directive de compilare care se execută dacă **identificator** nu a fost definit anterior.

Exemplu.
```
#ifdef MAXIM
        #define ALPHA char
#endif
#ifndef MAXIM
        #define BETA double
#endif
```

Directiva #error

Generează un mesaj de eroare. Are sintaxa:

#error text_mesaj_eroare.

Aceasta semnalează incompatibilitatea versiunilor unui program sau erori de sintaxă cauzate de omiterea sau folosirea incorectă a anumitor caractere ca: paranteze, acolade, punct şi virgulă etc.

Exemplu.
```
Error directive: Cannot find command procesor
Error syntax.CPP4: Undefined symbol "quotes" în
                                function main ( )
```

Ultima eroare de sintaxă se referă la linia a 4-a a fişierului sursă şi indică neincluderea unui mesaj între ghilimele.

Directiva #line

Aceasta permite precizarea *numărului liniei curente* compilate într-un program. Are sintaxa:

#line numar ["nume_fisier"]

Directiva specifică numărul liniei din fişierul cod sursă original. Cu ea putem realiza referinţe încrucişate sau identifica poziţia din codul sursă unde s-au produs erori.

https://biblioteca-digitala.ro / https://unibuc.ro

2.13 Complemente [46, 47]

Secțiunea prezentă completează și exemplifică mai bine o serie de noțiuni descrise înaintea altor elemente precum: pointeri, referințe, constructori, destructori, metode, etc și care ar fi necesitat cunoașterea acestora din urmă.

2.13.1 . Insertor și extractor

Operatorul de extragere (citire) >> se mai numește și **extractor**, iar cel de inserare (scriere)<< se numește și **insertor.** Ei returnează o referință către **stream**, care permite testarea stării streamului sub forma:

```
#include<iostream.h>
void main()
{ int n;
if (cin>>n) cout <<"citire cu succes";
else cout<<"eșec la citire";}
```

2.13.2 Stream de intrare

Tipurile de date predefinite aceeptate implicit de un stream de intrare sunt:

- char
- short
- int
- long
- float
- double

} cu sau fără unsigned

- șir de caractere

Dacă primul caracter citit în streamul de intrare, diferit de spațiu, este incorect pentru tipul citit, valoarea variabilei destinație rămâne nemodificată și se va seta un indicator "**fail**" al streamului de intrare, după care din streamul de intrare nu se va mai putea citi până la resetarea (anularea) indicatorului prin funcția clear( ) sub forma: **cin.clear( );**

2.13.3 Stream de ieșire

Tipurile de date predefinite acceptate implicit de un stream de ieșire sunt:

https://biblioteca-digitala.ro / https://unibuc.ro

char
short
int
long
float
double
} cu sau fără unsigned
long double
poineri la aceste tipuri
void*

Pointerul la caracter (char*) este folosit pentru tipărirea şirurilor de caractere.

Pointerul void* se foloseşte la tipărirea variabilelor de tip pointer

Exemple.

1.
```
#include<iostream.h>
void main( )
{char s[20];
cin>>s;
cout<<s;}
```

2.
```
#include<iostream.h>
void main( )
{char*x="abcde";
cout<<"şirul x= "<<x<<endl;
cout<<"Pointer x="<<(void*)x;}
```

În primul exemplu, numele s al tabloului are ca valoare adresa primului său element şi astfel s este de fapt un pointer constant către şirul de caractere citit in s.

2.13.4 Funcţii la nivel de caracter asociate unui stream de intrare şi de ieşire

Funcţiile uzuale la nivel de caracter asociate unui stream sunt:

- ***stream& put(char c);*** – inserează un caracter în şirul de ieşire şi întoarce streamul de ieşire.

Exemplu de apel: cout.put(x);

- ***int get(void);*** – extrage următorul caracter din streamul de intrare şi îl returnează; dacă se ajunge la sfârşitul şirului de intrare, returnează EOF, adică –1.

Exemplu de apel: ***x=cin.get( );***

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu complet care copiază caracterele de la intrare până ce întâlneşte ^Z, la ieşire:

```
#include<iostream.h>
void main( )
{char c;
while ((c=cin.get( ))!=EOF) cout.put(c);}
```

- istream& get(unsigned char&);
- istream& get(signed char&);

Ambele extrag următorul caracter din streamul de intrare în variabila dată ca argument şi întorc streamul de intrare.

Exemplu de apel. ***cin.get(x);***

- ***istream& get (unsigned char*c,int n, char='\n');***
- ***istream& get (signed char*c,int n, char='\n');***

Ambele forme extrag un şir de caractere din şirul de intrare în variabila tablou "c". Numărul maxim de caractere al şirului este dat de al doilea argument (se extrag n-1 caractere). Al treilea argument precizeaza caracterul de sfârşit, până la care se citeşte, fără a fi citit şi el.

Exemple de apel. ***cin.get(c,10);*** se extrag în c[10], din sirul de intrare cel mult 9 caractere sau până la întâlnirea caracterului implicit de sfârşit "\n".

cin.get(c,10,','); se extrag 9 caractere sau până ce se întâlneşte ',', care nu va mai fi extras din şir.

- ***istream& getline (unsigned*c, int n, char='\n');***
- ***istream& getline (signed*c, int n, char='\n');***

ca si get extrag şi depun în c eventual şi delimitatorul, dacă acesta este întâlnit înainte de a citi toate cele n-1 caractere ale şirului.

Exemple de apel. ***cin.getline(c,5);*** extrage 4 caractere sau până la caracterul '\n' care se extrage si el din stream şi se depune în c.

cin.getline(c,10,'.'); extrage 9 caractere sau extrage din şirul de intrare până ce întâlneşte caracterul '.' pe care îl extrage şi depune în c.

- ***istream& ignore (int n=1, int=EOF);***

Neglijează un număr maxim de caractere dat de primul argument ce are valoarea implicită 1 sau până la întâlnirea unui delimitator dat de al doilea argument ce are valoarea implicită –1. Functia întoarce o referință către streamul de intrare.

Exemple de apel. ***cin.ignore(80,'#');*** ignoră următoarele 80 de caractere sau până întâlneşte caracterul "#"

cin.ignore( ); ignoră următorul caracter.

https://biblioteca-digitala.ro / https://unibuc.ro

- ***int gcount( );*** întoarce numărul de caractere citite din stream la ultima operație de citire.

Exemplu de apel.

```
#include<iostream.h>
void main( )
{int x; char y[25];
cin>>x>>y;
cout<<cin.gcount( );}
```

Dacă tastăm 1234 abc<enter> se va obține rezultatul 3

- ***int peek( );*** întoarce următorul caracter fără să îl extragă din stream.
- **istream& putback(char c);** inserează caracterul c în stream ca prim caracter de citit.

Exemplu de apel. ***cin.putback('x')***; inserează 'x' în stream pe prima poziție ce va fi citită.

- ***istream& read(char*c,int n);***

Citește n caractere din streamul de intrare în tabloul c, fără ca citirea să fie întreruptă de vre-un caracter de delimitare.

Exemplu.

```
#include<iostream.h>
void main( )
{ char c[10];
cin.read(c,5); cout<<c<<endl;}
```

Programul citește 5 caractere din streamul de intrare la care adaugă caracterul '\0'

- ***ostream& write(char*c, int n);***

Inserează n caractere din șirul c în streamul de ieșire asociat.

2.13.5 Funcții de gestiune a ecranului și tastaturii

Biblioteca standard a limbajului C++ conține funcții pentru gestiunea ecranului în mod text sau în mod grafic. Funcțiile standard de gestiune a ecranului in mod text au prototipurile în fișierul antet **conio.h**.

În mod normal, ecranul este compus din 25 de linii și 80 de coloane. Colțul stânga sus al ecranului are coordonatele (1,1), iar cel din dreapta-jos are coordonatele (80,25).

Funcțiile uzuale pentru gestiunea ecranului sunt:

https://biblioteca-digitala.ro / https://unibuc.ro

- ***void window(int x1, int y1, int x2, int y2);*** defineşte o fereastră pe ecran de varf stânga-sus (x1,y1) şi de varf dreapta-jos (x2,y2).
- ***void clrscr(void);*** şterge fereastra activă şi duce cursorul în colţul stânga-sus al acesteia, adică în coordonatele (1,1).
- ***int wherex(void);*** dă numărul coloanei pe care se află cursorul.
- ***int wherey(void);*** dă numărul liniei pe care se află cursorul.
- ***void gotoxy(int x, int y);*** duce cursorul în punctul de coordonate (x,y) relative la fereastra activă .

În fişierul antet conio.h avem şi funcţii de lucru cu tastatura printre care:

- ***int getche( );***
- ***int getch( );*** - Acestea citesc de la intrarea standard caracterul curent şi întorc codul ASCII al caracterului citit. Ele au acces direct la caracter imediat după ce a fost tastat. Prima funcţie citeşte cu ecou, iar a doua fără ecou (fără a afişa caracterul pe ecran).
- ***int kbhit(void);*** întoarce o valoare diferită de 0 dacă a fost acţionată o tastă.

2.13.6 Vizibilitatea şi durata de viaţă a variabilelor [47]

Există două nivele posibile la care pot fi declarate variabilele.

a. **nivel intern** - variabilele se declară în interiorul blocurilor de program. Ele se numesc **variabile locale** sau **temporare**.

b. **nivel extern** – variabilele se declară în afara oricărui bloc al programului. Ele se numesc variabile globale sau permanente şi sunt accesibile din orice funcţie a fişierului sursă.

Toate funcţiile sunt globale, adică accesibile programului, deoarece C++ nu permite definirea unei funcţii în interiorul altei funcţii, aşa cum se întâmplă spre exemplu în PASCAL.

Cea mai mică unitate de program în care se pot declara şi unde sunt cunoscute variabilele este **blocul**.

Variabilele globale permit un transfer al informaţiilor între funcţii în locul transferului prin parametrii asociaţi funcţiilor şi prin valoarea returnată de funcţie. Această alternativă este recomandată când mai multe funcţii trebuie să aibă acces la un număr mare de variabile. **Domeniul de**

https://biblioteca-digitala.ro / https://unibuc.ro

vizibilitate (sfera de acțiune) al unei variabile este acea porțiune din program în care variabila este cunoscută și poate fi folosită. De exemplu, variabilele declarate în interiorul unei funcții nu pot fi folosite de nici o altă funcție, ele sunt deci locale funcției în care au fost declarate.

2.13.7. Clase de memorare [47]

Un identificator în C++ are două atribute: clasa de memorare și tipul valorilor pe care le poate lua identificatorul.

Clasa de memorare a unei variabile precizează durata ei de viață, adică timpul în care variabila este păstrată în memorie. Există patru clase de memorare: **statice, automatice, externe și registru**. Ele se pot declara implicit prin cuvintele: **static, auto, extern** și respectiv **register**.

Variabilele descrise prin *"static"* pot rămâne în memoria calculatorului pe tot parcursul executării programului păstrându-și aceeași adresă. Ele sunt alocate la compilare.

Variabilele descrise prin *"auto"* sunt variabile cu alocare automată a adresei în memorie. Ele sunt create și li se alocă spațiu în memorie numai la execuția blocului de program în care sunt declarate. La închiderea executării blocului, variabila dispare și spațiul de memorie se alocă altor blocuri. Dacă se revine ulterior în blocul unde au fost declarate, li se alocă din nou memorie putând primi altă adresă. Alocarea variabilelor din clasa *"auto"* se face pe stiva sistemului.

Variabilele locale, al căror domeniu de valabilitate se limitează la un bloc sunt implicit automatice, chiar dacă nu se menționează acest lucru. De asemenea, parametrii formali fac parte din clasa *"auto"* și deci se alocă pe stiva sistemului. Dacă vrem ca o variabilă locală să nu fie alocată pe stivă (deci să nu fie auto) vom preceda declarația ei prin cuvântul *"static"*.

Variabilele din clasa *"register"* sunt tot automatice, dar ele sunt asociate unor registri ai microprocesorului și nu unor locații de memorie, crescând viteza de execuție a programului.

Dacă dorim ca o variabilă globală să fie folosită și de funcții din alt fișier sursă, sau de funcții din același fișier sursă definite înaintea declarării variabilelor, trebuie ca variabila să fie declarată ca **externă**, precedând declarația sa prin cuvântul *"extern"* în funcțiile respective sau în afara oricărei funcții din noul fișier. Acum fișierele pot fi compilate separat și linkeditate împreună.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu.

Fişierul f1.CPP	*Fişierul f2.CPP*
.....	.....
int a,b;	*extern int a,b;*
	int prod( );
	*{return a*b;}*

Cele două fişiere se pot compila separat.

Pentru obţinerea codului executabil includem în f2.CPP fişierul f1.CPP prin directiva *#include<f1.CPP>*.

2.13.8. Categorii de pointeri [47]

Microprocesoarele 80x86 adresează 1MO memorie în blocuri de câte 64KO numite **segmente**. Adresarea octeţilor dintr-o zonă de 1MO necesită adrese de 20 biţi, iar dintr-o zonă de 64KO necesită adrese de 16 biţi.

Segmentarea face ca adresa fizică a unei locaţii de memorie de 1MO să fie alcatuită din două componente de câte 16 biţi, ceea ce constituie **adresa logică**. Prima componentă este adresa de segment pe 16 biti, iar a doua este deplasarea în cadrul segmentului, numită **deplasament** (offset). Astfel:

adresa_logică=adresa_segment:deplasament

Microprocesorul calculează automat adresa fizică prin formula:

adresa_fizică=16xadresă_segment+deplasament

Microprocesoarele dispun de 4 registri de segment specializate, in care memoreaza adresa de inceput de segment si prin care ele pot adresa următoarele 4 segmente:

a. **Segmentul de cod** (instrucţiuni în cod maşină) cu adresa memorată în registrul **CS (Code Segment)**. Deplasamentul segmentului de cod este memorat în registrul **IP (Instruction Pointer)** care este automat incrementat. Astfel, adresa **CS:IP** indică adresa următoarei instrucţiuni de executat.

b. **Segmentul de date** cu adresa în **DS(Data Segment)**. Deplasamentul este dat de modul de adresare folosit, prin care se stabilesc componentele ce se însumează (conţinutul registrilor BX sau BP şi SI sau DI).

c. **Segmentul de date suplimentar**, cu adresa în **ES (Extra Segment)**.

https://biblioteca-digitala.ro / https://unibuc.ro

d. **Segmentul de stivă**, cu adresa în **SS (Stack Segment)**. Deplasamentul vârfului stivei este memorat în **SP(Stack Pointer)** şi astfel vârful stivei se află la adresa SS:SP.

Pointerii de date din C++ sunt adrese dintr-un segment logic de date, iar pointerii de funcţii sunt adrese dintr-un segment logic de cod.

Dacă toate datele respectiv funcţiile sunt într-un singur segment se pot folosi variabile pointer de 16 biţi numite **pointeri near**. Ei conţin doar deplasamentul în segment.

Dacă datele respectiv funcţiile ocupă mai mult de 64KO se folosesc variabile pointer de 32 biţi numite **pointer far**. Ele conţin două componente de 16 biţi care formează adresa logică (adresa de segment şi deplasamentul).

Operaţiile aritmetice cu pointeri far modifică doar deplasamentul şi rezultatul este modulo 65536.

Operatorii relaţionali compară doar deplasamentul, iar operatorii == şi != compară pointerii ca valoare întreagă reprezentată pe 32 de biţi.

Pentru a uşura calculele cu **pointeri far** se folosesc pointeri care memorează adresa logică sub o formă normalizată numiţi **pointeri huge**.

Pointerii near, far şi huge se definesc în mod normal plasand aceste cuvinte cheie imediat după tipul datei de tip pointer.

Exemplu.

*float near *pn;*
*int far *pf;*
*double huge *ph;*

Compilatorul C++ dispune de 6 modele de organizare a memoriei, alese în funcţie de necesităţi. Acestea sunt:

TYNY = aici regiştrii de segment încep de la aceeaşi adresă, iar programul constă dintr-un singur segment de 64 KO pentru cod, date şi stivă. Modelul foloseşte **pointerii near**.

SMALL = aici segmentele de date şi cod diferă având fiecare câte 64KO, iar cele de stivă şi extra se suprapun. Modelul foloseste **pointerii near**. Este modelul cel mai indicat aplicaţiilor uzuale.

MEDIUM = aici datele statice se limitează la 64KO, iar codul poate ajunge la 1MO. Modelul foloseşte **pointeri far** pentru cod şi **pointeri near** pentru date. Este recomandat aplicaţiilor mari cu date puţine.

COMPACT = aici datele ocupă mai multe segmente până la 1MO, iar codul este limitat la 64KO. Se folosesc pointerii near pentru cod şi near şi far pentru date. Datele sunt organizate astfel:

https://biblioteca-digitala.ro / https://unibuc.ro

- un segment de 64KO pentru date statice; implicit pointeri near
- un segment de 64KO pentru stivă
- restul de memorie se alocă datelor dinamice care folosesc pointeri far

LARGE = folosit pentru programe foarte mari. Modelul foloseşte pointeri far pentru date şi cod. Aici datele şi codul pot ocupa fiecare până la 1MO.

HUGE = aici datele statice pot depăşi 64KO. Se folosesc pointeri far pentru date şi cod. Spaţiul de memorie este organizat astfel:

- până la 1MO pentru cod
- până la 1MO pentru date statice
- până la 1MO pentru date dinamice
- un segment de 64KO pentru stivă.

În orice model, codul obţinut după compilare trebuie să ocupe mai puţin de 64KO pentru a intra într-un singur segment de cod. Dacă ocupă mai mult de 64KO programul se împarte în mai multe fişiere sursă care se vor compila separat şi apoi se vor linkedita împreună. Se procedează analog şi pentru date.

Pointerii huge trebuie declaraţi explicit în fiecare model atunci când dorim să lucrăm cu ei.

2.13.9 Alocarea dinamică a memoriei

C++ foloseşte operatorii **new** şi **delete** creaţi în special pentru alocarea dinamică a memoriei. Operatorul new returnează un pointer la zona alocată dinamic. Dacă nu există memorie suficientă, alocarea nu reuşeşte şi new returnează pointerul NULL.

Operatorul new foloseşte sintaxele:

p=new tip;
p=new tip(expresie);
p=new tip[dim];

În a doua variantă, variabila dinamică creată cu new se iniţializează cu valoarea expresiei din paranteză.

A treia variantă alocă in mod dinamic **dim** variabile dinamice de tipul **tip** (un tablou cu dim elemente). În acest caz, tabloul nu se poate iniţializa.

Exemplu. ***int *p,*q,*r;***
p=new int; (se alocă memorie pentru un întreg)

https://biblioteca-digitala.ro / https://unibuc.ro

q=new int(7); (ca şi mai sus, şi, în plus, q se iniţializează cu valoarea 7)

r=new[10]; (alocă un tablou de 10 întregi)

Eliberarea zonei de memorie alocată cu new se face cu operatorul **delete**, sub forma:

delete variabilă;

Exemplu.

delete p;delete q; delete r;

2.13.10 Referinţe

Pentru a simplifica lucrul cu pointeri în C++ s-a introdus **tipul referinţă** care implementează perfect conceptul de transmitere a parametrilor prin referinţă.

O **referinţă** este un nume alternativ pentru un obiect.

Pentru tipul T, o referinţă la tipul T se declară sub forma **T&** sau **T &.**

O referinţă trebuie iniţializată când este declarată, iniţializare ce nu trebuie confundată cu atribuirea. Ea defineşte un **alias** (alt nume) al obiectului cu care a fost iniţializată.

O referinţă nu se mai poate modifica după iniţializare. Ea referă totdeauna acelaşi obiect stabilit prin iniţializare. Pentru a obţine un pointer la obiectul desemnat de referinţa r se poate folosi **&r**.

Exemplu.

int i=1;

int& r=i; (aici r şi i se referă la acelaşi obiect, iar r este un nou nume pentru i)

int x=r; (x va primi valoarea 1)

r=2; (variabila i primeşte valoarea 2)

Referinţele sunt utile când se folosesc ca argumente pentru funcţii.

Exemplu.

```
#include<iostream.h>
void interschimb(int& a, int& b)
{int z=a;
a=b;
b=z;}
void main( )
{int x=10,y=20;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
interschimb(x,y);
cout<<"x="<<x<<"y="<<y;}
```

La apelare, argumentele x şi y ale funcției **interschimb** devin alte nume pentru variabilele a şi b şi de aceea operațiile se fac direct asupra variabilelor x şi y.

Astfel, semantica transmiterii argumentelor este aceea a inițializării.

Dacă tipul unei funcții este o referință, ea trebuie să întoarcă un obiect de tipul respectiv.

Exemplu.

```
int& f( )
{static int x;
.....
return x;}
```

Funcția întoarce variabila de tip int cunoscută în interiorul funcției prin identificatorul x. În astfel de situații variabila returnată trebuie să fie **statică** sau alocată cu **new** astfel:

```
int& f( )
{int& x=*new int;  (se dă un nume variabilei anonime *new int)
.....
return x;}
```

Exemplu. Se reprezintă eficient o matrice simetrică în care pentru $\forall i \in \{0,...,n-1\}$ trebuie cunoscute elementele a[i][0], ... , a[i][i].

```
#include<iostream.h>
void alocare(int **&a, int n)          (aici a este variabilă de ieşire)
{int i;
a=new int*[n];          (aici a este pointer la n pointeri catre intregi)
for (i=0; i<n; i++)
  a[i]=new int[i+1]; }          (aici a[i] este pointer la i+1 întregi)
void eliberare (int **a, int n)
{int i;
for (i=0; i<n; i++)
  delete a[i];                    (şterge vectorul indicat de a[i])
delete a; }                (şterge vectorul de pointeri indicat de a)
void citire(int **a, int n)
{int i,j;
for (i=0; i<n; i++)
  for (j=0; j<=i; j++)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
    cin>>a[i][j];}
  void scriere(int **a,int n);
  {int i,j;
  for (i=0; i<n; i++)
    {for (j=0; j<=i; j++) cout<<a[i][j]<<" ";
    cout<<endl;} }
  void main ( )
  {int**a, n=4;
  alocare(a,n);
  citire(a,n);
  scriere(a,n); eliberare(a,n); }
```

Funcția **alocare** poate returna adresa la care este alocată matricea triunghiulară folosind definitia de mai jos, lasand celelalte funcții neschimbate si modificand funcția principală dupa cum urmeaza:

```
int **alocare(int n)
{int **a;
int i;
a=new int*[n];
for (i=0; i<n; i++)
  a[i]=new int[i+1];
return a; }
```

```
void main( )
{int n=4;
int **a=alocare(n);
citire(a,n);
scriere(a,n);
eliberare(a,n);}
```

2.13.11 Probleme rezolvate [47]

a. Transfer rapid de blocuri:

```
#include<iostream.h>
const dim=5;
int a[dim]={1,2,3,4,5},b[dim];
void main( )
{struct vect {
        int v[dim]; };
*(vect*)b=*(vect*)a;
cout<< "b=";
for (int i=0; i<dim; i++) cout<<b[i]<< " ";
cout<<endl; }
```

b. Fie dat un poligon cu n ≥ 3 vârfuri de coordonate $P(x_i,y_i)$, $i \in \{1,...,n\}$. Se cere aria poligonului și coordonatele centrului său de greutate.

https://biblioteca-digitala.ro / https://unibuc.ro

Solutie : Pentru arie folosim formula: $A= \frac{1}{2} |(x_1+x_2)(y_1-y_2)+...+(x_n+x_1)(y_n-y_1)|$, iar pentru centrul G(x,y) folosim formula:

$$x=\frac{1}{n}\sum_{i=1}^{n} x_i \quad y=\frac{1}{n}\sum_{i=1}^{n} y_i$$

```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
struct punct {
        float x,y; } ;
int citire (punct*&P)
{int n;
cout << "Număr puncte="; cin>>n;
P=new punct[n];  (alocare dinamică pentru P[1],...,P[n])
for (int i=1;i<=n; i++)
  {cout<<"P"<<i<<"(xy)=";
  cin>>P[i].x>>P[i].y;}
return n;}
float arie (punct P[ ], int n)
{float s=(P[n].x+P[1].x)*(P[n].y-P[1].y);
for (int i=1; i<n; i++) s+=(P[i].x+P[i+1].x)*(P[i].y-P[i+1].y);
return fabs (s)/2 ;}
punct centru (punct P[ ], int n)        (funcția poate returna o dată
                                         structurată)
{ punct G={0,0};
for (int i=1; i<=n; i++){
  G.x+=P[i].x; G.y+=P[i].y; }
G.x/=n; G.y/=n;
return G;}
void main( )
{punct *P;
int n=citire(P);
float S=arie(P,n);
punct G=centru(P,n);
cout<<setprecision(4);;
cout<< "Aria poligonului este:"<<S<<endl;
cout<< "Centrul de greutate este G("<<G.x<< ","<<G.y<<
").”<<endl; }
```

https://biblioteca-digitala.ro / https://unibuc.ro

c. **Lemă:** $\forall x \in (0,1) \cap Q$, $\exists q_1,\ldots,q_r$ naturale nenule astfel încât $0<q_1<\ldots<q_r$,

$$x = \frac{1}{q_1} + \frac{1}{q_2} + \ldots + \frac{1}{q_r}$$ şi pentru $\forall k=2,\ldots,r$, $q_{k-1} \mid q_k$

Demonstrație: Fie $x = \frac{m}{n}$ cu (m,n)=1 şi m<n.

Demonstrăm prin inducție după m.

Pentru m=1, $x = \frac{1}{n}$ care are forma de mai sus.

Presupunem afirmația adevărată pentru fracțiile $\frac{i}{n}$, $\forall i<m$ şi o demonstrăm pentru i=m.

Din $m<n \Rightarrow \exists q$ şi r astfel încât $n=m\cdot q+r$ şi $r>0$, $r<m \Rightarrow n=m\cdot(q+1)-(m-r)$.

Fie $q_1=q+1$ şi $m_1=m-r<m$. Astfel, $n=m\cdot q_1 - m_1$ (1)

$$\frac{m}{n} = \frac{m}{n+m_1}\cdot\frac{n+m_1}{n} = \frac{m}{n+m_1}\cdot\left(1+\frac{m_1}{n}\right)$$ dar $\frac{n+m_1}{m} = q_1$ din (1),

iar $m_1<m<n$ şi astfel, conform ipotezei de inductie pentru $m_1<m$ obtinem :

$$\frac{m}{n} = \frac{1}{q_1}\left(1+\frac{m_1}{n}\right) = \frac{1}{q_1}\left(1+\frac{1}{d_1}+\frac{1}{d_2}+\ldots+\frac{1}{d_k}\right)$$

unde $0<d_1<\ldots<d_k$ şi $d_{i-1} \mid d_i$, $\forall i=2,\ldots,k$. Astfel,

$$\frac{m}{n} = \frac{1}{q_1} + \frac{1}{d_1 q_1} + \frac{1}{d_2 q_1} + \ldots + \frac{1}{d_k q_1}$$

unde $0<q_1<d_1\cdot q_1<\ldots<d_k\cdot q_1$, $q_1 \mid d_1\cdot q_1$ şi pentru $\forall i=2,\ldots,k$ avem $d_{i-1}\cdot q_1 \mid d_i\cdot q_1$.

Programul care afişează un $x \in (0,1)$ sub forma de mai sus foloseşte funcția **fr(int m, int n, long q)** pentru a face acest lucru.

```
#include <iostream.h>
long cmmdc(long a, long b)
{if (b==0) return a;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
return cmmdc(b,a%b);}
void fr(int m, int n, long q)
{long d=cmmdc(m,n);
m=m/d; n=n/d;
if (m= =1) {q=1.0*n;
  cout<<"1/"<<q<<endl;}
else {q=q*(n/m +1);
  cout<<"1/"<<q<<"+";
  fr(m-n%m,n,q);}
void main( )
{int m,n;
cout<<"m<n="; cin>>m>>n;
cout<<m<<"/"<<n<<"="<<fr(m,n,1);}
```

2.13.12 Pointeri ce memorează adrese de funcții [47]

În declarația ***int f( );*** f este un pointer ce indică adresa lui f. Este un pointer constant, și deci nu poate fi modificat prin instrucțiuni din program.

O variabilă pointer care să memoreze adresa unei funcții se declară prin:

tip (*nume_pointer)(tip_parametri);

Exemplu. ***int (*adrF)( );***

Aici, adrF poate memora adresa unei funcții care nu are parametri și calculează un rezultat întreg.

Observație. Declarația ***int *adrF( );*** descrie o funcție care calculează un pointer la o valoare de tip int și nu o variabilă pointer la funcție.

Atunci când folosim doar numele funcției fără ( ) și fără lista parametrilor se obține adresa de început a funcției.

Astfel, ***adrF=f;*** atribuie variabilei pointer la o funcție adresa de început a funcției f care nu are parametri și întoarce un rezultat de tip întreg.

Atribuirile ***adrF=f( );*** sau ***adrF=&f( );*** sunt greșite deoarece f( ) reprezintă valoarea returnată de funcție și &f( ) este adresa rezultatului întors de funcție, dacă acest rezultat este o referință.

Apelarea funcției a cărei adresă este memorată într-o variabilă pointer se face, pentru exemplul de mai sus, prin ***adrF( )*** sau ***(*adrF)( )***.

Prin construcția ***typedef*** putem atribui un nume și tipului pointer la funcții sub forma: ***typedef double(*funcție)(double)***, unde ***funcție*** este identifi-

https://biblioteca-digitala.ro / https://unibuc.ro

catorul tipului de date, pointer la funcție care întoarce o valoare de tip double și are un argument double.

Acum are sens atribuirea **funcție f=sin;**

Exemplu. Ordonare crescătoare/descrescătoare a elementelor unui vector.

```
#include<iostream.h>
int asc(int a, int b) {return a<=b;}
int desc(int a, int b) {return a>=b;}
typedef int(*criteriu)(int,int);
void sort(int x[ ], int n, criteriu ord);
{int s=0;
while (!s);
{s=1;
for (int i=0; i<n-1; i++)
  if (!ord(x[i],x[i+1]))
{int aux=x[i]; x[i]=x[i+1]; x[i+1]=aux; s=0;}
n--;} }
void main( )
{int x[5]={3,5,1,8,7};
cout<<"Ordonare crescătoare:";
sort(x,5,asc);
for (int i=0; i<5; i++) cout<<x[i]<<" ";
cout<<endl<<"Ordonare descrescătoare:";
sort(x,5,desc);
for(i=0;i<5;i++) cout<<x[i]<< " "
cout<<endl;
}
```

2.13.13 Supraîncărcarea funcțiilor

Existența mai multor funcții cu același nume dar cu argumente diferite ca tip sau număr poartă numele de **supraîncărcarea funcțiilor**.

Compilatorul identifică funcția apelată examinând tipul argumentelor și prin încercarea de a face corespondența cu argumentele efective. El nu verifică tipul valorii returnate de funcție.

Funcțiile care au parametri declarați static și cele care au argumente de tip referință nu pot fi supraîncărcate.

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplu. ***int adună(int a,int b) {return a+b;}***
double adună (double a, double b) {return a+b;}
char *adună(char*a,char*b);strcat(a,b); return a;}

Aceste trei funcții sunt supraîncărcate.

Funcțiile: ***void f(int);***
void f(static int);
void f(int&); nu pot fi supraîncărcate deoarece pentru al doilea caz compilatorul nu poate face distincția între ***int*** și ***static int***, iar pentru ultimul caz, compilatorul nu distinge între int și int&, deci nu știe ce funcție să apeleze.

2.13.14 Clase, obiecte, metode, specificatori de acces [47, 58]

Până la limbajul C++ se făcea o distincție netă între structuri de date și funcțiile care operau cu acestea. Programarea pe obiecte strânge sub același acoperiș datele și operațiile de prelucrare a lor prin intermediul noțiunii de **clasă**. Prin conceptul de clasă putem defini mai multe tipuri de date care se manipulează într-o manieră identică cu cea a tipurilor predefinite int, char, float, etc.

Toate operațiile ce se pot face cu noul tip de date sunt prevăzute în definiția clasei. Ele sunt reprezentate prin funcții membre numite **metode**. O categorie specială de metode o constituie operatorii care permit folosirea unor notații mai naturale și mai expresive. O clasă are acces la datele membre doar prin intermediul metodelor.

Cuprinderea împreună a datelor și metodelor se numește **încapsulare**. Termenul sugerează că pentru utilizator structura obiectului din noul tip este neesențială și inaccesibilă. Pentru el obiectul din noul tip există doar sub forma tuturor metodelor de acces la el.

Membrii unei clase (date sau funcții) ce aparțin secțiunii **private** sunt accesibili numai funcțiilor membre și celor prietene clasei. Ei nu pot fi accesați direct din exteriorul clasei, ci numai prin intermediul funcțiilor membre publice sau a celor prietene clasei. Implicit, membrii unei clase au specificatorul de acces *private*.

Membrii publici sunt în mod normal metode (funcții). Ei pot fi accesați de orice utilizator.

Secțiunea **proteced** este intermediară între **private** și **public** și are rolul de a permite accesul la membrii săi dintr-o clasă derivată din clasa respectivă. Dintr-o astfel de clasă deivată nu putem avea acces niciodată

https://biblioteca-digitala.ro / https://unibuc.ro

la membrii privați, dar putem avea acces la membrii protejați și la cei publici.

Funcțiile prietene unei clase, cele care au în față atributul **friend,** au fost introduse pentru cazurile în care o funcție cooperează strâns cu o clasă dar nu este membră a clasei respective. O funcție poate fi prietenă mai multor clase.

Dacă o funcție este definită în interiorul unei clase, ea este implicit funcție *inline*. Dacă o funcție membră unei clase are un bloc de instrucțiuni foarte complex, ea va trebui definită în afara clasei, folosind operatorul de rezoluție : : plasat între numele clasei și numele funcției. El specifică apartenența funcției respective la clasa din antetul său.

În C++ la fiecare utilizare a unei variabile membru trebuie specificat obiectul căruia îi aparține și la fiecare apel de funcție membru trebuie specificat obiectul care o apelează.

Variabilele membre publice sunt desemnate prin operatorii "." și "→" după cum obiectele de care aparțin sunt desemnate printr-un nume si respectiv printr-un pointer. Variabilele membre private nu pot fi accesate decât în interiorul metodelor clasei respective.

Metodele pot fi apelate sub următoarele trei forme:

nume_obiect.nume_funcție(listă_parametri);

pointer_la_obiect→nume_funcție(listă_parametri);

folosirea de pointeri la metode (dacă L este o clasă, atunci definirea unui pointer la o metodă a clasei L se face prin L::*).

Exemple.

```
#include<iostream.h>
class k
{int a;
public:
  k(int n=0)      /constructor de inițializare
  {a=n;}
  int get( )      /metodă
  {return a++;}
};
void main( )
{int(k::*pm)( );    pm = variabilă pointer la metoda ce întoarce o
                    valoare de tip int si are o lista vidă de parametri
pm=k::get;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
k x, *y;
y=new k[5];
cout<<(x.*pm)()<<endl;
cout<<(y→*pm)()<<endl;}
```

Observăm că operatorii "." şi "→" sunt folosiți si pentru manipularea pointerilor la metode.

2.13.15 Constructori [46, 47, 58]

Constructorii unei clase sunt funcții membre speciale care stabilesc cum sunt create şi inițializate obiectele unei clase. Nu se deosebesc de funcțiile obişnuite şi pot efectua orice fel de acțiuni. Se particularizează prin:

- numele lor coincide cu numele clasei din care fac parte
- nu returnează valori
- nici la declarare şi nici la definire nu au tipul void ca tip returnat
- sunt generați de compilator dacă nu sunt definiți explicit
- adresa unui constructor nu se poate păstra într-un pointer
- o clasa poate avea mai multi constructori care difera prin tipul şi/sau numărul parametrilor
- pot avea în lista de parametri şi parametri cu valori implicite

Constructorii pot fi de următoarele patru tipuri, unde C este o clasă, iar K este o clasă diferită de C.

a) ***C::C( )*, constructor implicit**, are o listă vidă de parametri.
b) ***C::C(listă_parametri)*, constructor de inițializare**, cu listă nevidă de parametri.
c) ***C::C(C&,...)*, constructor de copiere**, care copiază un obiect de acelaşi tip. Alți parametri, dacă există, au valori implicite.
d) ***C::C(K&,...)***
***C::C(K,...)*, constructori de convertire a tipului si de copiere** a unui obiect de tip K în unul de tip C.

Constructorii cu lista de parametri nevidă se apelează prin:

C x=C(argumente); sau
C x(argumente); sau ***C *p=newC(argumente);*** sau chiar prin
C* p=newC; pentru listă vidă de argumente sau argumente cu valori implicite,

unde C este numele clasei, iar x este un obiect al clasei cu numele C.

Ultimele două forme de apel permit crearea obiectelor dinamice.

https://biblioteca-digitala.ro / https://unibuc.ro

Un **constructor implicit** se obține prin definirea de programator a unui constructor fără parametri, al cărui corp poate conține orice fel de instrucțiuni admise de un context dat, sau prin generarea automată de compilator, când nu există un alt constructor pentru acea clasă, caz în care corpul său nu conține nici o instrucțiune.

Un constructor implicit este necesar când declarăm tablouri de obiecte ale clasei respective.

Un constructor în care toate argumentele au valori implicite nu este un constructor implicit.

Constructorii impliciți se apelează prin: ***C x=C( );*** sau ***C x;***. Aici x este un obiect al clasei C, construit prin constructorul implicit C::C( );

Un **constructor de copiere** este un constructor al cărui prim parametru este o referință la același tip ca și clasa. Dacă mai există și alți parametri, ei trebuie să aibă valori implicite pentru a avea un constructor de copiere. El este folosit pentru a inițializa un obiect cu un alt obiect din aceeași clasă.

Un constructor al clasei C nu poate avea un parametru de tip C, deși un parametru referință la tipul C, adică C& este acceptat.

Orice clasă trebuie să aibă un constructor de copiere. În lipsa acestuia, compilatorul va genera unul, care va copia membru cu membru toate variabilele **argumentului** (obiectul curent) în cele ale obiectului care apelează metoda.

Un constructor de copiere se apelează prin: ***C x=y***; sau ***C x(y);*** unde x și y sunt obiecte ale clasei C.

Un **constructor de conversie tip** este un constructor al cărui prim parametru este diferit de tipul clasei. Dacă există și alți parametri, ei trebuie să aibă valori implicite, pentru a avea un constructor de conversie tip.

Se apelează sub una din formele:

a) ***C x=C(y);***
b) ***C x(y);***
c) ***C x;***
   x=C(y);

Aici y este convertit în x.

https://biblioteca-digitala.ro / https://unibuc.ro

2.13.16 Destructori [47, 58]

Destructorii sunt opusul (complementul) constructorilor. Ei eliberează memoria dinamică ocupată de membrii unui obiect înainte ca acesta să fie distrus.

Destructorii se caracterizează prin:

- au numele clasei de care aparţin precedat de ~
- nu returnează valori şi nu au listă de parametri
- nici la definire şi nici la declarare nu acceptă tipul void ca tip returnat
- adresa lor nu este accesibilă utilizatorilor
- o clasă nu poate avea decât cel mult un destructor. Dacă acesta lipseşte, compilatorul generează automat unul
- nu pot fi supraîncărcaţi
- sunt apelaţi automat la încheierea blocului în care a fost creat obiectul ce trebuie să dispară. Pentru variabile globale, destructorii sunt apelaţi la încheierea funcţiei principale.

În cazul variabilelor dinamice, destructorul nu este apelat implicit. Aceste obiecte pot fi distruse şi prin operatorul **delete** sub forma ***delete pointer;***

Pentru obiectele reprezentate doar de variabile referinţă, destructorii nu sunt apelaţi automat.

Dacă într-un program se foloseşte funcţia ***exit( )*** se vor apela automat doar destructorii variabilelor globale, nu şi cei ai variabilelor locale.

Dacă un program apelează funcţia ***abort( )*** nici un destructor nu va mai fi apelat automat. Obiectele se pot distruge prin apelarea explicită a destructorilor sub forma:

obiect.nume_clasă::~nume_clasă( );

Exemplu.

```
#include<iostream.h>
class C
{int n;
public:
        C(int k=1)
        {n=k;
        cout<<"Creem obiectul"<<n<<endl;}
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
    ~C( )
    {cout<<"Distrugem obiectul"<<n<<endl;} };
C x(10);                variabilă globală inițializată cu 10
void main( )
{C a,b=C(2),t(3),d=4,*p=new C(5);
C& y=*new C(6);         y este o referință la obiectul *new C(6).
                        Pentru acesta nu se apelează automat destructorul
delete p;               apelare explicită a destructorului pentru obiectul *p
y.C::~C( );             apelarea destructorului pentru obiectul desemnat
                         de referința y
}                       se apelează automat destructorii obiectelor a,b,t,d
```

2.13.17 Supraîncărcarea operatorilor [46, 47]

Unii operatori de bază din C++ pot fi redefiniți prin **funcții operator** care dau programatorului posibilitatea de a adăuga operatorilor și alte sensuri decât cele predefinite, nepierzând însă sensurile vechi. Nu se pot defini noi operatori prin astfel de funcții și, în plus, prioritatea și aritatea operatorilor nu se poate modifica prin funcțiile operator. Nu se pot redefini operatorii: .,*, ::, ? :.

Funcțiile operator pot fi membre unor clase sau prietene acestora. Dacă funcția operator nu este membră a unei clase, cel puțin unul din parametrii săi trebuie să fie un obiect dintr-o clasă.

Pentru apelarea unei funcții operator avem cazurile:

- dacă funcția operator este membră unei clase atunci:
  - dacă supraîncarcă un operator unar nu va avea nici un parametru, deoarece operatorul unar are nevoie de un simplu operand și anume obiectul curent care a apelat metoda
  - dacă supraîncarcă un operator binar, metoda are un singur parametru, care reprezintă al doilea operand deoarece primul este obiectul curent
- dacă funcția operator este sau nu prietenă unei clase, atunci când supraîncarcă un operator unar necesită un parametru, iar când supraîncarcă un operator binar, necesită doi parametri

https://biblioteca-digitala.ro / https://unibuc.ro

Observații:

a. Orice funcție operator prietenă clasei C trebuie să aibă cel puțin un parametru de tip C. Unica excepție de la această regulă o constituie operatorii new și delete.

b. Operatorii redefiniți nu moștenesc comutativitatea celor predefiniți.

c. Nu se poate defini o funcție operator comutativă care să fie metodă și să opereze cu operanzi de tipuri diferite, deoarece, implicit primul operand este obiectul care apelează metoda. Acest lucru se poate realiza ușor prin mecanismul de funcție prietenă.

d. Nu se poate defini o funcție operator ca metodă statică.

e. Operatorul de intrare ">>" numit și **extractor** deoarece la citire se extrag date dintr-un șir de intrare, poate fi supraîncarcat. Toate funcțiile extractor au sintaxa:

istream& operator>>(istream& șir,nume_clasă obiect)
{corpul extractorului
return șir;}

Primul parametru este o referință a unui obiect de tip istream, adică sir este un șir de intrare. Al doilea este obiectul de tip nume_clasă ce va fi citit (extras din șir).

Extractorul întoarce o referință a unui șir de tip **istream** ceea ce permite rezolvarea unei înlănțuiri de tipul: cin>>ob1>>ob2>>...;. Un extractor nu poate fi funcție membră a clasei în care operează deoarece, în caz contrar, ar trebui ca primul parametru să fie obiectul curent al clasei respective, dar acesta este un șir, și deci extractorul nu poate fi o funcție membră.

f. La supraîncărcarea operatorului de ieșire <<, numit și **insertor** deoarece la scriere el inserează informații în șirul de ieșire, se creează o funcție operator numită tot insertor. Toate funcțiile insertor au sintaxa:

ostream& operator<<(ostream& șir,nume_clasă obiect)
{corpul insertorului
return șir;}

Primul parametru este o referință a unui obiect de tip **ostream**, ceea ce spune că **șir** este un șir de ieșire. Al doilea parametru este obiectul ce va fi afișat. Insertorul întoarce o referință a unui șir de tip **ostream**, ceea ce permite rezolvarea înlănțuirilor de forma:

cout<<ob1<<ob2<<ob3<<....;

https://biblioteca-digitala.ro / https://unibuc.ro

Insertorul nu poate fi funcție membră a clasei în care operează deoarece, altfel, ar trebui ca primul parametru al său, adică operandul din stânga, să fie obiectul curent al clasei respective, dar acesta este un şir şi astfel insertorul nu poate fi o funcție membră.

Exemplu. [47] Vom descrie o clasă numită *fracție*, care oferă suport de lucru cu fracții ordinare. Notăm cu p/q fracția.

Vom folosi:

- un constructor implicit **fracție( )** pentru a putea defini vectori de fracții
- un constructor **fracție(long m, long n)** care inițializează numărătorul fracției cu m şi numitorul cu n
- două funcții prietene (implicit inline), **numărător(fracție f)** şi **numitor(fracție f)** care întorc numărătorul, respectiv numitorul fracției dată ca argument. Au corpul definit în interiorul clasei
- două funcții necesare simplificării fracțiilor, care au corpul descris în afara clasei. Prima, numită **simplific(f)** este prietenă clasei şi întoarce fracția obținută după simplificarea celei date ca argument, fără să afecteze fracția argument. A doua este o metodă (o funcție membră clasei fracție). Ea simplifică fracția (obiectul) pentru care se apelează.
- supraîncărcarea funcției ***abs*** încât să funcționeze şi când argumentul este de tip fracție
- supraîncărcarea operatorului binar **op**$\in$ {+,-,*,/} prin funcții operator prietene clasei fracție având corpul descris în afara clasei, pentru a avea sens expresii de forma $f_1 op f_2$, $nop f_1$ unde f_1 şi f_2 sunt obiecte de tip fracție, iar n este o expresie întreagă. Observăm de exemplu că funcția operator pentru expresii de forma nopf nu poate fi funcție membră a clasei fracție deoarece primul său argument (operatorul din stânga) nu este de tipul clasei curente.
- Supraîncărcarea operatorului binar **op**$\in$ {+,-,*,/} printr-o funcție operator membră a clasei. Primul operand este obiectul curent f_1, iar al doilea este fracția f_2. Cu ea putem evalua expresii de forma $f_1 op f_2$ unde f_1 şi f_2 sunt obiecte de tip fracție.
- Supraîncărcarea operatorilor relaționali = = şi != prin funcții operator inline implementate (descrise) în afara clasei, pentru a putea folosi expresii de forma $f_1 = = f_2$ sau $f_1 != f_2$, unde f_1 şi f_2 sunt obiecte de tip fracție.

https://biblioteca-digitala.ro / https://unibuc.ro

- Supraîncărcarea operatorilor de forma "op=" prin funcții membre clasei fracție, în care primul operand este obiectul curent, iar al doilea este o fracție. Funcțiile returnează obiectul curent. Cu ele au sens expresii de forma $f+=f_1$, $f*=n$, etc.
- Supraîncărcarea operatorilor unari + și – prin funcții membre care nu alterează argumentul folosit în apel. Astfel au sens expresii de forma +f și –f.
- Supraîncărcarea operatorului de **conversie tip** printr-o funcție care trebuie să fie totdeauna membră a clasei din care se face conversia. Declarația și definiția nu trebuie să specifice nici un tip rezultat deoarece acesta este implicit tipul al cărui nume îl poartă. Funcția lucrează asupra obiectului curent și deci nu are parametri.
- Folosirea cuvântului cheie ***this***. Se știe că apelul unei funcții membră a unei clase sub forma **obiect.metodă** realizează întâi transmiterea către metodă a pointerului asupra căruia acționează metoda. Ne putem referi la acest pointer și prin cuvântul ***this***. În orice funcție nestatică a unei clase C, pointerul ***this*** se declară implicit sub forma: ***C *this;*** și este inițializat să indice spre obiectul curent pentru care este apelată funcția membru. Astfel, ***this*** întoarce adresa de început a obiectului curent. Notația ****this*** înseamnă referirea la întregul obiect.

Clasa fracție declarată și descrisă în continuare poate fi inclusă într-un fișier antet "FRACTIE.H".

```
#include<math.h>
#include<iostream.h>
long cmmdc(long a, long b)
{a=labs(a); b=labs(b);
long aux;
while (b)
        {aux=a;
        a=b;
        b=aux%b;}
return(a>0)?a:1;}
class fracție
{private:
        long p,q;
public:
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        fracție( );
        {p=0;
        q=1;}
fracție(long m, long n=1)
        {p=m;
        q=n;}
friend long numărător(fracție& f)
        {return f.p;}
friend long numitor(fracție& f)
{return f.q;}
friend fracție simplific(fracție);
fracție simplific( );
friend fracție abs(fracție& f)
{return fracție(labs(f.p),labs(f.q));}
firend fracție operator +(fracție& f1, fracție& f2);
friend fracție operator +(fracție& f, long n);
friend fracție operator +(long n, fracție f)
{return f+n;}
fracție  operator –(fracție& f2);
friend fracție operator -(long n, fracție& f);
friend fracție operator -(fracție& f, long n);
friend fracție operator *(fracție& f1, fracție& f2);
friend fracție operator *(fracție& f, long n);
friend fracție operator *(long n, fracție& f);
friend fracție operator /(fracție& f1, fracție& f2);
friend fracție operator /(fracție& f, long n);
friend fracție operator /(long n, fracție& f);
friend int operator = =(fracție f1, fracție f2);
friend int operator !=fracție f1, fracție f2);
fracție& operator +=(fracție& f);
fracție& operator +=(long n);
fracție& operator -=(fracție& f);
fracție& operator -=(long n);
fracție& operator *=(fracție& f);
fracție& operator *=(long n);
fracție& operator /=(fracție& f);
fracție& operator /=(long n);
fracție operator +( )
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
{return *this;}
fracție operator -( )
{return fracție(-p,q);}
operator double( )
{return double(p)/q;}
operator float( )
{return float(p)/q;}
operator long( )
{return p/q;}
operator int( )
{return p/q;}
friend ostream& operator <<(ostream& șir,fracție& f)
{șir<<"("<<f.p<<"/"<<f.q<<")";
return șir;}
friend istream& operator >>(istream& s,fracție& f);
};
fracție simplific(fracție f)
{if (f.q<0) {f.q=-f.q; f.p=-f.p;}
long d=cmmdc(f.p,f.q);
f.p/=d; f.q/=d;
return f;}
fracție fracție::simplific( )
{if(q<0) {q=-q; p=-p;}
long d=cmmdc(p,q);
p/=d; q/=d;
return *this;}
inline fracție operator +(fracție& f1, fracție& f2)
{fracție f;
f.p=f1.p*f2.q+f2.p*f1.q;
f.q=f1.q*f2.q;
return simplific(f);}
inline fracție operator +(fracție f, long n)
{f.simplific( );
f.p=f.p+n*f.q;
return f;}
inline fracție fracție::operator -(fracție& f2)
{fracție f;
f.p=p*f2.q-q*f2.p;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
f.q=q*f2.q;
return simplific(f);}
inline fracție operator -(long n, fracție f)
{f.simplific( );
return fracție(n*f.q-f.p,f.q);}
inline fracție operator -(fracție f, long n)
{f.simplific( );
return fracție(f.p-n*f.q,f.q);}
inline fracție operator *(fracție& f1, fracție& f2)
{return simplific(fracție(f1.p*f2.p, f1.q*f2.q));}
inline fracție operator *(fracție& f, long n)
{return simplific(fracție(f.p*n,f.q));}
inline fracție operator *(long n, fracție& f)
{return simplific(fracție(f.p*n, f.q));}
inline fracție operator /(fracție& f1, fracție& f2)
{return simplific(fracție(f1.p*f2.q, f1.q*f2.p));}
inline fracție operator /(fracție& f, long n)
{return simplific(fracție(f.p,f.q*n));}
inline fracție operator /(long n ,fracție& f)
{ return simplific(fracție(f.q*n,f.p));}
inline int operator = =(fracție& f1, fracție& f2)
{
f1.simplific( );
f2.simplific( );
return (f1.p= =f2.p && f1.q==f2.q);}
inline int operator !=(fracție& f1, fracție& f2)
{f1.simplific( );
f2.simplific( );
return (f1.p!=f2.p| | f1.q!=f2)q;}
inline fracție& fracție :: operator +=(fracție& f)
{p=p*f.q+q*f.p;
q*=f.q;
simplific( );
return *this;
}
inline fracție& fracție::operator +=(long n)
{ simplific( );
p+=n*q;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
return *this;}
inline fracție& fracție::operator -=(fracție& f)
{p=p*f.q-q*f.p;
q*=q*f.p;
simplific( );
return *this;}
inline fracție& fracție::operator -=(long n)
{simplific( );
p-=n*q;
return *this;}
inline fracție& fracție::operator *=(fracție& f)
{p*=f.p;
q*=f.q;
simplific( );
return *this;}
inline fracție& fracție::operator *=(long n)
{p*=n;
simplific( );
return *this;}
inline fracție& fracție::operator /=(fracție& f)
{p*=f.q;
q*=f.p;
simplific( );
return *this;}
inline fracție& fracție::operator /=(long n)
{q*=n;
simplific( );
return *this;}
istream& operator >>(istream& s,fracție& f)
{while ((s>>f.p)==0){
s.clear( );
s.ignore( );}
while((s>>f.q)==0){
s.clear( );
s.ignore( );}
s.ignore( );
return s;}
```

https://biblioteca-digitala.ro / https://unibuc.ro

2.14 Aplicații C si C++ [12, 46, 47, 58]

In acest paragraf exemplificăm implementarea în C şi C++ a metodelor şi tehnicilor de elaborare a algoritmilor prezentate în prima parte, capitolul II.

2.14.1 Recursivitate [47]

2.14.1.1 Probleme rezolvate

1°. Un program C++ care citeşte de la tastatură un şir de caractere încheiat cu "." şi îl afişează inversat.

```
#include <iostream.h>
void invers()
{
 char c;
 c=cin.get();
 if (c!='.')
   {
     invers();
     cout<<c;
   }
}
void main()
{
 cout<<"Introdu un sir de caractere incheiat cu ."<<endl;
 invers();
}
```

2°. Fotografia alb-negru a unor obiecte este dată sub forma unui tablou a, bidimensional patratic de dimensiune n cu elemente în {0,1}. Elementul 1 corespunde poziţiilor ce fac parte din obiect. Să se determine numărul obiectelor din fotografie şi numărul elementelor 1 ce compun fiecare obiect.

Rezolvare. Vom borda tabloul cu elemente nule. Parcurgem apoi tabloul şi cât timp găsim elemente 1, alegem un astfel de element. Construim funcţia recursivă **int figura (int m, int n)** care înlocuieşte cu –1 fiecare element 1 ales şi toate elementele 1 ce fac parte din acelaşi obiect cu cel ales. Funcţia întoarce numărul poziţiilor 1 ce aparţin obiectului curent.

https://biblioteca-digitala.ro / https://unibuc.ro

Funcția **void generare()** va genera o configurație de obiecte. La început se introduce în stivă poziția (i,j), a unui element din obiectul curent, adică a[i][j]==1.

Cît timp stiva nu este vidă, extragem un element din stivă și apoi introducem în stivă pozițiile elementelor 1 vecine cu elementul extras, elemente pe care le numărăm. Această introducere este însoțită de înlocuirea în matrice a valorii 1 cu –1.

Programul este următorul:

```
#include <iostream.h>
#include <stdlib.h>
const n=8;
int a[n+2][n+2];
int figura( int i, int j)
{
 if (a[i][j]= =1)
    {
      a[i][j]=-1;
      return      1+figura(i-1,j-1)+figura(i-1,j)+figura(i-1,j+1)+figura(i,j-
1)+figura(i,j+1)+figura(i+1,j-1)+figura(i+1,j)+figura(i+1,j+1);
    }
 return 0;
}

void generare()
{
 int i, j, k, m;
 randomize();
 m=1+randomize(n*n);
 for (k=1; k<=m;  k++)
    {
        i=1+randomize(n);
        j=1+randomize(n);
        a[i][j]=1;
     }
}
void afisare()
{
 int i, j;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
 for (i=1; i<=n; i++)
    {
        for (j=1; j<=n; j++) cout<<" "<<a[i][j];
        cout<<endl;
    }
}
void main()
{
 int i, j, k=0;
 generare();
 afisare();
 for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    if (a[i][j]= =1)
     {
        k++;
        cout<<"Obiectul: "<<k<<"="<<figura(i,j)<<" elemente "<<endl;
      }
 for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) a[i][j]=-a[i][j];
}
```

Observație:
Deoarece apelurile recursive pot depăși usor capacitatea stivei vom prezenta și o variantă nerecursivă a rezolvării problemei. Am eliminat recursivitatea printr-o stivă compusă din vectorii x și y. Indicele k indică vârful stivei iar perechea (x[k],y[k]) reprezintă poziția unui element 1 din obiectul curent.

In program se înlocuie funcția recursivă **int figura(int m, int n)** prin funcția iterativă **int fig_nr(int m, int n)** iar în funcția main tot ce urmează după apelul funcției **afisare()**; prin setul de instrucțiuni:

```
k=0;
for (i=1; i<=n; i++)
 for (j=1; j<=n; j++)
  if (a[i][j]= =1)
    {
        k++;
        cout<<"obiectul:"<<k<<"="<<fig_nr(i,j)<<" elemente "<<endl;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
    }
```

Funcția iterativă are următorul aspect:

```
int fig_nr (int i, int j)
{
  int *x=new int[n], *y=new int[n], k=0, t=0;
  x[k]=i; y[k]=j; a[i][j]=-1;
  while (k>=0)
    {
        i=x[k]; j=y[k];
        k--;
        t++;
        if (a[i-1][j-1]= =1){k++; x[k]=i-1; y[k]=j-1; a[i-1][j-1]=-1;}
        if (a[i-1][j]= =1){k++; x[k]=i-1; y[k]=j; a[i-1][j]=-1;}
        if (a[i-1][j+1]= =1){k++; x[k]=i-1; y[k]=j+1; a[i-1][j+1]=-1;}
        if (a[i][j-1]= =1){k++; x[k]=i; y[k]=j-1; a[i][j-1]=-1;}
        if (a[i][j+1]= =1){k++; x[k]=i; y[k]=j+1; a[i][j+1]=-1;}
        if (a[i+1][j-1]= =1){k++; x[k]=i+1; y[k]=j-1; a[i+1][j-1]=-1;}
        if (a[i+1][j]= =1){k++; x[k]=i+1; y[k]=j; a[i+1][j]=-1;}
        if (a[i+1][j+1]= =1){k++; x[k]=i+1; y[k]=j+1; a[i+1][j+1]=-1;}
    }
  delete x; delete y;
  return t;
}
```

2.14.2 Divide et impera [47, 58]

2.14.2.1 Descriere

Dacă $a=(a_1,\ldots,a_n)$ este vector ce trebuie prelucrat, presupunem că pentru orice $p,q\in\{1,\ldots,n\}$ cu $p<q$, există m încât $p\le m<q$ și că prelucrarea secvenței $(a_p,\ldots,a_q)$ se poate face prelucrând secvențele $(a_p,\ldots,a_m)$ și $(a_{m+1},\ldots,a_q)$ și în final combinăm rezultatele pentru a obține prelucrarea secvenței dorite. De asemenea, presupunem că dacă lungimea secvenței de prelucrat este sub o valoare fixată, t, atunci prelucrarea secvenței nu mai necesită împărțirea în subprobleme ci se poate face direct. Algoritmul asociat metodei se poate sintetiza acum sub forma:

```
        procedure divetimp(p,q)
        {
         if q-p<t then rezultat←prel(p,q)
                  else  {
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                    m←divide(p,q)
                    rezultat1←divetimp(p,m)
                    rezultat2←divetimp(m+1,q)
                    rezultat←combina(rezultat1, rezultat2)
                  }
        return rezultat
      }
```

Procedurile:

- divetimp(p,q), prelucrează secvența (a_p,...,a_q) conform strategiei divide et impera
- prel(p,q), prelucrează direct secvența (a_p,...,a_q)
- divide(p,q), întoarce un index m ce împarte secvența (a_p,...,a_q) în subsecvențele (a_p,...,a_m) si (a_{m+1},...,a_q)
- combina(rezultat$_1$, rezultat$_2$), combină rezultatele prelucrării secvențelor (a_p,...,a_m) si (a_{m+1},...,a_q) pentru a obține rezultatul final al prelucrării secvenței (a_p,...,a_q)

2.14.2.2 Probleme rezolvate:

1^0. Să se determine elementul minim şi elementul maxim dintr-un şir de n elemente.

Rezolvare.

Fie a[n] un vector ce conține cele n elemente de prelucrat, S=(a[1],...,a[n]) secvența ce trebuie împărțită în două subsecvențe. Notăm cu max(S) elementul maxim din S şi cu min(S) elementul minim din S.

Inițial p=1 si q=n şi subsecvențele sunt:

S_1=(a[1],...,a[int((p+q)/2)])

S_2=(a[int((p+q)/2)+1],...,a[q])

Astfel max(S)=max(max(S_1), max(S_2)) şi

min(S)=min(min(S_1), min(S_2))

Dacă q-p$\in$ {0,1}, adică subsecvența are cel mult două elemente maximul şi minimul subsecvenței se determină direct, iar în caz contrar subsecvența va avea un rezultat obținut prin combinarea rezultatelor a două subsecvențe ale sale.

Cu acestea va rezulta următorul program:

```
#include <iostream.h>
#include <stdlib.h>
void initial( int a[], int n)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
{
 randomize();
 cout<<"\n(";
 for (int i=0; i<n; i++) cout<<(a[i]=random(100))<<",";
 cout<<"\b)";
}

void max_min(int a[], int p, int q, int& max, int& min)
{
 switch(q-p)
    {
       case 0: max=min=a[p];
               break;
       case 1: if (a[p]<a[q])
               {max=a[q]; min=a[p];}
               else { max=a[p]; min=a[q];}
               break;
       default: int m=(p+q)/2;                    ← divide
                max_min(a, p, m, max, min);
                int max1, min1;
                max_min(a, m+1, q, max1, min1);
                if (max<max1) max=max1;          ← combină soluțiile
                if (min>min1) min=min1;
    }
}

void main()
{
 const n=10;
 static int a[n];
 int max, min;
 initial(a,n);
 max_min(a, 0 ,n-1, max, min);
 cout<<"max="<<max<<", min="<<min;
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

2^0. Algoritmul lui Hoare pentru sortarea crescătoare a elementelor unui vector.

Rezolvare:

Presupunem că prin interschimbări de elemente putem face ca o secvență $a_p,\ldots,a_q$ cu $p<q$, să poată fi pusă sub forma: $x_p, x_{p+1},\ldots, x_m,\ldots, x_q$ atfel încât pentru orice $i=p,\ldots, m-1$ să avem $x_i \le x_m$ și pentru orice $i=m+1,\ldots, q$ avem $x_m \le x_i$. In acest caz pentru a sorta crescător $(a_p,\ldots,a_q)$ vom sorta $(x_p,\ldots,x_{m-1})$ și $(x_{m+1},\ldots,x_q)$ independent una de alta. Aici nu mai este nevoie de etapa combinării din metoda generală.

Partiționarea secvenței $(a_p,\ldots,a_q)$ se face după următorul algoritm:

1. alegem un element oarecare al secvenței, exmplu a_p și se atribuie lui x
2. parcurgem secvența de la stânga la dreapta până ce găsim un elemnt $a_i \ge x$
3. parcurgem secvența de la dreapta la stânga până ce găsim un element $a_j \le x$
4. dacă $i<j$, interschimbăm a_i cu a_j și continuăm cu pașii 2, 3, 4
5. dacă $i \ge j$, permutăm a_p cu a_j și astfel secvența este împărțită în două subsecvențe.

Putem simplifica testele completând la dreapta șirul cu o valoare foarte mare, spre exemplu cu maxlong.

Programul care implementează algoritmul lui Hoare este următorul:

```
#include <iostream.h>
#include <iomanip.h>
#include <stdio.h>
#include <values.h>
typedef long double tip;
const INF=MAXLONG;
int partitie(tip a[], int p, int q)
{
 tip x=a[p], aux;
 int i=p, j=q+1;
 for( ; ;) {
            while (x>a[++i])
              while (x<[--j])
               if (i<j) { aux=a[i]; a[i]=a[j]; a[j]=aux;}
                        else { a[p]=a[j]; a[j]=x; return j;}
          }
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
void sortHoare(tip a[], int p, int q)
{
 if (p<q) {
                int m=partitie(a, p, q);
                sortHoare(a, p, m-1);
                sortHoare(a, m+1, q);
          }
}
void afisare( tip a[], int n)
{
 cout<<"[";
 for(int i=0; i<n; i++) cout<<a[i]<<",";
 cout<"\b]";
}
void generare(tip a[], int n)
{
 for (int i=0; i<n; i++) a[i]=random(100);
 a[n]=INF;
}
void main()
{
 const n=25;
 randomize();
 tip a[n+1];
 generare (a,n);
 afisare (a, n); cout<<endl;
 sortHoare (a, 0, n-1);
afisare (a, n);
}
```

3^0. O tablă de şah cu latura de 2^n unităţi este compusă din 2^{2n} pătrate unitare, dintre care unul este acoperit. Să se realizeze o acoperire a suprafeţei rămase cu plăci formate din trei pătrate unitare de forma 

Rezolvare:
Impărţim tabla în 4 pătrate egale cu latura de 2^{n-1} unităţi. Una din aceste pătrate conţine pătratul unitar acoperit. Acoperim 3 pătrate unitare cu placa de mai sus încât în fiecare din cele 4 pătrate să avem un pătrat uni-

https://biblioteca-digitala.ro / https://unibuc.ro

tar acoperit. Putem astfel repeta procedeul pentru fiecare din cele 4 pătrate de latură 2^{n-1} deoarece fiecare este în condițiile tablei de șah inițiale.

Implementând acest algoritm obținem următorul program:

```
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include<conio.h>
const n=16;
int a[n][n];
void divide(int i, int j, int n);          ← (i,j) colțul stânga-sus
int ex(int i, int j, int n);
void afisare() ;
void main()
{
clrscr() ;
randomize() ;
a[random(n)][random(n)]=1;
divide(0, 0, n);
afisare();
}
void divide(int i, int j, int n)
{
static int p=1 ;
if(n>1) {
        p++;
        int m=n/2;
        if(ex(i, j, m)) {
                a[i+m][j+m-1]=p;
                a[i+m][j+m]=p;
                a[i+m-1][j+m]=p;
                }
        else if(ex(i, j+m, m)) {
                        a[i+m-1][j+m-1]=p;
                        a[i+m][j+m-1]=p;
                        a[i+m][j+m]=p;
                         }
        else if(ex(i+m, j, m)) {
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                                    a[i+m-1][j+m-1]=p;
                                    a[i+m][j+m]=p;
                                    a[i+m-1][j+m]=p;
                                     }
          else if(ex(i+m, j+m, m)) {
                                        a[i+m-1][j+m-1]=p;
                                        a[i+m][j+m-1]=p;
                                        a[i+m-1][j+m]=p;
                                       }
          divide(i, j, m);
          divide(i, j+m, m);
          divide(i+m, j, m);
          divide(i+m, j+m, m);
        }
}
int ex(int i, int j, int n)
{
int s, t ;
  for(s=i ;s<i+n ;s++)
        for(t=j ;t<j+n; t++)
           if(a[s][t]) return 1 ;
  return 0;
}
void afisare()
{
int i, j;
  for(i=0 ;i<n ;i++) {
        for(j=0 ;j<n ;j++)
           cout<<setw(4)<<a[i][j];
        cout<<endl;
        }
}
```

4^0. Problema turnurilor din Hanoi

Se consideră 3 tije şi n discuri perforate, de diametre diferite, numerotate cu 1,….,n în ordinea crescătoare a diametrelor lor. Inițial discurile sunt aşezate pe prima tijă în ordinea crescătoare a razelor privind dinspre vârful tijei spre bază. Se pune problema deplasării turnului de discuri de pe

https://biblioteca-digitala.ro / https://unibuc.ro

tija 1 pe tija 2, folosind ca tijă intermediară, tija 3 şi respectând următoarele reguli:

- la fiecare mutare se deplasează un singur disc
- pe fiecare tijă, peste un disc se pot pune discuri de raze mai mici decât raza acestuia.

Rezolvare :

Dacă notăm cu a,b,c cele 3 tije, atunci mutarea celor n discuri de pe a pe b necesită mutarea a n-1 discuri de pe a pe c folosind pe b ca tijă ajutătoare, mutarea ultimului disc de pe a pe b şi în final, deplasarea celor n-1 discuri de pe c pe b folosind tija a ca tijă intermediară.

Vom lua vectorii a[], b[] şi c[] care conţin numerele de ordine ale discurilor de pe tija asociată. In a[0], b[0] şi c[0] punem numărul de discuri de pe tijele a, b şi respectiv c.

Implementarea algoritmului conduce la următorul program C++ :

```
#include<iostream.h>
#include<conio.h>
const max=25;
int a[max], b[max], c[max];
int n;
void intializare()
{
a[0]=n;
for(int i=1 ;i<=n ;i++)  a[i]=n+1-i ;
  b[0]=c[0]=0;
}
void afisare()
{
int i;
clrscr();
gotoxy(10,25);  cout<<"a";
for(i=1 ;i<=a[0] ;i++) {
                        gotoxy(10,24-i);
                        cout<<a[i];
                        }
gotoxy(20,25);  cout<<"b";
for(i=1 ;i<=b[0] ;i++) {
                        gotoxy(20,24-i);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                        cout<<b[i];
                        }
gotoxy(30,25); cout<<"c";
for(i=1 ;i<=c[0] ;i++) {
                        gotoxy(30,24-i);
                        cout<<c[i];
                        }
gotoxy(35,25); cout<<"tasteaza <C R> ";
getch();
}
void deplasare( int x[], int y[])   ← mută discul din vârful tijei x pe tija y.
{
y[0]++ ;
y[y[0]]=x[x[0]] ;
x[0]-- ;
afisare() ;
}
void hanoi( int n, int a[], int b[], int c[])
{
if(n>0) {
        hanoi(n-1, a, c, b) ;
        deplasare(a, b) ;
        hanoi(n-1, c, b, a);
        }
}
void main()
{
n=7;
initializare();
afisare();
hanoi(n, a, b, c);
}
```

5^0. O foaie de hârtie pătratică de latură $n = 2^m$ este compusă din $n^2 = 2^{2m}$ pătrățele unitare. Dintre cele n^2 pătrățele au fost înegrite p pătrățele, unde $p < n^2/5$. Să se realizeze o decupare a unei porțiuni din foaie încât să avem satisfacute condițiile :

- toate pătrățelele înegrite să fie în porțiunea decupată ;

https://biblioteca-digitala.ro / https://unibuc.ro

- în orice pătrat decupat aria pătrățelelor înegrite să fie între 1/5 şi 4/5 din aria pătratului decupat.

Soluție :

Vom reprezenta foaia printr-o matrice în care pătratelele înegrite sunt reprezentate prin 1, iar celelalte prin 0. Vom împărți pătratul inițial de latură 2^m în 4 pătrate congruente de latură 2^{m-1}.

Dar din $p < n^2/5$ rezultă $p < (n/2)^2 \cdot 4/5$ şi astfel fiecare din cele 4 pătrate deținute are înegrită o zonă mai mică decât 4/5 din aria sa.

Dintre aceste 4 pătrate , cele care au acoperită o zonă mai mare decât 1/5 din aria lor, verifică a doua restricție impusă mai sus. Restul pătratelor din descompunere care au înegrită o porțiune mai mică decât 1/5 din aria lor, se împart din nou în 4 pătrate congruente de latura $n/4 = 2^{m-2}$, s.a.m.d.

Procedând astfel putem rezolva problema prin următorul program C++ :

```
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include<conio.h>
const n=8;
int a[n][n];
void generare()     ← genereaza o configuratie de patrate negre.
{
randomize();
int i, j, k, p=n*n/5;
for(k=1 ;k<=p ;k++)  a[random(n)][random(n)]=1;
for(i=0 ;i<n ;i++) {
                for(j=0 ;j<n ;j++)  cout<<setw(2)<<a[i][j];
                cout<<endl;
                    }
}
void divide( int x, int y, int n)
{
int p=0, i, j ;
for(i=x ;i<x+n ;i++)
        for(j=y ;j<y+n ;j++)
                if(a[i][j])  p++ ;      ← numără pătratele negre
        if(p>n*n/5)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        cout<<"[("<<x+1<<","<<y+1<<")("<<x+n<<","<<y+n<<")]"<<
p<< "patrate"<<endl;
        else if(p>0) {
                        n=n/2;
                        divide(x, y, n);
                        divide(x+n, y, n) ;
                        divide(x, y+n, n) ;
                        divide(x+n, y+n, n) ;
                      }
}
void main()
{
clrscr() ;
generare() ;
divide(0, 0, n) ;
}
```

6^0. Un turneu de fotbal la care participă $n = 2^m$ echipe se termină după n-1 zile dacă se respectă condițiile următoare :

- fiecare echipă joacă cu fiecare altă echipă ;
- fiecare echipă joacă un singur meci pe zi.

Să se realizeze o programare a meciurilor turneului care să respecte aceste condiții.

Rezolvare:

La început împărțim echipele în două părți egale astfel:

$(1, 2,..., 2^{m-1}), (2^{m-1}+1,..., 2^m)$

Fiecare echipă din prima grupă va juca succesiv cu fiecare echipă din grupa a doua . Obținem o configurație de forma:

$(1, 2^{m-1}+1), (2, 2^{m-1}+2), ..., (2^{m-1}, 2^m)$

$(1, 2^{m-1}+2), (2, 2^{m-1}+3), ..., (2^{m-1}, 2^{m-1}+1)$

.....

$(1, 2^m), (2, 2^{m-1}+1), ..., (2^{m-1}, 2^m-1)$

Se repetă apoi procesul pentru cele două grupe $(1, ...,2^{m-1})$ si $(2^{m-1}+1, ...,2^m)$.

Vom păstra perechile într-o matrice a[n-1][n+1]. Pentru orice $k = \overline{0, n-2}$ numărul de coloane ale liniei k completate în ziua k se vor găsi în a[k][0].

https://biblioteca-digitala.ro / https://unibuc.ro

Procedând astfel obținem următorul program :

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
const n=8;
char a[n-1][n+1];

void divide( int p, int q, int k)     ← se prelucrează grupa (p, ..., q).
{
if(q>p) {
        int m=(p+q)/2;
        int d=m-p+1 ;
        for(int t=0 ;t<d ;t++) {
                for(int i=p ;i<=m ;i++) {
                                a[k][++a[k][0]]=i;
                                a[k][++a[k][0]]=m+1+(i-t+1)%d;
                                                permutări circulare.
                                        }
                               k++;
                                }
            divide(p, m, k);
            divide(m+1, q, k) ;
        }
}

void main()
{
clrscr() ;
divide(1, n, 0) ;
for(int i=0 ;i<n-1 ;i++) {
                        cout<<"ziua"<<i+1<<";";
                        for(int j=1 ;j<=n ;j+=2)

        cout<<"("<<int(a[i][j])<<","<<int(a[i][j+1])<<")";
                        cout<<endl;
                          }
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

2.14.3 Backtracking [46, 47]

2.14.3.1 Descriere

Denumirea metodei a fost dată de D. H. Lehmar în 1950.

Se aplică problemelor în care soluția se poate pune într-un vector $x=(x_1, \ldots, x_n)$, unde $x_k \in S_k$, $k = \overline{1,n}$, card $S_k=m_k$, vector ce optimizează o funcție criteriu $P(x_1, \ldots, x_n)$.

Prin această metodă orice vector soluție se construiește progresiv, începând cu x_1 și mergând către x_n, verificând de fiecare dată (prin funcția criteriu $P(x_1, \ldots, x_k)$) dacă vectorul ce se formează tinde spre soluție. Vectorul soluție trebuie să verifice două categorii de condiții:

- condiții explicite, care cer ca x_k să ia valori doar din S_k (exemplu $x_k=0$ sau $x_k=1$ sau $S_k=\{0,1\}$, $l_k \le x_k \le u_k$ sau $S_k=[l_k, u_k]$)
- condiții implicite (interne sau de continuare), care descriu relații între x_k și celelalte componente $x_1, \ldots, x_{k-1}$ ale vectorului, păstrând doar vectorii de la prima condiție care satisfac funcția criteriu.

Prin atribuiri sau încercări de atribuire eșuate datorita nerespectării condițiilor implicite, anumite valori din S_k sunt testate. Fie C_k mulțimea valorilor testate din S_k.

La început metoda pornește cu $C_k = \varnothing$, $k = \overline{1,n}$, iar pe parcursul aplicării metodei apar 4 situații posibile :

a) Atribuire și avansare . Acest caz apare când la pasul k avem $C_k \neq S_k$, iar valoarea aleasă $x_k \in S_k - C_k$ are proprietatea că $x_1, \ldots, x_{k-1}$, x_k satisfac condițiile interne. In această situație $C_k \leftarrow C_k \cup \{x_k\}$ și apoi se trece la componenta k+1.

b) Incercare eșuata. Acest caz apare când la pasul k, $C_k \neq S_k$, iar valoarea aleasă $x_k \in S_k - C_k$ face ca $x_1, \ldots, x_{k-1}, x_k$ să nu verifice condițiile de continuare. In acest caz $C_k \leftarrow C_k \cup \{x_k\}$ și nu se avansează la componenta k+1.

c) Revenire. Acest caz apare când $C_k=S_k$. In această situație se revine la componenta k-1, încercând alegerea altei componente $x_{k-1} \in S_{k-1} - C_{k-1}$ și reluând căutarea pentru componenta k de la început, adică făcând $C_k \leftarrow \varnothing$.

d) Revenire după construirea unei soluții. Acest caz apare când a fost construită o soluție $x = (x_1, \ldots, x_n)$. In vederea obținerii altei soluții se revine la componenta n, căutand o altă valoare pentru $x_n \in S_n - C_n$.

https://biblioteca-digitala.ro / https://unibuc.ro

Observaţie : Cazul d) este un caz particular de revenire , deoarece dacă presupunem că vectorul x are o compontă suplimentară cu indicele n+1 care ia valori în $S_{n+1} = \varnothing$, atunci când k=n+1 avem $S_k = \varnothing$, şi astfel se revine la componenta anterioară , adică componenta n. Condiţia k= =n+1 se foloseşte în practică pentru a sesiza obţinerea unei soluţii.

Evident dacă dorim toate soluţiile procedăm astfel :

- pentru k=n şi $C_n = S_n$ se revine la n-1 ;
- pentru k=n-1 şi $C_{n-1} = S_{n-1}$ se revine la n-2 ;
- .....
- pentru k=2 şi $C_2 = S_2$ se revine la componenta 1 ;
- pentru k=1 şi $C_1 = S_1$ se revine la componenta 0 ;
- pentru k=0 fapt imposibil deoarece vectorul soluţie este $x_1, \ldots, x_n$; se încheie procesul de căutare a soluţiilor.

Astfel condiţia k==0 se folseşte în practică pentru a sesiza încheierea parcursului de căutare a soluţiilor.

Algoritmul backtracking se poate sintetiza şi sub forma următoare:

```
procedure BKTI1
initializam S1, ..., Sn
 Ci ← ∅;  i =1,..., n
 k=1
 while k > 0 {
       if k=n+1 then { prelucreaza solutie x=(x1, ..., xn)
                       k ← k-1  }   ← revenire după construcţia unei
                                      soluţii.
       else if  Sk ≠ Ck then  alege xk ∈ Sk - Ck
                              Ck ← Ck ∪ {xk}
                              if (x1, ..., xk) satisfac conditiile interne then
                              k ← k+1
                              endif
                         else  Ck ← ∅
                               k ← k-1
            endif
       endif
 repeat
end
```

Daca $S_k = \{x \in Z \mid a_k \le x \le b_k\}$, $k = \overline{1,n}$, algoritmul devine:

https://biblioteca-digitala.ro / https://unibuc.ro

```
procedure BKTI2
for i =1, n
        x_i ← a_i-1
repeat
k ← 1
while k > 0
        if  k = n+1  then  prelucreaza solutie x=(x_1, ..., x_n)
                           k ← k-1
                    else if  x_k < b_k  then   x_k ← x_k+1
                                               if   x_1, ..., x_k satisfac cond interne
                                               then  k ← k+1
                                               endif
                                          else  x_k ← a_k-1
                                                k ← k-1
                               endif
        endif
repeat
end.
```

Variantele recursive ale acestor doi algoritmi se descriu folosind funcțiile recursive BKTR1, respectiv BKTR2.

```
Procedure BKTR1(k)
 If k=n+1 then prelucreaza solutia x=(x_1,...,x_n)
           else C_k←∅
                while S_k≠C_k
                      alege x_k€ S_k-C_k
                      C_k← C_k ∪{x_k}
                      if(x_1,...,x_k) satisface cond. interne then BKTR1(k+1)
                       endif
                repeat
endif
end
Procedure BKTR2(k)
if k=n+1 then prelucreaza solutia x=(x_1,...x_n)
          else for i=1,n_k
                    x_k←s_ki
                    if x_1 ,...,x_k satisfac conditiile interne then BKTR2(k+1)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
            endif
        repeat
endif
end
```

2.14.3.2 Probleme rezolvate

1°**Colorarea hărților**. Se colorează o hartă ce conține n țări folosind 4 culori încât orice două țări cu frontiera vecină să aibă culori diferite .

Rezolvare: Reprezentăm harta printr-un tablou bidimensional pătratic a de dimensiune n, în care punem a[i][j]=1 dacă țările i și j au frontiera comună și a[i][j]=0 în cazul contrar.

Scriem o soluție sub forma C=($c_1,c_2,\ldots,c_n$) unde c_i este culoarea atașată țării i.

Condițiile interne sunt:-$\forall i=\overline{1,n}$, $c_i \in \{1,2,3,4\}$

-$\forall i=\overline{1,k-1}$,dacă țara i este vecină cu țara k atunci

$c_i \neq c_k$ și aceasta pentru $\forall k=\overline{2,n}$

Cu acestea se obține programul următor:

```
#include <iosream.h>
#include<conio.h>
const n=6
const
int a[n][n]={{0,1,0,1,0,1},{1,0,1,1,1,0},{0,1,0,0,1,1},{1,1,0,0,1,1},{0,1,1,1,0,1},{1,0,1,1,1,0} };
int c[n]
void prelucrare( )
{ if (wherey( )==25){cout<<"tasteaza <ENTER>";
                      getch( );
                      clrscr( );
                     }
for(int i=0;i<n;i++) cout <<c[i]<<" ";
cout<<endl;
}
int alegeculoare(int k,int culoare)
{for(int i=0;i<k;i++)
 if (a[k][i]==1&& culoare==c[i]) return 0;
 return 1;
}
void bkt(int k)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
{if(k==n) prelucrare( );
          else for (c[k]=1;c[k]<=4;c[k]++)
              if(alegeculoare(k,c[k])) bkt(k+1);
}
void main( )
{
bkt(0);
}
```

2°.Să se construiască o matrice 5×5 conținând numerele 1,2,..,25 și respectând condițiile:

-poziția lui 1 se citește la intrare

-dacă i=1,..,24 este plasat pe pozitia (x,y) atunci i+1 poate fi plasat numai pe una din pozițiile (x+3,y), (x-3,y), (x,y+3), (x,y-3), (x+2,y+2), (x+2,y-2), (x-2,y+2), (x-2,y-2).

Rezolvare:

Folosind condițiile de continuare din enunț rezultă programul următor:

```
#include<conio.h>
#include<iostream.h>
#include<iomanip.h>
const  int n=5;
struct deplasare {
                    int x,y;
                  };
deplasare d[8]={{3,0},{-3,0},{0,3},{0,-3},{2,2},{2,-2},{-2,2},{-2,-2}};
int m=0, nrsel;
int a[n][n];
void afisare( )
{
 cout<<"solutia"<<(++m)<<endl;
 for(int i=0;i<n;i++){
                       for(int j=0;j<n;j++)cout<<setw(5)<<a[i][j] ;
                       cout<<endl;
                      }
getch( );
}
void bkt(int x0,int y0,int k)
{int x,y,i;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
a[x0][y0]=k;
if (m==nrsel) return;
if (k==n*n) afisare();
  else for (i=0;i<8;i++){x=x0 +d[i]·x;
                         y=y0+d[i]·y;
                         if (x>=0 && x<n && y<n && a[x][y]==0){
                         bkt(x,y,k+1);
                         a[x][y]=0;
                               }
   }
}
void main( )
{clrscr( );
 int x0,y0;
cout<<"pozitia lui 1(1<=x0,y0<="<<n<<"):";
cin>>x0>>y0;
--x0;--y0;nrsel=1;
bkt(x0,y0,1);
}
```

3°Să se determine toate modalitățile de descompunere a unui număr natural nenul n sub forma : $n=x_1+x_2+\ldots+x_n$, unde $0 \le x_1 \le x_2 \le \ldots \le x_n$.

Rezolvare:

Vom reprezenta fiecare descompunere sub forma unui vector cu n elemente și vom genera aceste descompuneri în ordine lexicografică pornind cu vectorul (0,....,0,n) și terminând cu vectorul (1,1,...,1). Fie $S_{k-1}=x_1+\ldots+x_{k-1}$. Numărul x_k poate fi plasat pe poziția k dacă mai există numere $x_{k+1},..,x_n$ care să verifice relațiile:

$x_k \le x_{k+1} \le \ldots \le x_n$

$S_{k-1}+x_k+\ldots+x_n=n$

Astfel condiția internă (de continuare pentru x_k) este:

$S_{k-1}+(n-k+1)x_k \le n$

De aici rezultă că vom genera numerele $x_1,x_2,..,x_{n-1}$ și apoi vom lua $x_n=n-S_{n-1}$. Evident din modul de generare $x_{n-1} \le x_n$.

Procedând astfel putem scrie următorul program C++ care rezolvă problema prin tehnica Backtracking iterativă.

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

https://biblioteca-digitala.ro / https://unibuc.ro

```
#include<iomanip.h>
const n=13;
int x[n+1];
void prelucrare( )
{
 static int k;
 if (wherey( )==25){
                        cout<<"tasteaza<CR>";
                         getch( );
                         clrscr( );
                       }
cout<<setw(3)<<(k++)<<":"<<n<<"=";
for (int i=1; i<=n;i++)
      if(x[i]) cout<<x[i]<<"+";
      cout<<"\b\n";
}
void descompunere( )
{
 static int S;
 x[1]=-1;
 int k=1;
 while(k>0)
   if(k==n){
              x[n]=n-S;
              prelucrare( );
              S-=x[--k];
            }
   else
       if(S+(n-k+1)*(x[k]+1)<=n){
          S+=++x[k];
          x[++k]=x[k]-1;}
       else S-=x[--k];
}
void main( )
{
 descompunere( );
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

Vom prezenta acum o variantă recursivă a programului de mai sus.

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
const n=13;
int x[n+1];
void prelucrare( )
{ static int k;
  if(wherey( )==25){cout<<"Tasteaza<CR>";
                    getch( );
                    clrscr( );
                   }
cout<<setw(3)<<(k++)<<":"<<n<<"=";
for(int i=1; i<=n; i++)  if(x[i]) cout<<x[i]<<"+";
cout<<"\b\n";
}
void descompunere (int k,int S )
{
   if(k==n){ x[n]=n-S;
             prelucrare( );
           }
  else for(x[k]=x[k-1];S+(n-k+1)*x[k]<=n;x[k]++)
                descompunere(k+1,S+x[k]);
}
void main( )
{clrscr( );
  descompunere(1,0);}
```

4°**Problema damelor pe tabla de şah**.

Pe o tablă de şah de dimensiune n×n se cere plasarea a n regine (dame) încât să nu se atace una pe alta conform regulilor din şah , adică să nu se afle două sau mai multe dame pe aceeaşi linie , coloană sau diagonală, oricare ar fi acestea .

Considerăm liniile tablei , coloanele sale şi cele n regine numerotate cu 1,2,..,n. De asemenea vom reprezenta o soluţie sub forma unui vector $(x_1,..x_n)$, unde x_k reprezintă coloana pe care se află regina k.

Astfel condiţiile explicite sunt : $S_k=\{1, 2,\ldots,n\}$ pentru orice k=1,..,n iar condiţiile de continuare sunt:

https://biblioteca-digitala.ro / https://unibuc.ro

x_k distincte , adică nu se află două regine pe aceeaşi coloană
$|x_k-x_i| \neq |k-i|$, adică reginele de pe liniile k şi i nu se află pe aceiaşi diagonală.
Vom scrie un program ce rezolvă această problema prin tehnica Beck-tracking atât în varianta iterativă cât şi în varianta recursivă.

A. Varianta iterativă:

```
#include<math.h>
#include<iostream.h>
const n=8;
char x[n];
int validare(int k)
{
 for(int i=0;i<k;i++)
    if (x[i]==x[k]  ||abs( x[i]-x[k])==k-i) return 0;
    return 1;
}
void prelucrare ( )
{
  for(int i=0;i<n;i++) cout <<x[i]+1<<" ";
cout<<endl;
}
void dameIT ( )
{for(int k=0;k<n;k++) x[k]=-1
  k=0;
while(k>=0)
 if (k==n){
            prelucrare ( );
            k--;}
     else{x[k]++;
       if (validare(k)) k++;
          }
        else x[k--]=-1;
  }
void main ( )
{   dameIT( ) ;
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

B. Varianta recursivă

```
#include<math.h>
#include<iostream.h>
const n=8;
char x[n];
int validare(int k)
{
 for(int i=0;i<k;i++)
    if (x[i]==x[k] ||abs( x[i]-x[k])==k-i) return 0;
    return 1;
}
void prelucrare ( )
{
  for(int i=0;i<n;i++) cout <<x[i]+1<<" ";
cout<<endl;
}
void dameREC (int k )
{
  if (k==n) prelucrare ( );
   else
  for(x[k]=0 ; x[k]<n ; x[k]++)
                        if(validare(k)) dameREC(k+1);
}
void main( )
{
 dameREC(0);
}
```

5° Fie T={1,2,...,n}.Sa se genereze P(T) – familia submulțimilor lui T.

Rezolvare: Fie A∈P(T). Vom reprezenta A prin vectorul său caracteristic a(n) definit prin : a_i=if i∈A then 1 else 0.

De asemenea putem reprezenta A printr-un vector a(n+1) în care elementele lui A vor fi plasate la sfârșitul vectorului, restul componentelor lui a fiind zero. Pentru a simplifica testele ce se vor face, vom considera că totdeauna a[0]=0 . In această reprezentare vom genera submulțimile lui T în ordine crescătoare a cardinalului lor .

https://biblioteca-digitala.ro / https://unibuc.ro

A. Varianta iterativă pentru reprezentarea submulțimilor lui T prin vectorul lor caracteristic:

```
#include<conio.h>
#include<iostream.h>
const n=7;
char a[n];
void prelucrare( )
static int nrsel=0;
if (wherey( )= =25){
        cout<<"Tasteaza <CR>";
        getch( ); clrscr( );
        cout<<++nrsel<<":{";
                for (int i=0; i<n; i++)
                if (a[i]) cout<<i+1<<" ";
        cout<<"\b}"<<endl;
}

void submultimeIT( )
{
 for (int k=0; k<n; k++) a[k]=-1;
 k=0;
 while (k>=0)
                if (k==n) {
                  prelucrare( );
                  k--;
                  }
                else if (a[k]<1) a[k++]++;
                        else a[k--]=-1;
}

void main( )
{
 clrscr( );
 submultimeIT( );
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

B. **Varianta recursivă** pentru reprezentarea submulțimilor lui T prin vectorul lor caracteristic.

```
#include <conio.h>
#include <iostream.h>
const n=8;
char a[n];
void prelucrare( )
{
 static int nrsel=0;
 if (wherey( )==25){
        cout<<"tasteaza <CR>";
        getch( );
        clrscr( );
 cout<<++nrsel<<":{";
 for (int i=0; i<n; i++)
  if (a[i]) cout<<i+1<<" ";
cout<<"\b"<<endl;
}
void submultimeREC(int k)
{
 if (k= =n) prelucrare( );
 else {
        a[k]=0; submultimeREC(k+1);
        a[k]=1; submultimeREC(k+1);
        }
}
void main( )
{
clrscr( );
submultimeREC(0);
}
```

C:Varianta recursivă pentru generarea submulțimilor lui T în ordine crescătoare a cardinalului lor:

```
#include<iostream.h>
#include<conio.h>
const  n=8;
char a[n+1];
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
void prelucrare()
{
    static  int nrsel=0;
  if (wherey()==25 )  {  cout<<"Testeaza <CR>";
                         getch();  clrscr();  }
        cout<<++nrsel<<": {";
  for (int i=1 ; i<n ;i++ )
      if (a[i])  cout<<int(a[i])<<" ";
  cout<<"\b"<<endl;
}
void submultime(int k)
{
   if (k==n+1) prelucrare();
      else  for (a[k]=a[k-1]? a[k-1]+1 :0;  a[k]<=k; a[k]++)
                                          submultime(k+1);
}
void main()
{
   clrscr();
   submultime(1);
}
```

6°. Fie T ={1,2,...,n}. Să se genereze toate partițiile lui T. O partiție a lui T se va reprezenta printr-un vector x cu n componente în care x[i] reprezintă numărul blocului ce conține elementul i∈T. Pentru a evita repetarea partițiilor generate la un moment dat, vom impune condițiile ca x[i] să ia doar valorile 1,2,..i și aceasta pentru orice i= $\overline{1,n}$.
Vom prezenta atât varianta iterativă cât și cea recursivă a algoritmului de rezolvare a problemei.

A. Varianta iterativă a generării partițiilor unei mulțimi finite:

```
#include<iomanip.h>
#include<conio.h>
const int n=5;
int x[n];
int maxim(int k)       //dă nr. claselor partiției curente pentru mulțimea 1,2,..k-1
{
   int i,m=0;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
   for (i=0;i<k;i++)
        if (x[i]>m) m=x[i];
   return m;
}
void prelucrare ()
{
  static int nrsel=0;
   if (wherey()==25)   {  cout<<"Testeaza <CR>";
                          getch(); clrscr();
   cout<<setw(3)<<;++nrsel<<":";
   int m=maxim(n);
   for (int k=1;k<=m;k++)
        {   cout<<"(";
            for (int i=0;i<n;i++)
                 if ( x[i]==k)  cout<<i+1<<",";
            cout<<"\b}∪";
        }
   cout<<"\b\b"<<endl;
}
void partIT()
{
   int k=0;
   while (k>=0)
          if (k==n)  {     prelucrare();
                           k--;
                     }
          else  if (x[k]<=maxim(k))  x[k++}++;
                else x[k--]=0;
}

void main ()
{  clrscr();
    partIT();
}
```

B. Varianta recursivă a generării partițiilor unei mulțimi finite:

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

https://biblioteca-digitala.ro / https://unibuc.ro

```
#include<iomanip.h>
const int n=5;
int x[n];
void prelucrare (int b)                    //b=nr. blocurilor partiției curente
                                           pentru mulțimea 1,2,..k-1
{
  static int nrsel=0;
   if (wherey()==25)   {  cout <<"Testeaza <CR>";
                          getch();clrscr();
                       }
cout<<setw(3)<<++nrsel<<":";
for (int k=1; k<=b; k++)
      {   cout<<"{";
          for ( int  i=0; i<n; i++)
                if (x[i]==k)  cout<<i+1<<",";
          cout<<"\b}∪";
        }
  cout<<"\b\b"<<endl;
}
void partREC( int k,int b)
{
   if (k==n)  prelucrare(b);
   else {   for (x[k]=1; x[k]<=b; x[k]++)  partREC (k+1,b);
            partREC (k+1,b+1);
        }
}
void main()
{ clrscr();
   partREC(0,0);
}
```

7°. Fie T={1,2,..,n} şi m≤n fixat. Să se genereze toate submulțimile lui T de câte m elemente cu proprietatea că oricare două submulțimi diferă prin natura elementelor lor:

Rezolvare: Se ştie că numărul submulțimilor cu proprietatea din enunț este C_n^m

https://biblioteca-digitala.ro / https://unibuc.ro

Deoarece în cadrul submulțimilor nu contează ordinea elementelor vom genera vectorul a cu m elemente în care elementele verifică relațiile $1 \le a_1 < a_2 < \ldots < a_m \le n$,unde pentru orice $i=\overline{1,m}$, $a_i \in T$.

Pentru a uşura testele vom borda vectorul a(m) cu componentele a[0] şi a[m+1].

Datorită condiției impuse între componentele vectorilor generați se observă că pentru orice $k=\overline{2,n}$, elementele de pe poziția k sunt: a[k-1]+1, a[k-1]+2,...,n-m+1

Astfel :S_1={1,2,...,n-m+1} iar S_k ={a[k-1]+1, a[k-1]+2,...,n-m+k} pentru $k=\overline{2,m}$.

A. Varianta iterativă a generării submulțimilor:

```
#include<conio.h>
#include<iostream.h>
  const n=7, m=5;
  char a[m+2];
   void prelucrare()
   { static int nrsel=0;
      if (wherey()==25)  {   cout<<"Testeaza <CR>";
                             getch(); clrscr();
                         }
 cout<<++nrsel<<":{";
 for (int i=1; i<=m; i++)   cout<<int(a[i])<<" ";
 cout<<"\b}"<<endl;
   }
void combinIT()
{
   a[1]=0;
   int k=1;
   while (k>0)
      if (k==m+1)    {   prelucrare();
                         k--;
                     }
  else if (a[k]<n-m+k)  {   a[k++]++;
                            a[k]=a[k-1];
                        }
       else k--;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
}
void main()
{ clrscr();
   combimIT();
}
```

B. Varianta recursivă a generării submulțimilor:

```
#include<iostream.h>
#include<conio.h>
const n=7, m=4;
char a[m+1]
void  prelucrare()
{   static int nrsel=0;
   if (wherey()==25)   {   cout<<"Testeaza<CR>";
                           getch(); clrscr();
                       }
   cout<<++nrsel<<":{";
   for (int i=1;i<=m; i++)   cout<<int(a[i])<<".";
   cout<<"\b"<<endl;
}
void combinREC(int k)
{
   if (k==m+1 )  prelucrare();
   else  for (a[k]=a[k-1]+1;  a[k]<=n-m+k; a[k]++)     combinREC(k+1);
}
void main()
{ a[0]=0;
combinREC(1);
}
```

2.14.4 Metoda Greedy [12, 47]

2.14.4.1 Descriere

Stim că metoda se aplică în general problemelor în care se dă o mulțime A cu n elemente şi se cere o submulțime B care să îndeplinească anumite condiții precizate printr-o funcție de selecție. Metoda nu determină toate soluțiile posibile şi apoi să aleagă pe cea optimă conform

https://biblioteca-digitala.ro / https://unibuc.ro

funcției de selecție ci ea alege pe rând câte un element urmărind să-l "înghită" eventual în soluția optimă, şi de aici denumirea metodei (greedy=lacom).

Astfel metoda furnizează o soluție posibilă şi nu totdeauna optimă. Dacă se cere optimul, atunci algoritmul ce foloseşte această metodă trebuie însoțit de demonstrarea faptului că soluția generată este optimă.

Dacă vom considera o funcție **select** care să furnizeze elementul ce trebuie înghițit în soluție şi să țină evidența elementelor neselectate încă, atunci pseudocodul algoritmului ce foloseşte această metodă ar putea fi:

```
procedure greedy (A,n)
B←∅
for k=1,n
      x←select (A,k)
      if posibil (B,x) then B←adaug (B,x)
      endif
repeat
end.
```

Funcția **posibil** verifică dacă x satisface condițiile funcției de selecție, iar **adaug** construieşte $B \leftarrow B \cup \{x\}$.

2.14.4.2 Probleme rezolvate:

Vom exemplifica metoda Greedy pe problema arborelui parțial de cost minim (APM) într-un graf conex cu muchii marcate.

Se cunosc doi algoritmi clasici de rezolvare a acestei probleme şi anume: algoritmul lui Prim şi algoritmul lui Kruskal.

1 Algoritmul lui Prim porneşte construcția unui APM de la un nod oarecare al grafului şi necesită următorii paşi:

P1: Se inițializează mulțimea muchiilor selectate cu ∅ şi mulțimea nodurilor selectate cu un nod oarecare al grafului.

P2: Dintre muchiile neselectate se alege una din muchiile de cost minim ce are o singură extremitate într-unul din nodurile arborelui deja construit.

P3: Se repetă **P2** până ce se aleg n-1 muchii (n este numărul nodurilor grafului marcat).

Pentru implementarea în C++ a acestui algoritm vom numerota vârfurile grafului, vom reține extremitățile muchiilor în matricea m de dimensiune

https://biblioteca-digitala.ro / https://unibuc.ro

nm x 2 (nm numărul muchiilor) şi vom reține costurile celor nm muchii în vectorul w.

Pentru a implementa pasul **P2** al algoritmului vom folosi funcția **int alege (int k)** care necesită doi vectori ajutători : v cu n componente şi a cu nm componente definiți astfel:

- v[i]= if i se află în arborele deja construit then 1 else 0
- inițial a=(0,1,...,nm-1) memorează la fiecare pas k al algoritmului muchiile neselectate încă. Ele se vor afla în componentele a[0],a[1],...., a[nm-k].

Programul C++ asociat este următorul:

```
#include<conio.h>
#include<iostream.h>
#include<values.h>
const  int n=5;
const int nm =8;
int m[nm][2]={{0,1},{0,3},{0,4},{1,2},{1,3},{1,4},{2,3},{3,4}};
int w[nm]={12,21,32,5,11,51,14,4};
int x[n-1];                          //vector soluție
 int* initializare (int n)
 {
   int* x=new int[n];
   for (int i=0;i<n; i++)   x[i]=i;
   return x;                         //întoarce un vector inițializat
 }
 int alege()
 {
 static int* a= initializare(nm)
 static int v[n]={1};    //căutarea începe de la nodul 0 , adică v=(1,0,...,0)
 static int nmn=nm;
 int costmin=MAXINT,i,j;
 for (i=0;i<nmn;i++)   {    int p=m[a[i]][0],q=m[a[i]][1];
                                  if(v[p]+v[q]==1&&w[a[i]]<costmin){
                                                costmin=w[a[i]];
                                                j=i;}
                           }
 i=a[j];
v[m[i][0]]=v[m[i][1]]=1;                  //alege o muchie
a[j]=a[--nmn];
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
return i;
}
int Prim()
{
   int cost=0;
   for (int k=0;k<n;k++)    {  x[k]=alege (k);
                               cost+=w[x[k]];          //alege n-1 muchii
                            }
   return  cost;
}
void main()
{
  clrscr();
  cout<<"Graful initial:"<<endl;
  for (int i=0;i<nm;i++)
        cout<<"("<<m[i][0]<<","<<m[i][1]<<") cost"<<w[i]<<endl;
  long cost=Prim();
  cout<<"Arborele de cost minim (APM):"<<endl;
  for (i=0;i<n;i++)
       cout<<"("<<m[x[i]][0]<<","<<m[x[i]][1]<<")
cost"<<w[x[i]]<<endl;
  cout<<"Cost total minim:"<<cost<<endl;
}
```

2.14.5 Programarea dinamică [47, 58]

2.14.5.1 Descriere

Se foloseşte în probleme de optim în care soluţia poate fi privită ca rezultatul unei succesiuni de decizii. Spre deosebire de metoda greedy unde o succesiune optimă de decizii se poate obţine prin luarea deciziilor locale la un anumit moment , în programarea dinamică nu se pot obţine cele mai bune decizii numai pe baza informaţiilor locale. Metoda reduce numărul seturilor de decizii prin evitarea acelor succesiuni de decizii care nu pot fi optime. Ea se bazează pe principiul optimalităţii lui R. Bellman care spune că " O soluţie optimă a unei probleme se obţine din soluţii optime a subproblemelor sale".

https://biblioteca-digitala.ro / https://unibuc.ro

Metoda programării dinamice generează secvențe de decizii și renunță la acele secvențe care conțin subsecvențe suboptimale.

Exemplu: Fie G=(X,U) un graf orientat marcat , s și t două noduri ale sale. Notăm cu L(s,t) lungimea minimă a unui drum de la s la t și prin cost(i,j) lungimea (costul)arcului ij din graf,

Pentru a găsi un drum minim de la s la t, va trebui să decidem care este al doilea vârf după s, care este al treilea vârf și așa mai departe, până ce se ajunge la vârful t astfel încât drumul obținut să aibă cea mai mică lungime. Fie $sv_1v_2 \dots v_pt$ un astfel de drum minim.

Se poate găsi un drum de lungime minimă pe următoarele trei căi:

1. Notăm $A_s=\{p \in X \mid (s,p) \in U\}$. Pentru fiecare $p \in A_s$ luăm D_p cel mai scurt drum de la p la t și astfel cel mai scurt drum de la s la t se obține luând cel mai scurt drum dintre $\{(s,D_p)\ p \in A_s\}$. Se obține următoarea relație de recurență:

$$L(s,t) = \min \{cost(s,p) + L(p,t) \mid p \in A_s\}$$

Această relație impune luarea deciziilor în ordinea $v_k, v_{k-1}, \dots, v_1$. Fiecare decizie v_i depinde de deciziile $v_{i+1}, v_{i+2}, \dots, v_k$. De aici si denumirea metodei de programare dinamică cu acces înainte.

2. Notăm $A_t= \{p \in X \mid (p,t) \in U\}$. Pentru fiecare $p \in A_t$ luăm D_p cel mai scurt drum de la s la p și astfel cel mai scurt drum de la s la t se obține luând cel mai scurt drum dintre drumurile $\{(D_p, t) \mid p \in A_t\}$. Se obține următoarea relație de recurență:

$$L(s,t) = \min \{L(s,p) + cost(p,t) \mid p \in A_t\}$$

Relația impune luarea deciziilor în ordinea $v_1, v_2, \dots ,v_k$. Fiecare decizie vi depinde de deciziile $v_1, v_2, \dots , v_{i-1}$. De aici și denumirea de programare dinamică cu acces înapoi.

3. Fie p un vârf intermediar pe drumul s, $v_1, \dots , p, w_1, w_2, \dots$, t de lungime minimă de la s la t. Atunci drumurile s, $v_1, v_2, \dots$, p si p, $w_1, \dots$, t sunt evident drumuri de lungime minimă de la s la p și respectiv de la p la t.

Considerăm că p este vârful de indice maxim de pe drumul s, $v_1, \dots , p, w_1, w_2, \dots$, t . Astfel drumurile de la s la p și de la p la t vor avea ca vârfuri intermediare doar vârfuri de indice cel mult p-1.

Dacă drumul minim de la s la t trece prin p atunci avem $L_p(s,t) = L_{p-1}(s,p) + L_{p-1}(p,t)$, pentru $p \geq 1$, unde $L_0(i,j) = cost(i,j)$. Dacă drumul minim de la s la t nu trece prin p atunci nici un nod intermediar nu are indice mai mare decât p-1. De aceea $L_p(s,t) = L_{p-1}(s,t)$.

Cu acestea putem obține următoarea relație de recurență

https://biblioteca-digitala.ro / https://unibuc.ro

$L_p(s,t) = \min\{L_{p-1}(s,t), L_{p-1}(s,p) + L_{p-1}(p,t)\}$, $p \geq 1$.

Fiecare decizie v_i depinde atât de deciziile $v_1, \ldots, v_{i-1}$ cât şi de deciziile $v_{i+1}, v_{i+2}, \ldots$.

De aici şi denumirea de programare dinamică mixtă.

2.14.5.2 Probleme rezolvate

1. Fie $a_1, a_2, \ldots, a_n$ un şir de numere reale. Să se determine cel mai lung subşir $a_{i_1}, a_{i_2}, \ldots, a_{i_p}$ al şirului dat cu proprietăţile:

a) $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_p}$

b) $i_1 \leq i_2 \leq \ldots \leq i_p$.

Rezolvare.

Notăm cu L[p] lungimea celui mai lung subşir crescător ce are ca prim termen pe a_p, p = n, n-1, ... , 1

Evident avem:

- L[n] = 1
- Pentru orice k = p+1,..,n dacă $a_p > a_k$ atunci L[p]=1
- In general L[p] = max{1+L[k] | k=p+1,...,n si $a_p \leq a_k$}

Vom construi soluţia într-un vector x cu n componente definit astfel:

Pentru p = n,n-1,..., 1 luăm

x[p] = 0 daca L[p] = 1 sau

x[p] = k daca L[p] = 1+L[k]

Relaţia x[p] = 0 spune că elementul a_p nu mai are succesori în subşirul crescător iar x[p] = k spune că a_k este succesorul lui a_p.

Vom mai lua L[0] = max{L[1], ... , L[n] }= $L[i_{max}]$ si x[0] = i_{max}.

Cu aceste pregătiri se obţine următorul program C++:

```
#include<iostream.h>
#include<stdlib.h>
const int n=10;
int a[n+1], x[n+1];
int subsir(int a[], int n, int x[])
{
int L[::n+1];
L[0]=L[n]=1;
x[0]=x[n]=0;
for (int p=n-1;p>=1;p--){
        x[p]=0;L[p]=1;
        for (int k=p+1;k<=n;k++)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                if(a[p]<=a[k] && L[p]<1+L[k]){
                        L[p]=1+L[k];
                        x[p]=k;
        if(L[p]>=L[0]){
                L[0]=L[p];
                x[0]=p;
                }
        }
int i=x[0]; p=1;
while(x[i]) x[p++]=i=x[i];
return p;}
void generare( )
{
        randomize( );
        for(int i=1,i<=n;i++) cout<<(a(i)=random(15))<<" ";
        cout<<endl;
}
void afisare(int a[], int x[], int p)
{
        for (int i=0; i<p; i++) cout<<a[x[i]]<<" ";
        cout<<endl;
}
void main ( )
{
        generare();
        int p=subsir(a,n,x);
        afisare(a,x,p);
}
```

2. Fie G=(X,U) un digraf pe n nivele cu arce marcate. Să se determine un drum de cost minim de la un nod inițial s la un nod terminal t.

Rezolvare

Din definiția grafului pe n nivele rezultă că X admite o partiție din n blocuri $V_1, V_2, \ldots, V_n$ unde

$V_1 = \{s \in X \mid d^-s=0\}$ submulțimea nodurilor inițiale

$V_n = \{t \in X \mid d^+t=0\}$ submulțimea nodurilor terminale.

Pentru orice k=2,..,n-1 avem:

$V_k=\{y \in X \mid$ există $x \in V_{k-1}$ astfel încât $(x,y) \in U\} \cup \{y \in X \mid$ există $x \in V_{k+1}$ astfel încât $(y,x) \in U\}$

https://biblioteca-digitala.ro / https://unibuc.ro

Pentru orice k=1,...,n avem (V_k xV_k) $\cap$ U = Ø (nu există arce între nodurile unui acelaşi bloc).
Se observă că orice drum de la s$\in V_1$ la t$\in V_n$ se obţine ca rezultat a n-2 decizii. Decizia de ordin k=1,...,n-2 necesită găsirea unui nod din V_{k+1} care să fie pe drumul optim. Pentru nivelul k luăm toate drumurile de cost minim de la vârfurile v$\in V_k$ la t. Dacă notăm prin cost(k,v) costul drumului minim de la nodul k la nodul v şi dacă folosim metoda accesului înainte obţinem relaţia:

cost(k,v) = min {cost(v,w) + cost(k+1, w) | w$\in V_{k+1}$ si (v,w) $\in$U}

unde cost(n-1,v) = if (v,t) $\in$U then cost(v,t) else +∞.
In acest caz drumul de lungime minimă se obţine luând D(k,v)=w, care intervine în relaţia de mai sus.

Pentru implementarea in C++ a algoritmului vom impune o ordine a vârfurilor sub forma:

- vârful s are indicele 1
- indicele vârfurilor din V_k sunt mai mari decât indicii vârfurilor din V_{k-1}.

De aici rezultă că vectorii cost şi D se pot calcula în ordinea n-1, n-2,...,1
.
Graful pe nivele va fi reprezentat prin două liste:

- Ld[i] ce conţine lista descendenţilor direcţi ai nodului i
- c[i] ce conţine lista costurilor ataşate arcelor de forma (i,j) $\in$U cu j din Ld[i].

Programul în C++ este următorul:

```
#include<iostream.h>
const float infinit = 1.e10;
void digraf(int& nv, int**& Ld, float**& c)
{
        cout<<"Numarul varfurilor digrafului: ";
        cin>>nv;
        Ld=new float*[nv-1]-1;
        c=new float*[nv-1]-1;
        for(int i=1; i<nv;i++){
                int nd;
                cout<<"Varful"<<i<<". nr. descendenti:";cin>>nd;
                Ld[i]=new int[nd+1];
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                c[i]=new float[nd]-1;
                Ld[i][0]=nd;
                for(int j=1;j<=nd;j++){
                        cout<<"arcul"<<i<<"- - >";cin>>Ld[i][j];
                        cout<<"cost:";cin>>c[i][j]
                                        }
                                }
}
float drumscurt(int nv, int** Ld, float** c, int*& x, int& n)
{
float* cost=new float[nv]-1                     // cost[1],...,cost[nv]
int* D=new int[nv-1]-1;                         //D[1], ...,D[nv-1]
cout<<"Numar de blocuri: ";cin>>n;
x=new int[n]-1;
cost[nv]=0;
for (int k=nv-1; k>=1; k--){
        cost[k]=infinit;
        for(int j=1; j<=Ld[k][0];j++){
                int r =Ld[k][j]                 //descendent
                float L=c[k][j]+cost[r];
                if(L<cost[k]){
                        cost[k]=L; D[k]=r ;}
                        }
                }
        x[1]=1; x[n]=nv;
        for(int j=2; j<n;j++) x[j]=D[x[j-1]];
        return cost[1];
}
void main()
{
int n, nv;
int** Ld; int* x; float** c;
digraf(nv, Ld,c);
float cost = drumscurt(nv, Ld, c, x, n);
cout<<"Drumul minim: ";
for(int i=1; i<=n; i++) cout<<x[i]<<",";
cout<<"\b cost: "<<cost<<endl;
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

3. Fie n $\geq$ 3 matrice, A_i de dimensiune d_i x d_{i+1}, oricare ar fi i=1,...,n. Să se determine produsul A_1 x A_2 x ... x A_n folosind un număr minim de înmulțiri.

Rezolvare

Pentru două matrice A, B de dimensiuni m x n şi respectiv n x p se fac m x n x p înmulțiri.

Notăm m(i,j) – numărul minim de înmulțiri necesare pentru a calcula produsul A_i x A_{i+1} x...x A_j.

Evident au loc relațiile:

a. m(i,i) = 0 , oricare ar fi i=1,..., n
b. m(i, i+1) = $d_i d_{i+1} d_{i+2}$, oricare ar fi i=1,..., n-1
c. m(i,j)=min{m(i,k) + m(k+1, j) + $d_i d_{k+1} d_j$} pentru k=i,...,j-1.

La început vom pune m(i,i)=0, pentru orice i=1,...,n. Vom determina apoi pentru fiecare i=1,...,n-1 valorile m(i,j) pentru care j-i=1, ceea ce se poate deoarece în calculul lui m(i,j) intră doar valorile m(i,k) şi m(k+1,j) pentru care diferența indicilor este strict mai mică decât k, valori care sunt deci calculate.

Dacă în m(j,i) memorăm şi indicele k pentru care se realizează minimul m(i,j), cea ce înseamnă că produsul A_i x ... x A_k x ... x A_j se face în ordinea (A_1 x ... x A_k) x (A_{k+1} x ... x A_j), vom putea găsi şi ordinea în care trebuie înmulțite matricele pentru a avea un număr minim de înmulțiri.

Folosim doi vectori PS si PD în care pe pozitia i punem numărul de paranteze ce se deschid în fața lui A_i şi respectiv numărul de paranteze ce se închid după A_i, oricare ar fi i=1,...,n.

Programul în C++ care rezolvă această problemă are forma următoare:

```
#include<iostream.h>
#include<values.h>
const dim=20;
int d[dim];                    // vectorul conține dimensiunile matricelor
long m[dim][dim];
int PS[dim], PD[dim];
const long infinit = MAXLONG;
void produs (int n)
{
int i, k;
for(int l=1; l<n;l++)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        for(i=0;i<n-1;i++){
                int j=i+1;
                long min=infinit;
                int r;
                for (k=1; k<j; k++){
                        long t=m[i][k]+m[k+1][j]+d[i] *d[k+1]*d[j+1];
                        if(t<min){
                                min=t;
                                r=k;
                                }
                        }
                m[i][j]=min;
                m[j][i]=r;
                }
}
int citire( )
{
cout<<"Introdu sirul dimensiunilor incheind cu \".\":";
int n=0;
while ((n<dim)&&(cin>>d[n])) n++;
cin . clear();
cin . ignore(80,"\n");
return n-1;
}
void paranteze(int i, int j)
{
switch(j-i)
{
        case 0:break;
        case 1:PS[i]++;
                PD[j]++;
                break;
        default: PS[i]++;
                 PD[j]++;
                 int k=m[j][i];
                 paranteze (i,k);
                 paranteze (k+1,j);
        }
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
}
void  afisare(int n)
{
int p;
        for(int i=0; i<n;i++){
                for(p=1; p<=PS[i];p++) cout<<"(";
                cout<<"A"<<i+1<<" ";
                for(p=1; p<=PD[i]; p++) cout<<")";
                }
        cout<<"=M("<<d[0]<<","<<d[n]<<")\n";
        cout<<"Se fac "<<m[0][n-1]<<"inmultiri"<<endl;
}
void main()
{
int n=citeste();
produs(n);
paranteze(0,n-1);
afisare(n);
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL III Implementarea în C++ a structurilor de date şi a operaţiilor cu fişiere

3.1 Structuri de date de tip listă [12, 47, 90]

Prin listă înţelegem o succesiune finită de elemente ce aparţin unei mulţimi de date, esenţială fiind ordinea în care apar elementele.

Ne interesează să ştim primul şi ultimul element al unei liste precum şi ce element precede şi ce element urmează un anumit element dat al listei.

Ne interesează să putem:

- accesa un element al unei liste, dat prin valoarea sa sau prin numărul său de ordine în listă ;
- elimina un element din listă;
- reordona elementele unei liste după un anumit criteriu;
- concatena două sau mai multe liste;
- descompune o listă în două sau mai multe liste.

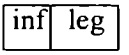
Listele se pot memora prin alocare secvenţială (statică) sau prin alocare înlănţuită (dinamică).

Memorarea prin alocare secvenţială conduce la plasarea elementelor listei în locaţii succesive, conform ordinii lor în listă. Accesul la elemente se face prin variabile de indexare.

În memorarea prin alocare înlănţuită fiecare element al listei se înlocuie cu o celulă sau nod formată din două părţi :

- partea de informaţie;
- partea de legătură, în care se află adresa următorului element al listei.

Partea de legătura a ultimului element al listei primeşte valoarea NULL indicând că după acesta nu mai urmează alt element al listei.

Celulele se reprezintă sub forma : 

iar legăturile între ele prin săgeţi → . Şi în acest caz trebuie memorată adresa de bază a listei (adresa primului element). Dacă în partea de legătură a ultimului element dintr-o listă vom pune adresa primului element al său obţinem o listă circulară. Memorarea înlănţuită elimină

https://biblioteca-digitala.ro / https://unibuc.ro

necesitatea deplasării elementelor în operatiile de inserare şi scoatere de elemente în şi din listă.

În C++ putem defini tipuri de date care se autoreferă (recursive) sub forma din exemplul următor :

```
struct nod { int inf;
             struct nod *leg; };
```

Crearea unei liste cu elemente de tip nod presupune folosirea variabilelor cu alocare dinamică, variabile care se nasc şi dispar în cursul execuţiei programului, la cererea programatorului prin folosirea funcţiilor **new** şi **delete**.

Exemplu :

```
void main ( )
{ nod *p,*q;
p=new nod;
p →inf =5;
p →leg =0;
q=p;                    ( q indică aceiaşi celulă ca şi p)
q=new nod;
q →inf = p →inf;        ( copiază informaţia din celula indicată de p
                          în celula indicată de q)
*q=*p;                  ( copiază conţinutul celulei indicată de p în
                          celula indicată de q)
p →leg = q;             ( în partea de legătura a variabilei indicată de
                          p trece adresa variabilei desemnate de q)
q →leg = 0;             ( partea de legatură a variabilei indicate de q
                          pointează către o zonă de memorie nulă)
}
```

Tipurile **class** ca şi cele **struct** sunt recursive (auto-referibile) .

3.1.1 Stive [47, 90]

Prin stivă înţelegem o listă liniară în care inserarea şi scoaterea unui element se face totdeauna la acelaşi capat al listei, numit **vârful stivei.** Celălalt capăt al listei se numeşte **baza stivei** .

3.1.1.1 Alocarea secvenţială a unei stive [47]

În acest tip de memorare, elementele stivei sunt puse într-un vector s[ ] cu **nmax**_componente, unde nmax este capacitatea maximă a

https://biblioteca-digitala.ro / https://unibuc.ro

stivei. Dacă **nel** reprezintă numărul efectiv al elementelor din stivă, atunci s[nel] este elementul din vârful stivei.

În cele ce urmează vom construi **clasa stiva** care conține operațiile de *inserare, ştergere*, etc. de elemente în/din stivă. Cu aceasta cream un fişier antet numit ***stivaS.h***

```
#ifndef tipinfS
        #define tipinfS int
#endif
#ifndef MAX
        #define MAX 100
#endif
class stiva
{ tipinfS *s;                          ( adresa primului element al stivei)
  int nmax;                            ( capacitatea maximă a stivei)
  int nel;                             ( numărul curent al elementelor stivei)
public:
     stiva( )                           ( constructor implicit)
    {
      nmax= MAX;
      s=new tipinfS[nmax];
      nel=0;
    }
stiva ( stiva& s1, int max=100);       ( constructor de copiere )
~ stiva( )
{
delete s;                               (destructor)
}
stiva& operator << (tipinfS inf);       (supraîncărcarea insertorului care
                                                pune inf pe stivă)
stiva& operator >> ( tipinfS& inf);    (supraîncărcarea extractorului care
                                        extrage în inf un element din stivă)
friend tipinfS virf(stiva& st);
tipinfS virf( );                       ( funcții care furnizează valoarea
                                        elementului din vârful stivei)
tipinfS operator [ ](int i)            (supraîncarcă operatorul de indexare)
{
return s[i-1];
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
int getn( )
{ return nel;
}
friend int getn( stiva& st)              ( funcţii ce întorc numărul de
                                           elemente din stivă)
{ return st.nel;
}
int getmax( )
{ return nmax;                           ( funcţii ce întorc capacitatea maximă a
                                                                    stivei)
}
friend int getmax( stiva& st)
{ return st.nmax;
}
stiva& operator =(stiva& s1);      ( supraîncarcă operatorul de atribuire)
int expand(int max);               ( expandează stiva la max elemente)
operator void*( )                  (dă un sens numelui obiectului prin
                                    supraîncărcarea operatorului void*)
{
if (nel==0) return 0;               ( stivă vidă)
if (nel==nmax) return (void*)1;     ( stivă plină)
return this;                        ( stivă nici vidă nici plină)
}
                                    (acum au sens instrucţiuni de forma:
                                         if (s)... sau if (s==(void*)1)...)
friend int operator ==( stiva& s1, int n)
{
if (s1==( void *)n) return 1;
return 0;
}                          ( acum au sens expresii de forma s==0 sau s==1)
};
stiva& stiva :: operator <<(tipinfS inf)
{
if (nel >=nmax)
                    { cout << " stivă plină ";
                      return *this;
                     }
s[ nel ++ ] = inf;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
return *this;
}
stiva& stiva :: operator >>(tipinfS& inf)
{
if (nel >0)
            inf =s[ -- nel];
            return *this;
}
tipinfS varf( stiva& st)
{
 if (st.nel) return  st.s[st.nel-1];
}
tipinfS stiva :: varf ( )
{
if (nel >0) return s[nel-1];
}
stiva :: stiva(stiva& s1, int max)        (definim constructorul de copiere)
{
s1.nel <=max ? nmax=max : nmax=s1.nmax;
nel= s1.nel;
s= new tipinfS[nmax];
for ( int i=0; i<nel; i++)
                              s[i]= s1.s[i];
}
stiva& stiva :: operator =(stiva& s1)
{
  nel= s1.nel;
  for (int i=0; i<s1.nel; i++)    s[i]=s1.s[i];
return *this;
}
int stiva :: expand(int max)
{
if ( max <= nmax) return 0;
tipinfS *s1= new tipinfS[nmax];
for ( int i=0; i<nel; i++)   s1[i]=s[i];
delete s;
s=s1;
nmax=max;
```

https://biblioteca-digitala.ro / https://unibuc.ro

return nmax;
}

Putem folosi acum fişierul ***stivaS.h*** pentru inserare de obiecte în stivă, pentru extragere de elemente din stivă, pentru listare şi pentru alte operaţii ca în exemplul care urmează.

```
#include <iostream.h>
#include " stivaS.h"
#define tipinfS float
void main( )
{ stiva S(5);
float x;
int opt;
do {
        cout<<endl;
        cout << "1: Introducere " <<endl;
        cout << "2: Extragere " << endl;
        cout << "3: Listare " << endl;
        cout << "4: Stop " << endl;
        cout << " Tastaţi opţiunea: 1,2,3,4";
    cin >> opt;
switch (opt)
    { case 1:  if (S==1) cout << " Stivă plină: " <<endl;
               else { cout << " Introdu valoarea elementului:" ;
                      cin >>x;
                      S<<x;
                    }
                    break;
      case 2 :  if (S==0) cout << "Stivă vidă " <<endl;
                else { S >>x;
                       cout <<"Am extras elementul :"<<x<<endl;
                     }
                     break;
      case 3 :  cout << "Conţinutul stivei :"<<endl;
                for (int i=1; i<= getn(S); i++) cout << S[i]<<" " ;
                cout <<endl;
                     break;
    }
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
  }
while(op!=4);
}
```

3.1.1.2 Alocarea înlănțuită a unei stive [47]

În acest tip de alocare legaturile dintre elemente merg de la elementul din vârful stivei(ultimul element) către cel de la baza stivei(primul element), aşa cum se vede pe figura următoare:

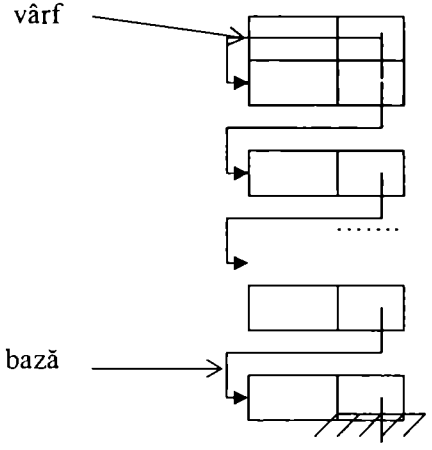


Fig. 2.1

Vom crea acum un fişier antet cu numele **stivaI.h** în care vor fi descrise 3 clase numite **nodS**, **stiva,** şi **itstiva** cu următoarele scopuri :

a) clasa nodS conține câmpurile unei celule (inf pentru partea de informație şi next pentru partea de legătură), precum şi operațiile ce se pot face cu tipul de date nodS (doi constructori şi o funcție ce întoarce valoarea memorată în inf)
b) clasa stivă încapsulează o variabilă de tip pointer către nodS ce memorează adresa elementului din vârful stivei precum şi metodele de lucru cu aceasta
c) pentru a putea defini **cicluri** peste elementele conținute într-un obiect de un anumit tip definit de utilizator, fără ca acestea să depindă de detaliile de implementare ale tipului respectiv, este necesară definirea unui aşa numit **iterator**. Un tip poate avea definite mai multe tipuri iterator diferite pentru a putea face diverse tipuri de ciclări (iterații). De asemenea, pentru un anumit obiect pot fi activi mai mulți iteratori.

https://biblioteca-digitala.ro / https://unibuc.ro

Clasa itstiva implementează un iterator pentru stive.

```
 #ifndef tipinfS
        #define tipinfS int
 #endif
class nodS
{
 tipinfS inf;
 nodS *leg;
public:
     nodS( )
        { inf =0;
          leg =0;
        }
        nodS (tipinfS infl, nodS  *p=0)
         { inf= infl;
           leg =p;
        }
        tipinfS get( )
         { return inf;
        }
friend class stiva;                    (clase prietene)
friend class itstiva;
};
class stiva
{
  nodS *cap;                           (dă adresa elementului din vârful listei)
 public :
      stiva ( )
        { cap=0;
        }
    ~ stiva ( );
   stiva& operator <<(tipinfS inf);    (insertor ce pune inf pe stivă)
   stiva& operator >>(tipinfS& inf); (extractor ce extrage elementul din
tipinfS varf( )                              vârful stivei)
     { return cap →get( );       (intoarce elementul din vârful stivei)
     }
operator void*( )          ( supraîncarcă operatorul de conversie void*
                           pentru personalizarea numelui obiectului. Funcția
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
      { return cap;      întoarce o valoare nenulă când stiva nu este vidă.
                         Astfel după declarația stiva S ; are sens instructi-
unea
                         if (S)...
       }
friend class itstiva ;   ( clasă prietenă)
};
class itstiva
{
  nodS *p;               (adresa unui element al stivei)
public :
        itstiva ( )      ( constructor implicit care permite definirea
        { p=0;                    unor tablouri de iteratori)
          }
void init(stiva& S)      ( inițializează p cu adresa elementului din vârful
   {                                stivei dată ca argument)
     p= S.cap;
    }
tipinfS operator ( )(void)   ( supraîncarcă operatorul apel fel de funcție.
  { nodS *q =p;              Primul operand este obiect de tip itstiva iar
    p= p→ leg;               al doilea este în lista de argumente, vidă în
     return q→inf;           acest caz)
  }
operator void*( )
  { return p;            ( supraîncarcă operatorul de conversie  void* pentru
    }                    personalizarea numelui obiectului. Funcția
                         întoarce o valoare nenulă dacă în stivă mai
                         sunt elemente neiterate.Astfel pentru declarația
                         itstiva it; are sens instrucțiunea if (it)...)
};
stiva :: ~stiva ( )
{ while (cap) {
                    nodS *p= cap;
                    cap= cap→leg;
                    delete p;
                  }
}
stiva& stiva :: operator <<(tipinfS inf)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
{
  cap= new nodS(inf, cap);
  return *this;
 }
stiva& stiva :: operator >>(tipinfS& inf)
{
  if (cap) {  inf= cap→inf;
              nods *p= cap;
                 cap= cap→leg;
                 delete p
                }
  else inf =0;
return *this;
}
```

Putem folosi fişierul antet **stivaI.h** pentru inserare de elemente în stivă, pentru eliminare de elemente din stivă, pentru listare, etc, ca în exemplul următor :

```
#include <iostream.h>
#define tipinfS float
#include "stivaI.h"
void main( )
{
   stiva S;
   itstiva it;
   float x;
   int opt;
   do {
                  cout <<endl;
                  cout << "1 : Inserare          elemente" <<endl;
                  cout << "2 : Eliminare elemente" <<endl;
                  cout << "3 : Listare stiva" <<endl;
                  cout << "4 : Stop" <<endl;
                  cout << "Tastati optiunea dorita: 1,2,3,4  ";
                  cin >> opt;
        switch (opt) {
        case 1 :  cout << "Tastati valoarea elementului : ";
                  cin >> x;
                  S << x;
```

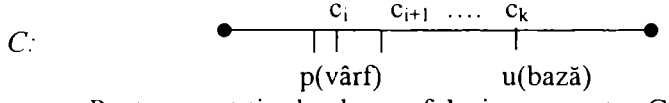
https://biblioteca-digitala.ro / https://unibuc.ro

```
            break;
    case 2 : if (!S) cout << "Stiva vida" << endl;
             else {
                    S >> x;
                    cout << "Am eliminat elemental : "<<x << endl;
                  }
             break;
    case 3 : cout << "Continutul stivei : " << endl;
             it.init(S);          ( obiectul it este iterator pentru stiva S)
             while ( it ) cout << it( ) << " " ;
              cout << endl;
              break;
              }
while(op!=4);
}
```

3.1.2 Cozi [47, 58]

Numim coadă o listă liniară în care inserările de elemente se fac la un capăt al listei numit **baza cozii** iar extragerile de elemente se fac la celălalt capăt al listei numit **vârful cozii.**

3.1.2.1 Alocarea secventiala a unei cozi



Pentru acest tip de alocare folosim un vector C cu n componente în care, dacă elementele cozii sunt memorate în complementele c_i ,c_{i+1} ,..., c_k , unde c_i este vârful cozii, adică primul său element iar c_k baza cozii adică ultimul său element, folosim variabilele p şi u cu valorile p=i-1 şi u=k ce indică indicele ultimului element extras respectiv al ultimului introdus în coadă. Dacă p=u coada este vidă. Prin scoaterea unui element din coadă incrementam p iar prin introducerea unui element în coadă incrementam u. Astfel coada migrează către dreapta şi poate ajunge în situaţia p=u=n când desi coada este vidă nu putem introduce nici un element în coadă. Eliminăm acest inconvenient dacă presupunem că după c_n urmează c_1 Cu aceasta presupunere putem avea p=u atât pentru coada vidă cât şi pentru coada plină (u înaintează până ce ajunge să fie p, care

https://biblioteca-digitala.ro / https://unibuc.ro

dacă nu înaintează coada este plină). Pentru a elimina această ambiguitate vom folosi efectiv numai n-1 componente ale vectorului C situație în care coada vidă se identifică prin p=u iar coada plină prin p=u+1 (mod n).

La inițializarea cozii punem p şi u pe valoarea n.

În cele ce urmează vom crea fişierul antet **coadaS.h** în care definim clasa coada ce ne va permite să efectuăm operații asupra unei structuri de date de tip coada.

```
#ifndef tipinfC
        #define tipinfC int
#endif
#define MAXI = 100
class coada
{ tipinfC *c;
  int nmax;
  int p,u;

public:
    coada( )
   {
     nmax= MAXI+1;
     c=new tipinfC[nmax-1];
     p=u=nmax;
    }
coada(int max);
{
  nmax= max+1;
   c=new tipinfC[nmax-1];
   p=u=nmax;
}
~ coada( )
{
c++;
delete c;
}
coada& operator <<(tipinfC inf);
coada& operator >>( tipinfC& inf);
friend tipinfC& varf(coada&);
tipinfC& varf( );
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
tipinfC operator [ ](int i)
{
return c[p+i >max ? p+i-nmax : p+i];
}
int getn( )
{ int l=u-p;
  return l<0 ? l+nmax : l;
}
friend int getn( coada& c1)
{ int l=c1.u-c1.p;
   return l<0 ? l+c1.nmax : l;
}
int getmax( )
{ return nmax-1;
}
friend int getmax( coada& c)
    { return c.nmax-1;
    }
operator void*( )
   {
if (getn( ) > 0) return this;
     else return 0;
     }
int operator ==( int n)
    {
      if (n==0) return p= =u;
      if ( n== 1) return p== ( u==nmax ? 1: u+1);
      return -1;
    }
 };
coada& coada :: operator <<(tipinfC inf)
{   int k=(u==nmax ? 1 : n+1);
            if (k==p)
                    { cout << " Coadă plină " << endl;
                      return *this;
                     }
c[ u=k ] = inf;
return *this;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
}
coada& coada :: operator >>(tipinfC& inf)
{
if (p!=u)
        p=( p==nmax ? 1 : p+1);
        inf =c[ p];
        return *this;
}
tipinfC& varf( coada& c1)
{
 if (c1.p != c1.u) return  c1.c[c1.p];
}
tipinfC& coada :: varf ( )
{
if (p!=u) return c[p];
}
```

Vom scrie acum un program care foloseşte fişierul antet **coadaS.h** pentru a realiza operaţiile de bază cu o astfel de structură de date, operaţii precum introducere, listare şi extragere, etc.

```
#include <iostream.h>
#define tipinfC float
#include "coadaS.h"
void main( )
{ coada C(5);
  float x;
  int opt;
do {
        cout<<endl;
        cout << "1 : Inserare        elemente" << endl;
        cout << "2 : Extragere elemente" << endl;
        cout << "3 : Listare coada" << endl;
        cout << "4: Stop " << endl;
        cout << " Tastaţi opţiunea: 1,2,3,4";
    cin >> opt;
    switch (opt)
        { case 1:  cout << " Tastati valoarea elementului:" ;
                   cin >>x;
                   C<<x;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                break;
        case 2 : if (!C) cout << "Coadă vidă " <<endl;
                 else { C >>x;
                        cout <<"Am extras elementul :"<<x<<endl;
                      }
                 break;
        case 3 : cout << "Conținutul cozii :"<<endl;
                  for (int i=1; i<= C.getn( ); i++) cout << c[i]<<" " ;
                 cout <<endl;
                 break;
      }
  }
while(op!=4);
}
```

3.1.2.2 Alocarea înlănțuită a unei cozi [47]

Acest tip de alocare necesită cunoaşterea adreselor vârfului cozii şi a bazei sale, adică a primului său element respectiv a ultimului său element. Aici legăturile între elemente merg în sensul de la vârf spre bază ca în figura alăturată:

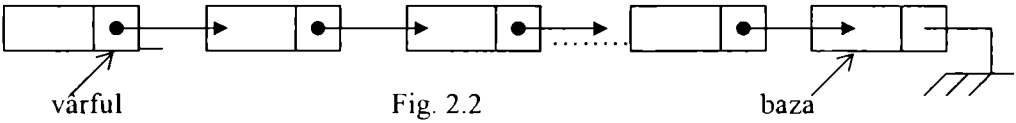


Fig. 2.2

Vom crea fişierul antet **coadaI.h** ce conține ca şi fişierul antet stival.h 3 clase cu semnificații analoage.

```
 #ifndef tipinfC
        #define tipinfC int
 #endif
class nodC                      (conține câmpurile unei celule (inf şi leg)
{                               precum şi operațiile ce se pot face cu
 tipinfC inf;                   obiecte de tip nodC ( 2 constructori şi o
 nodC *leg;                     funcție ce întoarce valoarea memorată
public:                                 în vârf)
     nodC( tipinfC infl; nodC *p=0)
       { inf =infl;
         leg =p;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        }
        tipinfC& get ( )
         { return inf;
        }
friend class coada;
friend class itcoada;
};
class coada              ( încapsulează două variabile de tip pointer către
{                         nodC  care dau adresa primului element al cozii şi
  nodC *prim;             ultimului său element, precum şi metode de
  nodC *ultim;            lucru cu aceste variabile)
 public :
      coada( )
        { prim=ultim=0;
        }
   coada& operator <<(tipinfC inf);
   coada& operator >>(tipinfC& inf);
   tipinfC& varf ( )
     { return prim →inf;
      }
tipinfC& baza( )
     { return ultim →inf;
      }

operator void*( )
       { return prim;
        }
friend class itcoada ;
};
class itcoada                (implementează un iterator pentru cozi)
{
  nodC *p;
public :
        itcoada ( )          ( constructor implicit care permite definirea
        { p=0;                 unor tablouri de iteratori)
          }
itcoada(coada& C)            ( iniţializează p cu adresa primului element al
    {                          cozii dată ca argument)
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
      p=C.prim;
     }
void init(coada& C)
    {
      p= C.prim;
     }
tipinfC operator(void)        ( supraîncarcă operatorul apel de funcţie;
  { nodC *q =p;                 Primul operand este obiect de tip itcoada iar
    p= p→ leg;                  al doilea este în lista de argumente, care aici
    return q→inf;               este vidă)
  }
operator void*( )
  { return p;               (supraîncarcă operatorul de conversie void* pentru
    }                             personalizarea numelui obiectului. Funcţia
                                  întoarce o valoare nenulă dacă în coadă mai
                                  sunt elemente neiterate )
};
coada& coada :: operator <<(tipinfC inf)
{
if (prim) ultim=ultim→leg=new nodC(inf,0);
else prim=ultim=new nodC(inf,0)
 return *this;
 }
coada& coada :: operator >>(tipinfC& inf)
{
  if (prim) {  inf= prim→inf;
               nodC *p= prim;
                  prim= prim→leg;
                  delete p
                 }
  else inf =0;
return *this;
}
```

Folosim acum fişierul antet **coadaI.h** pentru a realiza principalele operaţii ce se pot efectua asupra unei cozi aşa cum ar fi : inserare, ştergere, listare, etc.

```
#include <iostream.h>
#define tipinfC float
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
#include "coadaI.h"
void main( )
{
   coada C;
   itcoada it;                    (obiectul it este iterator pentru coada C)
   float x;
   int opt;
        do {
                cout <<endl;
                cout << "1 : Inserare          elemente" << endl;
                cout << "2 : Extragere elemente" << endl;
                cout << "3 : Listare coada" << endl;
                cout << "4: Stop " << endl;
                cout << "Tastati optiunea dorita : 1,2,3,4"
                cin >> opt;
        switch (opt) {
        case 1 :  cout << "Tastati valoarea elementului : ";
                   cin >> x;
                   C << x;
                        break;
                case 2 :  if (!C) cout << "Coada vida" << endl;
                           else {
                                   C >> x;
                                   cout << "Am extras elementul : "<<x <<
        endl;
                                 }
                     break;
                case 3 :  cout << "Continutul cozii : " << endl;
                           it.init(C);
                           while ( it ) cout << it( ) << " " ;
                            cout << endl;
                            break;
          }
        }
while(op!=4);
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

3.2 Structuri de date de tip arbore

Numim **arbore** o mulțime finită A de elemente numite noduri cu proprietățile:

a. Există un singur nod distinct r din A numit **rădăcină**.
b. Elementele din A-{r}, dacă aceasta este nevidă, se împart în două sau mai multe submulțimi disjuncte, fiecare formând câte un nou arbore, numiți subarbori ai lui r.

Dacă unui nod nu îi corespund alți arbori, el se numește **nod terminal** al arborelui de rădăcină r sau **nod frunză.**
Un nod rădăcină se mai numește **nod tată**, rădăcina unui subarbore al său se mai numește **nod fiu**, iar subarborii săi se numesc descendenți.
Se mai spune că rădăcina r a unui arbore este situată la nivelul 1, fiii săi la nivelul 2 s.a.m.d. Daca un nod este la nivelul n atunci fii sai (dacă există) sunt la nivelul n+1.
Dacă pentru fiecare nod într-un arbore, subarborii săi sunt într-o anumită ordine avem un **arbore ordonat.**

3.2.1 Arbori binari [47, 58, 80]

Spunem că un arbore este binar dacă rădăcina și toate nodurile neterminale au cel mult doi subarbori considerați într-o anumită ordine: subarbore stâng și subarbore drept.

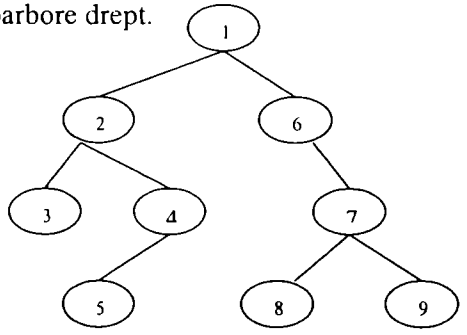


Fig 2.3

Un exemplu de arbore binar este cel din figura de mai sus.
Cu arborii binari se pot efectua următoarele operații:

- crearea unui arbore
- parcurgerea unui arbore
- cautarea unui nod într-un arbore
- inserarea unui nod

https://biblioteca-digitala.ro / https://unibuc.ro

- ştergerea unui nod
- ştergerea unui arbore

Nodurile unui arbore se pot parcurge în următoarele trei moduri: preordine, inordine şi postordine.

La parcurgerea in **preordine** se vizitează rădăcina şi apoi tot în preordine se vizitează nodurile subarborelui stâng, şi apoi cele ale subarborelui drept. Pentru exemplul de mai sus obţinem lista: 1, 2, 3, 4, 5, 6, 7, 8, 9.

La parcurgerea in **inordine** se vizitează în inordine nodurile subarborelui stâng, apoi se vizitează rădăcina şi în cele din urmă se vizitează în inordine nodurile subarborelui drept. Pe exemplul de mai sus obţinem lista: 3, 2, 5, 4, 1, 6, 8, 7, 9.

La parcurgerea in **postordine** se vizitează în postordine nodurile subarborelui stâng, apoi în postordine nodurile subarborelui drept, şi în final se vizitează rădăcina. Pentru exemplul de mai sus obţinem lista: 3, 5, 4, 2, 8, 9, 7, 6, 1.

Un arbore binar se implementează uşor prin pointeri. Fiecare nod conţine informaţia propriuzisă asociată nodului şi două câmpuri pentru legăturile dintre noduri, unul pentru adresa fiului stâng al nodului şi alta pentru adresa fiului drept al acestuia.

Arborele binar din figura anterioară se reprezintă sub forma:

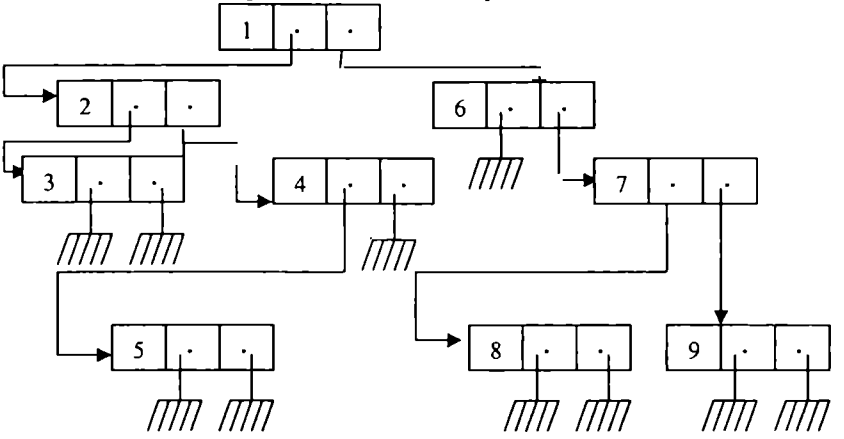


Fig. 2.4

In cele ce urmează vom construi un fişier antet **arboreB.h** prin care implementăm operaţiile de mai sus asupra arborilor binari. El conţine două clase nodA şi arbB care oferă suport de lucru cu arbori.

#include<iostream.h>

https://biblioteca-digitala.ro / https://unibuc.ro

```
class nodA
{public:
        int inf;
        nodA *ls,*ld;                   date publice
        nodA( )                         constructor implicit
        {ls=ld=0;
        inf=0; }
nodA(int infl,nodA *ls1=0,nodA *l1=0)       constructor
{inf=infl;
ls=ls1;
ld=ld1;}
 };
void creA(nodA *&p)     creează arborele furnizând nodurile în preordine
{ int x;                se va obține un arbore de sortare-căutare
cout<<"Informație nod=";
if(cin>>x){
        p=new nodA(x);
        cout<<"Nodul din stanga=";
        creA(p->ls);
        cout<<"Nodul din dreapta=";
        creA(p->ld); }
else {cin.clear( );
        cin.ignore(80,"\n");} }
void actA(nodA *&p, int x)              actualizare arbore de sortare
if (p)                                  prin inserarea lui x în arbore
        if (x<p->inf) actA(p- >ls, x);
        else actA(p->ld,x);
else p=new nodA(x); }
void copA(nodA *p,nodA *&pn)        copiază un arbore în altul (cel cu
                                    rădăcina la adresa p în cel cu
{if(p!=0) {                         rădăcina la adresa pn)
        pn=new nodA(p->inf);
        copA(p->ls,pn->ls);
        copA(p->ld,pn->ld); } }
void prelN(int x)                       prelucrarea unui nod (pot fi şi alte
{cout<<x<<" ";}                         tipuri de prelucrări)
void inord(nodA *rad)                   parcurgerea în inordine
{if (rad){
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        inord(rad->ls);
        prelN(rad->inf);
        inord(rad->ld);} }
void preord(nodA *rad)                  parcurgerea în preordine
{if(rad){
        prelN(rad->inf);
        preord(rad->ls);
        preord(rad->ld); } }

void postord(nodA *rad)                 parcurgerea în postordine
{if(rad){
        postord(rad->ls);
        postord(rad->ld);
        prelN(rad->inf); } }
void caut(nodA *rad,int x, nodA *&adr)  caută un element în arborele de
{if (adr= =0) {                         sortare-căutare
        if(rad->inf= =x) {
                adr=rad;
                return;}
        if (rad->ls) caut(rad->ls,x,adr);
        if(rad->ld) caut(rad->ld,x,adr); } }
void stergA(nodA *rad)                  şterge un arbore prin parcurgere în
{if(rad!=0){                            preordine
        stergA(rad->ls);
        stergA(rad->ld);
        delete rad;} }
class arbB
{nodA *rad;
public:
arbB( )
        {rad=0;}
~arbB( )
        {stergA(rad);}
nodA *getrad( )
        {return rad;}
void creA( )
        {::creA(rad);}          folosim :: deoarece sunt apelate functii
                                definite în afara clasei arbB
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
void inord( )
        {::inord(rad);}
void preord( )
        {::preord(rad);}
void postord( )
        {::postord(rad);}
void actAS(int x)                actualizare arbore de sortare
        {::actA(rad,x);}
arbB& operator =(arbB& A)
{::copA(A.rad,rad);
return*this;}
nodA *caut(int x)
{nodA *adr=0;
if(rad)::caut(rad,x,adr);
return adr;}
};
```

Clasa arbB oferă suport de lucru cu arbori.

Putem folosi fişierul antent **arboreB.h** pentru a efectua operaţii asupra arborilor binari după cum se vede în cele ce urmează:

```
#include<iostream.h>
#include"arboreB.h"
void main( )
{ arbB a;
int x;
cout<<"Tastati un sir de numere naturale incheind cu ."<<endl;
for( ; ; )
        if(cin>>x) a.actAS(x);
        else break;              creează un arbore de sortare
cin.clear( );
cin.ignore(80,'\n');
cout<<"Sirul sortat este:"<<endl;
a.inord( );
cout<<endl;
arbB b,c;
cout<<"Copiez arborele a in arborele b:"<<endl;
b=a;
cout<<"Arborele b in inordine este:"<<endl;
b.inord( );
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
cout<<endl;
cout<<"Tastati in preordine elementele unui arbore:"<<endl;
cout<<"Marcati cu . subarborele vid"<<endl;
c.creA();
cout<<"Arborele c in preordine:"<<endl;
c.preord();
cout<<"Tastati 10 numere naturale pe care sa le cautam in arbore:"<<endl;
for (int i=1;i<11;i++) {
        cin>>x;
        nodA *p=c.cauta(x);
        if(p) cout<<x<<"se afla in arbore"<<endl;
        else cout<<x<<"nu este in arbore"<<endl;} }
```

3.2.2 Arbori oarecare [12, 47]

Un exemplu de arbore oarecare este prezentat în figura de mai jos. Aici pot exista noduri cu 3 sau mai mulți fii.

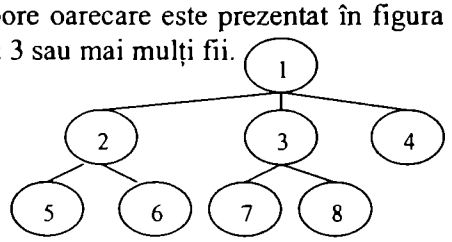


Fig. 2.5

Pentru reprezentarea unui astfel de arbore considerăm că fiecare nod conține informația asociată nodului, numărul de descendenți ai săi și adresa fiecărui descendent.

Pentru crearea arborelui vom parcurge nodurile sale în lățime astfel: pentru fiecare nod citim informația utilă și numărul descendenților. Păstrăm nodurile într-o stivă până ce apare nodul tată. În acest moment scoatem fiii din stivă, actualizăm referințele descendenților (de la tată la fii) și apoi punem tatăl în stivă. La sfârșit singurul nod din stivă va fi rădăcina arborelui, iar arborele va fi creat.

Pentru parcurgerea arborelui vom folosi o coadă în care punem nodurile ce urmează a fi prelucrate. Arborele se parcurge pe orizontală, nivel după nivel. Inițial punem în coada vidă rădăcina arborelui. Scoatem apoi din coadă rând pe rând câte un nod și introducem descendenții săi.
Vom construi un fișier antet **arboreO.h** în care implementăm principalele operații cu arbori oarecare: creare, prelucrare, ștergere, parcurgere și

https://biblioteca-digitala.ro / https://unibuc.ro

traversarea arborelui pe nivele.

```
#include<iostream.h>
#include<stdlib.h>
class nodA;
#define tipinfS nodA*
#define tipinfC nodA*
#include"\CPP\stivaI.h"
#include "\CPP\coadaI.h"
class nodA
{ public:
        int inf;
        int nfii;
        nodA **leg;
        nodA( );
        {inf=0;
        nfii=0;}
        nodA(int inf1, int nf=0)
        {inf=inf1;
        nfii=nf;
      if (nf>0) leg=new nodA*[nfii];}
      };
class arb
{nodA *rad;
public:
        arb( )
        {rad=0;}
        nodA *getrad( )
        {return rad;}
        int creA( );
        void preord( );
        void postord( );
        void trav( );
        ~arb( );
};
void prelN(int);
```

prototipul unei funcții ce va fi definită în fișierul unde va fi necesară

```
int arb::creA( )
{stiva S;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
int n;
int inf, nfii;
nodA *p;
cout<<Tastati numarul nodurilor arborelui:";
cin>>n;
cout<<"Tastati nodurile in postordine:"<<endl;
for (int i=1; i<n; i++) {
        cout<<"inf.nod="<<i<<":";
        cin>>inf;
        cout<<"numar fii";
        cin>>nfii;
        p=new nodA(inf, nfii);
        for (int k=nfii; k>=1; k- -)
            if(S) S>>p->leg[k-1]        extrage adresa elementului din
                                        varful stivei
            else {cout<<"Ati gresit numarul descendentilor!"<<endl;
                    return 0;}          eşec la creare
                S<<p;}                  introduce p în stiva
        S>>rad;                         extrage rădăcina din stivă
        return 1;}                      succes la creare
void arb::trav( )
{if rad = =0) return;
coada C;
C<<rad;                 introduce rădăcina în coadă
nodA *p;
while(C) {
        C>>p;           scoate din coadă rând pe rând câte un nod
        prelN(p->inf);
        for(int i=0; i<p->nfii; i++)
                C<<p->leg[i];} }
void preord(nodA *rad)
{
if(rad){
        prelN(rad→inf);
        for(int i=0; i<rad→nfii;i++)
                preord[rad→leg[i]);
                }
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
void arb::preord( )
{::preord(rad);}
void postord(nodA *rad)
        {if (rad) {
                for (int i=0; i<rad->nfii; i++)
                        postord(rad->leg[i]);
                prelN(rad->inf);} }
void arb::postord( )
{::postord(rad);}
void sterg(nodA *rad)
{if(rad) {
        for (int i=0; i<rad->nfii; i++)
                sterg(rad->leg[i]);
                delete(rad);
                rad=0;} }
arb::~arb( )
{::sterg(rad);}
```

Vom folosi acum fişierul antet **arboreO.h** de mai sus pentru crearea unui arbore oarecare, parcurgerea în preordine şi postordine şi pentru traversarea pe nivele a acestuia.

```
#include<iostream.h>
#include"arboreO.h"
void prelN(int x)
{cout<<x<<" ";}
void main( )
{arb A;
A.creA( );
cout<<"Parcurgerea in preordine da lista:"<<endl;
A.preord( );
cout<<endl;
cout<<"Parcurgerea in postordine da lista:"<<endl;
A.postord( );
cout<<endl<<"Traversarea arborelui pe nivele da lista:"<<endl;
A.trav( );
cout<<endl;
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

3.3 Fişiere în C ++ [47]

Pentru a efectua I/O cu fişiere, trebuie să includem în programe fişierul ***fstream.h*** care conţine clasele **ifstream, ofstream** şi **fstream.** Aceste clase sunt derivate din clasele **istream** şi **ostream** care la rândul lor sunt derivate din clasa **ios.** Ele sunt definite în fişierul ***iostream.h***. Astfel declaraţiile claselor **ios, istream** şi **ostream** rămân valabile şi pentru lucrul cu fişiere.

În C++ un fişier este deschis prin cuplarea lui la un **stream** care poate fi de intrare (I), de ieşire (O) şi de intrare/ieşire (I/O).

Pentru a crea un **stream de I** el trebuie declarat de tip **ifstream** iar pentru a crea un **stream de O** el trebuie declarat de tip **ofstream.** Un **stream de I/O** se crează prin declararea lui de tip **fstream.**

Constructorii impliciţi ai claselor ifstream, ofstream şi fstream iniţializează streamuri fără a deschide fişiere.

Funcţia **open**, membră a tuturor celor 3 clase de tip stream permite asocierea unui fişier la un stream, după crearea streamului. Ea are prototipul :

void open(char *nf, int md, int mp);

unde

- nf = numele fişierului;
- md= modul de deschidere cu valorile : ios ::în, ios ::out, ios ::ate, ios ::app, ios :: trunc ios:: nocreate, ios:: noreplace, ios :: binary
- mp= modul de protecţie: cu valorile 0,1,2,4 şi 8

Semnificaţiile acestor moduri sunt :

- ios :: in – deschidere pentru citire(fişierul trebuie să existe).Valoare implicită pentru ifstream este **in**
- ios :: out – deschidere pentru scriere(dacă fişierul nu există el se crează, iar dacă există conţinutul său se pierde)
- ios :: ate – deschidere şi poziţionare la sfârsit de fişier
- ios :: app – deschidere pentru scriere la sfârşit de fişier(fişierul trebuie să existe)
- ios :: trunc – deschidere şi trunchiere de fişier(la lungime 0), dacă există
- ios :: nocreate – fişierul trebuie să existe la deschidere, altfel eroare
- ios :: noreplace – fişierul trebuie să fie nou la deschidere, altfel deschiderea eşuează

https://biblioteca-digitala.ro / https://unibuc.ro

- ios :: binary – este opusul fişierului "text" care este şi valoare implicită.
- 0 – fişier normal fără restricţii de acces(valoare implicită)
- 1 – fişier de tip "read-only"
- 2 – fişier ascuns
- 4 – fişier de tip sistem
- 8 – bit de arhivare activ

Putem folosi mai multe atribute folosind operatorul **sau** binar "|".

Exemplu.

```
ifstream f;
f.open("f1.dat");
ofstream g;
g.open("f2.dat");
fstream h;
h.open("f3.dat", ios ::în|ios ::out);
```

Clasele ifstream, ofstream şi fstream conţin constructori care deschid în mod automat fişierele. Ei au aceiaşi parametrii şi valori implicite ca şi funcţia open. Astfel putem folosi direct instrucţiuni de tipul:

```
ifstream f.open("f1.dat");
ofstream g.open("f2.dat");
fstream h.open("f3.dat", ios :: în|ios :: out);
```

Dacă deschiderea fişierului eşuează streamul conţine valoarea zero şi deci putem testa dacă operaţia de deschidere a reuşit.

Destructorii claselor ifstream, ofstream şi fstream golesc tamponul fişierului şi închid fişierul, dacă acesta nu este deja închis. Inchiderea unui fişier se face cu funcţia membră : **void close(void);**

Funcţia membră **int eof( )** întoarce o valoare diferită de zero dacă s-a atins sfârşitul fişierului şi zero în caz contrar.

Scrierea şi citirea într-un/dintr-un fişier text deschis se face prin operatorii << respectiv >> ca şi pentru operaţiile de I/O de la consola dar în loc să folosim dispozitivele standard cout şi cin folosim streamul cuplat la fişier.

În operaţii de I/O cu fişiere text, caracterele **newline** se transformă în combinaţia de caractere cr/lf. În fişierele de tip binary aceste caractere nu se traduc.

Pentru fişierele binare funcţiile de I/O sunt :

https://biblioteca-digitala.ro / https://unibuc.ro

1. **int get( );** extrage din streamul asociat un caracter şi intoarce codul caracterului extras.
2. **istream get(char &c);** citeşte un caracter din streamul asociat şi îl memorează în variabila de ieşire c. Ea intoarce o referinţă la streamul asociat care va fi zero când s-a atins sfârşitul fişierului.
3. **istream get(char *p, int n, char delimitator='\n');** citeşte n caractere din streamul asociat sau până la întâlnirea delimitatorului dat ca argument şi le depune în zona indicată de p. Delimitatorul nu se extrage din şir şi implicit este \n.
4. **istream& getline(char *p, int n, delimitator='\n');** ca la 3 dar extrage eventual şi delimitatorul.
5. **ostream& put(char c);** scrie în streamul asociat caracterul dat ca argment
6. **int peek( ) ;** furnizează următorul caracter din streamul de intrare fără să îl extragă din stream.
7. **istream& putback(char c);** reinsereză un caracter în şir.
8. **istream& read(unsigned char *p, int n);**
   ostream& write(const unsigned char *p, int n); scriu respectiv citesc blocuri de date binare.

Funcţia read citeşte n octeţi din streamul asociat şi îi depune în zona indicată de p. Dacă se detectează sfârşitul fişierului înainte de citirea tuturor octeţilor , read opreşte citirea. Numărul caracterelor citite este furnizat de funcţia următoare:

9. **int gcount( );** întoarce numărul caracterelor citite la ultima operaţie de intrare.

În operaţiile de I/O la fişiere binare se folosesc 2 pointeri pentru acces la fişiere şi anume : **get pointer** şi **put pointer**. Primul indică poziţia din fişier de unde se va face citirea iar al doilea dă poziţia la care se va face următoarea scriere. La fiecare operaţie de I/O pointerul corespunzător avansează secvenţial în mod automat. Un acces aleator la astfel de fişiere (binare) se va face prin funcţiile :

10. **ostream& seekg(long n);** poziţionează **get pointer** pe octetul n în fişier (numerotarea începe cu 0)
11. **ostream& seekg(long n, dir origine);** deplasează **get pointer** asociat fişierului cu n octeţi faţă de origine.

dir este un tip enumerare cu valorile : ios :: begin=0, ios :: current=1, ios :: end=2, semnificând începutul, poziţia curentă, respectiv sfârşitul streamului.

https://biblioteca-digitala.ro / https://unibuc.ro

12. **ostream& seekp(long n);**
ostream& seekp (long n, dir origine); sunt ca şi cele de la 10 şi 11 dar se aplică pointerului put pointer.

Aplicaţii [47]

1. Program pentru listarea unui fişier text la imprimantă

```
#include < istream.h>
#include < iomanip.h>
#include < math.h>
#include < string.h>
#include < conio.h>
void listf(char nf[25])                    (listare la imprimantă)
{
   ifstream finput(nf);
   if (!finput) {
                  cout << "fişier inexistent"<<endl;
                  return;
                 }
   ofstream fout("lpt1");                  (asignăm imprimanta la streamul
                                                          fout)
   char sir[81];
   int i=0;
  while (!finput.eof( )) {
                      i++;
                      if (i>64) {
                            cout << "schimb pagina"<<endl;
                            getch( );
                            i=0;
                           }
                      finput.getline(sir, 80,'\n');
                      fout.write( sir, sizeof(sir));
                      fout <<end;
                      }
       finput.close( );
       fout.close( );
}
void main( )
{
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
  char nf[25];
  cout <<"Tastati numele fişierului de listat la imprimanta :";
cin>>nf;
listf(nf);
}
```

2. Program pentru crearea, actualizarea şi listarea unui fişier binar[47]

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
ofstream list("lpt1");
class articol
{ protected:
        int doc,dep;
        char den[20];
        float pret,cant;
 public:
        friend int operator >>(istream&,articol&);
        friend ostream& operator <<(ostream&,articol&);
        friend ifstream& operator >>(ifstream,articol&);
        friend ofstream& operator <<(ofstream&,articol&);
        friend void situatie(char*,ostream&);
        operator unsigned chr*()
        {
        return(unsigned chr*) this;
        }
};
int operator >>(istream& s,articol& art)
{
 static r=0;
 r++;
 cout<<"articolul"<<r<<endl;
 cout<<"document : ";
 s>>art.doc;
 if (s==0) {
        r--;
        s.clear();
        s.ignore(80,'\n');
         return 0;}
 cout<<"depozit : ";
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
 s>>art.dep;
 cout<<"denumire: ";
 s>>art.den;
 cout<<"cantitate : ";
 s>>art.cant;
 cout<<"pret : ";
 s>>art.pret;
 if (s==0)  {
          r--;
          s.clear();
         s.ignore(80,'\n');
         return 0;}
  return 1;
}
ostream& operator <<(ostream& s,articol& art)
{
 s<<setprecision(2);
 s<<setw(2)<<art.doc<<" "<<setw(2)<<art.dep<<"
"<<setw(20)<<art.den<<setw(10) <<art.cant<<setw(10)<<art.pret;
 return s;
}
ifstream& operator >>(ifstream& s,articol& art)
{
 s>>art.doc;
 s>>art.dep;
 s>>art.den;
 s>>art.cant;
 s>>art.pret;
 return s;
}
istream& read(istream& s,articol& art)
{
s.read((unsigned char*) art,sizeof(art));
 return s;
}
ostream& write(ostream& s,articol& art)
{
s.write((unsigned char*) art,sizeof(art));
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
 return s;
}
void actualizare(chr* denFis)
{
ofstream f(denFis, ios::app::binary);
 articol art;
 while (cin>>art) write(f,art);
 f.close();
}
void creare(chr* denFis)
{
 ofstream f(denFis, ios::out::trunc::binary);
 f.close();
 actualizare(denFis);
}
void modificare(char* denFis)
{
 fstream f(denFis,ios::in::out::binary);
 articol art;
 f.seekp(0,ios::end);                    // sfarsitul streamului
 long nr=f.tellp()/sizeof(art);          // nr. de articole
 cout<<"nr. de articole eronate : ";
 long n;
 if ((cin>>n) && n>0 && n<=nr);
   {
    f.seekg((n-1)* sizeof(art));
    read(f,art);
    cout<<art<<endl;
    cin>>art;
    f.seekp((n-1)* sizeof(art));
     write(f,art);
   }
  f.close();
}
void afisare(char* denFis,ostream& ot)
{
  ifstream f(denFis,ios::in::binary);
  static articol art;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
  while (1)
       {read(f,art);
         if (f.eof()) break;
         ot<<art<<endl;
       }
 f.close();
}
void situatie(chr *denFis, ostream& ot)
{
  ifstream  f(denFis,ios::in::binary);
   articol art;
   float v=0, tvd=0, tvg=0;
   read(f,art);
   int d=art.doc;
   ot<<"SITUATIA APROVIZIONARII"<<endl;
   ot<<setprecision(2)<<setiosflags(ios::fixed);
   while(f.eof()) {
       if (d!=art.doc) {
                        ot<<setw(2)<<d<<setw(50)<<tvd<<char(4)<<endl;
                          tvg+=tvd;
                           d=art.doc;
                        }
       v=art.cant* art.pret;
       ot<<art<<setw(10)<<v<<endl;
        tvd+=v; read(f,art);
             }
tvg+=tvd;
ot<<setw(2)<<d<<set(50)<<tvd<<char(4)<<endl;
ot<<setw(50)<<tvg<<"*"<<endl;
f.close();}
void main()
{
 int op;
 char denf[30];
  do {
     cout<<endl;
     cout<<"1- creare"<<endl;
     cout<<"2- actualizare"<<endl;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
    cout<<"3- modificare"<<endl;
    cout<<"4- listare la consola"<<endl;
    cout<<"5- listare la imprimanta"<<endl;
    cout<<"6- listare in fisier text"<<endl;
    cout<<"7- editarea situatiei la consola"<<endl;
    cout<<"8- editarea situatiei la imprimanta"<<endl;
    cout<<"9- editarea situatiei in fisier text"<<endl;
    cout<<"10- stop"<<endl;
    cout<<" Tasteaza optiunea: 1,2,3,...,10";
    cin>>op;
    switch(op) {
          case 1: creare("fis.dat");break;
          case 2: actualizare("fis.dat");break;
          case 3: modificare("fis.dat");break;
          case 4: clrscr();
                  listare("fis.dat",cant);
                  getch();break;
          case 5: listare("fis.dat",list);break;
          case 6: cout<<"denumire fisier: ";
                  cin>>denf;
                  ostream g(denf,ios::trunc);
                  listare("fis.dat,g);
                  g.close();break;
          case 7: clrscr();
                  situatie("fis.dat",cant);
                  getch(); break;
          case 8: situatie("fis.dat",list);break;
          case 9: cout<<"Denumire fisier: ";
                  cin>>denf;
                  ofstream h(denf,ios::trunc);
                  listare("fis.dat",h);
                  h.close();break;
                }
    }
while (op!=10);
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL IV Elemente de grafică pe calculator

4.1 Generalităţi [6, 46, 68]

Dinamica fără egal a graficii pe calculator este strâns legată de facilităţile grafice ale calculatoarelor personale, de apariţia şi dezvoltarea unui soft specializat.

Pentru înţelegerea modului de realizare a funcţiilor grafice şi pentru a putea crea soft în acest domeniu este necesar un studiu atent al fundamentelor graficii computerizate cum ar fi: *sisteme grafice, funcţii, algoritmi, metode şi modele, instrumente şi caracteristicile lor, etc.*

Prin ***sistem grafic*** înţelegem un ansamblu de resurse fizice şi logice specializat în introducerea, stocarea, extragerea şi prelucrarea informaţiei astfel ca informaţia vizualizată să fie de natură grafică.

Sistemul grafic interactiv este acel sistem grafic care permite interacţiunea cu utilizatorul prin dispozitive grafice precum: *mouse, joystick, creion optic*, etc.

Sistemele grafice se împart în 3 clase:

1. **sisteme de sinteză** a imaginilor care generează şi afişează o imagine grafică prin algoritmi de generare a primitivelor grafice precum: *puncte, segmente, linii frânte, conice, curbe oarecare şi suprafeţe.*
2. **sisteme de prelucrare şi analiză a imaginilor** care au la intrare o reprezentare numerică a imaginii reale pe care o supun unor transformări specifice pentru a îmbunătăţi calitatea imaginii şi a evedenţia anumite aspecte ale ei. La ieşire se obţine o imagine constituită din *puncte sau din vectori pentru dispozitive vectoriale.*
3. **sisteme de prelucrare a datelor volumetrice** care redau structura internă a unui obiect şi realizează secţiuni în obiecte pornind de la elemente de volum (voxeli) organizate în grile tridimensionale. Datele volumetrice se obţin prin *eşantionare cu tomograf, ultrasunete, rezonanţă magnetică*, etc sau printr-un *model matematic.*

Sistemele grafice se bazează pe unităţi centrale cu microprocesoare rapide sau pe sisteme multiprocesor. **Procesorul grafic** execută algoritmi

https://biblioteca-digitala.ro / https://unibuc.ro

specifici prelucrărilor grafice: *generare de primitive grafice, decupare, eliminare de suprafete ascunse, tratarea iluminării, etc.*

Memoria sitemelor grafice este compusă din câteva categorii de module funcționale:

- ***memoria internă*** a sistemului de calcul
- ***memoria specială*** a procesorului grafic
- ***memoria locală*** specifică echipamentelor de vizualizare grafică
- ***memoria video, etc.***

Imaginea generată de un sistem grafic este caracterizată de noțiunile urmatoare:

- *dimensiunea unui punct* (diametrul unui punct generat de echipamentul de iesire)
- *adresabilitatea* (numărul punctelor pe care le poate crea echipamentul pe unitatea de lungime); se măsoară în pct/inch.
- *rezoluția* (spațierea minimă între şiruri de linii negre alternate cu linii albe, percepute de om)
- *aliasing* (asperități sau rupturi ale liniilor sau curbelor trasate prin puncte)
- *antialiasing* (o metodă de netezire pentru înlăturarea efectului aliasing)

4.2 Fişierul <graphics.h> [47, 68]

Pentru a realiza grafică în C++ şi text pe ecran trebuie inclus în program fişierul antet **graphics.h** care conține o multime de funcții grupate pe urmatoarele categorii :

- Funcții pentru inițializare modului de lucru grafic
- Funcții relative la fereastra de afişare
- Funcții relative la cursorul grafic
- Funcții relative la puncte şi segmente de dreaptă
- Funcții relative la culori
- Funcții relative la grosime, stil şi mod de desenare a liniilor
- Funcții relative la cercuri, arce, curbe
- Funcții relative la haşuri
- Funcții pentru salvări şi restaurări de imagini
- Funcții orientate spre texte

https://biblioteca-digitala.ro / https://unibuc.ro

- Funcţii relative la pagini grafice
- Funcţii pentru gestionarea culorilor
  etc.

4.2.1 Iniţializarea modului de lucru grafic folosind funcţiile :

1. **void detectgraph(int *gd, int *gm);**
   - Funcţia detectează adaptorul grafic şi întoarce driverul şi un mod grafic suportate de acest adaptor în *gd respectiv *gm.
   - Pentru *gd putem folosi constantele: DETECT care cere autodetectare, CGA=1, MCGA=2, EGA=3, EGA64=4, EGAMONO=5, IBM8514=6, HERCMONO=7 , ATT400=8, VGA=9, PC3270=10, CURRENT _DRIVER =-1
   - Pentru *gm putem folosi constantele :

CGAC0=0	MCGAC0=0	EGALO=0	ATT400C0=0	VGALO=0
CGAC1=1	MCGAC1=1	EGAHI=1	ATT400C1=1	VGAMED=1
CGAC2=2	MCGAC2=2	EGA64LO=0	ATT400C2=2	VGA HI=2
CGAC3=3	MCGAC3=3	EGA64HI=1	ATT400C3=3	PC3270HI=0
CGAHI=4	MCGAMED=4	EGAMONOHI=0	ATT400MED=4	IBM8514LO=0
	MCGAHI =5	HERCMONOHI=0	ATT400HI=5	IBM8514HI=1

Aici primele caractere din constante sunt pentru driverul folosit iar C0, C1, C2, C3, HI, MED se referă la paletele de culori folosite.

2. **void initgraph(int *gd, int *gm, char *cale);**
   - Funcţia iniţializează modul de lucru grafic
   - *gd precizează driverul grafic iar *gm - modul grafic .
   - **cale** precizează directorul care conţine fişierul .bgi curent.
3. **void closegraph( );** - închide modul de lucru grafic

4.2.2 Pagini grafice

Zona de memorie video necesară memorării unui ecran se numeşte **pagină grafică.**
Un adaptor poate avea mai multe pagini grafice, dar la fiecare moment o singură pagină grafică se afişeaza pe ecran, pagina numită **pagină vizibilă.**

https://biblioteca-digitala.ro / https://unibuc.ro

Pagina din memoria video în şi din care programatorul scrie sau citeşte la un moment dat se numeşte **pagina activă.** Implicit pagina vizibilă coincide cu cea activă.
Mai multe pagini grafice se folosesc pentru animaţie. Ele sunt numerotate începând cu 0.

Exemple.

- adaptorul VGA (=9) în modul grafic VGALO (=0) dispune de 4 pagini grafice a 640 x 200 pixeli.
- Adaptorul VGA (=9) în modul grafic VGAMED (=1) dispune de două pagini grafice a 640 x 350 pixeli.
- Adaptorul VGA (=9) în modul grafic VGAHI (=2) dispune de o pagină grafică a 640 x 480 pixeli.

Activarea şi vizualizarea unei pagini se face prin funcţiile :
4. void setactivepage(int pg); - activează pagina grafică pg
5. void setvisualpage(int pg); - vizualizează pagina grafică pg.

4.2.3 Gestiunea culorilor

Mulţimea culorilor ce pot fi afişate simultan pe ecran se numeşte **paletă.** Culorile sunt codificate cu numere întregi consecutive începand cu 0, pentru negru.
Exemplu.
Având un adaptor EGA putem folosi 64 de culori cu coduri în intervalul 0,1,...,63. Nu se pot afişa simultan decat cel mult 16 culori pe ecran.

Culorile unei palete pot fi modificate de programator prin funcţiile standard următoare :

6. void setpalette(int vechi, int nou); unde:

- vechi – reprezintă culoarea care se modifică
- nou – reprezintă culoarea nouă.

Exemplu.
setpalette(DARKGRAY, 45); modifică culoarea de cod 8 cu culoarea de cod 45.

Culorile au următoarele denumiri simbolice:

- pentru paleta implicită :

https://biblioteca-digitala.ro / https://unibuc.ro

  - BLACK=0, BLUE =1, GREEN=2, CYAN=3, RED=4, MAGENTA=5, BROWN=6, LIGHTGRAY=7, DARKGRAY=8, LIGHTBLUE=9, LIGHTGREEN=10, LIGHTCYAN=11, LIGHTRED=12, LIGHTMAGENTA=13, YELLOW=14, WHITE=15.
- pentru CGA avem:
  - paleta C0 :
    - CGA_LIGHTGREEN=1,
    - CGA_LIGHTRED=2,
    - CGA_YELLOW=3.
  - paleta C1:
    - CGA_ LIGHTCYAN=1,
    - CGA_LIGHTMAGENTA=2,
    - CGA_ WHITE=3.
  - paleta C2 :
    - CGA_GREEN=1,
    - CGA_ RED=2,
    - CGA_BROWN=3.
  - paleta C3 :
    - CGA_CYAN=1,
    - CGA_ MAGENTA=2,
    - CGA_LIGHTGRAY=3.
- pentru EGA avem :
  - EGA_BLACK=0, EGA_BLUE =1, EGA_GREEN=2, EGA_CYAN=3, EGA_RED=4, EGA_MAGENTA=5, EGA_BROWN=6, EGA_LIGHTGRAY=7, EGA_DARKGRAY=56, EGA_LIGHTBLUE=57, EGA_LIGHTGREEN=58, EGA_LIGHTCYAN=59, EGA_LIGHTRED=60, EGA_LIGHTMAGENTA=61, EGA_YELLOW=62, EGA_WHITE=63.

4.2.4 Funcţii pentru ferestre

7. void setviewport(int x1, int y1, int x2, int y2, int clip);
Funcţia defineşte o fereastră pe ecranul grafic de colţ stanga-sus (x1,y1) şi colţ dreapta-jos (x2,y2).

https://biblioteca-digitala.ro / https://unibuc.ro

Dacă clip=0 imaginea din afara ferestrei poate fi alterată de desenele ce nu încap în fereastra curenta, iar dacă clip≠0, imaginea din afara ferestrei nu va fi alterată.
8. void clearviewport( ); şterge fereastra grafică curentă folosind drept culoare de ştergere, culoarea de fond.
9. void cleardevice( ); şterge ecranul grafic folosind culoarea de fond ca şi culoare de ştergere.

4.2.5 Relativ la cursorul grafic avem funcţiile:

10. int getx( );
11. int gety( ); funcţii ce dau poziţia curentă a cursorului grafic, poziţie relativă la fereastra activă.
12. void moveto(int x, int y); mută cursorul grafic în punctul (x,y) (coordonatele sunt relative la fereastra activă)
13. void moverel(int dx, int dy); mută cursorul grafic din poziţia curentă (x,y) în poziţia (x+dx, y+dy)
14. int getmaxx( );
15. int getmaxy( ); funcţii ce dau numărul maxim de coloane respectiv numărul maxim de linii folosite de driverul curent.

4.2.6 Relativ la culori avem funcţiile

16. void putpixel(int x, int y, int cul); desenează şi colorează punctul de coordonate (x,y), relativ la fereastra activă, iar culoarea este dată de parametrul **cul**.
17. void setcolor(int cul); fixează culoarea pentru desenare prin valoarea parametrului **cul** care este un număr între 0÷15
18. void setbkcolor(int cul); fixează culoarea de fond prin valoarea parametrului **cul** care este un număr intre 0÷15
19. int getcolor( );
20. int getbkcolor( ); funcţii ce dau indexul culorii curente de desenare şi respectiv indexul culorii curente de fond
21. int getmaxcolor( ); returnează indexul maxim pentru culoare.
Exemplu:
pentru adaptorul VGA, getmaxcolor ( ); returnează 15 indicând că putem folosi culori de la 0 la 15.

https://biblioteca-digitala.ro / https://unibuc.ro

22. unsigned getpixel(int x, int y); returnează culoarea pixelului din punctul de coordonate (x,y)

4.2.7 Relativ la puncte şi segmente de dreaptă avem funcţiile

23. void line(int x1, int y1, int x2, int y2); desenează un segment de capete (x1,y1) şi (x2,y2). Poziţia cursorului grafic nu se modifica.
Culoarea segmentului este cea de desenare dată de setcolor, iar tipul şi grosimea segmentului sunt date de funcţia **setlinestyle** ce se va prezenta mai jos.
Punctele (x1,y1) şi (x2,y2) sunt relative la fereastra activă.
24. void linerel(int dx, int dy); desenează un segment între poziţia curentă a cursorului (x,y) şi punctul (x+dx, y+dy). Cursorul grafic se va poziţiona în punctul(x+dx, y+dy).
25. void lineto(int x, int y); desenează un segment între poziţia curentă a cursorului şi punctul (x,y), iar cursorul se duce în (x,y).
26. void rectangle(int x1, int y1, int x2, int y2); desenează un dreptunghi de colţ stânga-sus (x1,y1) şi dreapta-jos (x2,y2).
27.void drawpoly(int np, int coord[ ]); desenează un poligon folosind coordonatele date în vectorul coord [ ];

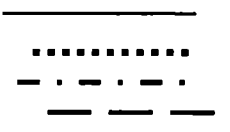
Exemplu:

```
1) int triunghi[ ] ={100,100,200,0,300,100,100,100};
   drawpoly (4, triunghi);
2) int P [ ] ={0,0,50,0,75,50,100,150,200,250};
   int np= sizeof (P)/ (2*sizeof (int));
   drawpoly (np,P);
```

4.2.8 Relativ la grosime, stil şi mod de desenare a liniilor avem funcţiile

28. void setlinestyle(int tip, unsigned model, int gr);
Funcţia setează tipul şi grosimea liniei prin constantele predefinite:

- SOLID_LINE =0 linie plină
- DOTTED_LINE =1 linie punctată
- CENTER_LINE=2 linie mixtă
- DASHED_LINE=3 linie întreruptă



https://biblioteca-digitala.ro / https://unibuc.ro

- USERBIT_LINE =4 un tip dat în biţi de către utilizator pentru tip şi prin constantele predefinite:
  - NORM_WIDTH=1 grosime normalî de 1 pixel
  - TICK_WIDTH =3 linie groasî

  pentru grosime (gr).

Parametrul **model** este folosit doar pentru tip=4, când descrie modelul liniei printr-o valoare întreagă pe 16 biţi, altfel este ignorat. Aici fiecare bit 1 înseamnă pixel în culoarea de desenare şi fiecare bit 0 înseamnă pixel în culoarea fondului.

Exemplu: setlinestyle(4, 0x4e3f, 1);

baza 16

indică linie cu modelul 0100111000111111

29. setwritemod(int mod); defineşte modul de desenare a liniilor prin parametrul **mod** care poate fi una din constantele :

COPY_PUT=0 (linie peste ecran) şi XOR_PUT=1 (XOR între punctele liniei şi punctele de pe ecran)

30. void getaspectratio(int *xasp, int *yasp); întoarce în *xasp şi *yasp mărimile din care se formează raportul dimensional al unui pixel.

Observaţie. Funcţiile din această categorie sunt valabile doar pentru funcţiile line, linerel, lineto, rectangle şi drawpoly.

4.2.9 Relativ la cercuri, arce şi curbe avem funcţiile

31. void circle(int x, int y, int r); desenează cercul C ((x,y), r) .

32. void arc(int x, int y, int ui, int uf, int r); desenează un arc din cercul C((x,y), r) cuprins între unghiurile de început **ui** şi de sfârşit **uf**.

33. void pieslice(int x, int y, int ui, int uf, int r);desenează şi haşurează un sector de cerc din cercul C ((x,y), r) cuprins între unghiurile ui şi uf.

34. void elipse(int x, int y, int ui, int uf, int rx, int ry); desenează un arc de elipsa din E((x,y), rx, ry) cuprins între unghiurile de început ui şi sfârşit uf.

35. void fillelipse(int x, int y, int rx, int ry); desenează şi haşurează interiorul elipsei E((x,y), rx, ry).

36. void sector(int x, int y, int ui, int uf, int rx, int ry); desenează şi haşurează un sector de elipsa din E((x,y), rx, ry) cuprins între unghiurile ui şi uf.

https://biblioteca-digitala.ro / https://unibuc.ro

4.2.10 Relativ la haşuri avem funcţiile :

37. setfillstyle(int mod, int cul); fixează modul de haşurare prin **mod** şi culoarea haşurii prin parametrul **cul**.
Pentru mod putem folosi constantele:

- EMPTY_FILL=0	umple zona în culoarea fondului
- SOLID_FILL=1	nuanţă uniform temperată
- LINE_FILL=2	linii orizontale
- LTSLASH_FILL=3	linii subţiri de forma ///
- SLASH_FILL=4	linii groase de forma ///
- BKSLASH_FILL=5	linii groase de forma \\\
- LTBKSLASH_FILL=6	linii subţiri de forma \\\
- HATCH_FILL=7	model în pătrăţele
- XHATCH_FILL=8	model în pătrăţele oblice
- INTERLEAVE_FILL=9	punctat des
- WIDE_DOT_FILL=10	punctat rar
- CLOSE_DOT_FILL=11	punctat intermediar
- USER_FILL=12	model definit de utilizator prin funcţia setfillpattern

38. void setfillpattern(char model[8], int cul); defineste un model de haşurare stabilit de utilizator prin parametrul model şi culoarea cul.
Tabloul model[8] descrie codificat haşura prin 1-bit în culoarea de desenare şi 0- bit în culoarea fondului.

Exemplu:
Haşura 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
se realizează astfel :

```
char model[8] ={0xAA,0x55,0xAA,0x55,0xAA,0x55, 0xAA,0x55};
setfillpattern(model, getcolor ( ));
```

https://biblioteca-digitala.ro / https://unibuc.ro

4.2.11 Privitor la salvări şi restaurări de imagini avem funcţiile

39. void getimage(int x1,int y1, int x2, int y2, void *p);

- salvează într-o zonă tampon dreptunghiul de pe ecran dat de colţul stânga-sus (x1,y1) şi dreapta-jos (x2,y2)
- p este pointer care dă adresa de memorie unde se salvează zona grafică.

40. void putimage(int x, int y, void *p, int mod);

- restaurează zona salvată cu getimage într-o zonă de pe ecran. Punctul (x,y) reprezintă colţul stanga-sus al zonei de pe ecran unde se restaurează imaginea.
- p este pointer ce dă adresa unde se află imaginea salvată cu getimage, ce trebuie restaurată.
- Parametrul mod indică modul de restaurare, adică stabileşte operaţia ce se face între pixelii de pe ecran şi cei care se restaurează.

Acest parametru poate lua valoriile:

- COPY_PUT=0 inlocuire
- XOR_PUT=1 XOR
- OR_PUT=2 OR
- AND_PUT=3 AND
- NOT_PUT=4 NOT

4.2.12 Privitor la funcţiile orientate spre texte găsim în fişierul antet funcţiile

41. void outtext(char *text); afişează şirul de caractere indicat de text începând cu poziţia curentă a cursorului grafic

42. void outtextxy(int x, int y, char *text); ca şi mai sus dar începand de la poziţia grafică de coordonate (x,y)

43. void settextstyle(int tipC, int Dir, int DimC); stabileşte tipul, direcţia şi dimensiunea caracterelor.

Pentru **tipC** putem folosi constantele:

- DEFAULT_FONT=0 (caractere în mod text),
  TRIPLEX_FONT=1, SMALL_FONT=2,
  SANS_SERIF_FONT=3, GOTHIC_FONT=4,
  SCRIPT_FONT=5, SIMPLEX_FONT=6,
  TRIPLEX_SCR_FONT=7, COMPLEX_FONT=8,
  EUROPEAN_FONT=9, BOLD_FONT=10.

https://biblioteca-digitala.ro / https://unibuc.ro

Pentru **Dir** folosim constantele :

- HORIZ_DIR=0 (de la stanga la dreapta), VERT_DIR=1 (de jos în sus).

Pentru **DimC** avem valorile:

- 1 (caracterele au dimensiunea implicitî 8x8 pixeli),
- 0 (dimensiunea caracterelor este dată de funcţia **returncharsize**)
- n=2,3,...caz în care dimensiunea implicită este mărită de n ori.

44. void returncharsize(int mx, int dx, int my, int dy); stabileşte lăţimea şi înălţimea caracterelor. Lăţimea implicită este înmulţită cu mx/dx, iar înălţimea implicită este înmulţită cu my/dy.

45. void settextjustify(int oriz, int vert); stabileşte alinierea, orizontală prin oriz şi vertivală prin vert, a textului.

Parametrul **oriz** poate avea valorile:

- LEFT_TEXT=0 aliniere stânga
- CENTER_TEXT=1 centrat orizontal
- RIGHT_TEXT=2 aliniat la dreapta

Parametrul **vert** poate fi :

- BOTTOM_TEXT=0 aliniat jos
- CENTER_TEXT=1 centrat vertical
- TOP_TEXT=2 aliniat sus

Exemplu [47].

Reprezentarea unor curbe precum: cicloida, cardioida, etc.

```
#include < conio.h>
#include < graphics.h>
#include < math.h>
#include < iostream.h>
struct coor { double x,y;
              }
typedef coor(*fun)(double,float);
const double pi=3.14159265358979;
int x0,y0;
void initgr( )
{ int gd,gm;
  detectgraph(&gd,&gm);
  initgraph(&gd,&gm, "c:\\CPP\\bgi");
void gr(fun f, double ls, double ld, float a, int x0,int y0)
{ float pas=pi/180;          ( argumentul f de tip fun este funcţia pe
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                                    care vrem să o reprezentăm)
  for (float t=ls;t<ld; t+=pas) {
                                        coor P= f (t,a);
                                        int x=x0 + P.x;
                                        int y=y0 – P.y;
                                        putpixel( x,y,getcolor( ));
                                    }
}
coor cicloida(double t, float a)    (curba descrisă de un punct situat pe un
                                     cerc care se rostogoleşte de-a lungul
                                     unei drepte)
{ coor P;
P.x=a*(t-sin(t));
P.y=a*(1-cos(t));
return P;
}
coor desf(double t, float a)                (desfăşurata cercului)
{ coor P;
P.x=a*( cos(t) + t* sin(t));
P.y=a*( sin(t) – t*cos(t));
return P;
}
coor cardioida(double t, float a)    ( curba descrisă de un punct situat pe
                                       un cerc ce se rostogoleşte pe un alt
                                       cerc de aceeaşi rază)
{ coor P;
  P.x=a*(2*cos(t) – cos(2*t));
  P.y=a*(2*sin(t) – sin(2*t));
  return P;
}
coor roza(double t, float a)
{coor P;
 double r=a*sin(3*t);
 P.x=r *cos(t);
 P.y=t * sin(t);
 return P;
}
void main( )
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
{
initgr( );
int x0=getmaxx( )/2, y0=getmaxy( )/2;
gr(cicloida, 0, 8*pi, 25, 0, y0);
getch( );
cleardevice( );
gr(desf, 0, 2*pi, 25, x0, y0);
getch( );
cleardevice( );
gr(cardioida, 0, 2*pi, 50, x0, y0);
getch( );
closegraph ( );
}
```

4.3 Elemente de geometrie [5, 6, 47, 68]

În spatiu un corp material este mărginit şi marginea lui se numeşte **suprafaţa corpului.**
O porţiune de pe suprafaţa corpului este mărginită de o **curbă.**
O porţiune de pe curbă începe şi se termină cu un **punct**.
O curbă plană se poate da prin ecuaţii de forma:

- **explicită** $y = f(x)$
- **implicită** $f(x, y) = 0$ (curbe închise)
- **parametrică** $x = f(t), y = g(t)$ (curbe închise sau curbe ce se intersectează)
- **polară** $r = f(\theta)$

Numim ***figură geometrică*** o mulţime de puncte, curbe şi suprafeţe.
Numim ***transformare geometrică*** tridimensională o aplicaţie bijectivă a spaţiului pe el însuşi.
Două figuri geometrice se numesc ***congruente (omoloage)*** dacă se obţin una din alta printr-o transformare goemetrică.

Vom evidenţia ecuatiile a câteva figuri geometrice ce intervin frecvent în grafica pe calculator.

4.3.1 Dreapta

$ax + by + c = 0$, ecuaţia carteziană a dreptei

https://biblioteca-digitala.ro / https://unibuc.ro

$y = mx + n$, ecuatia dreptei sub formă redusă

$x/a + y/b = 1$ ecuația dreptei prin tăieturi

$x\cos\theta + y\sin\theta = p$, forma normală a ecuatiei dreptei, dată de Cauchy în anul 1826.

4.3.2 Cercul

$(x - x_0)^2 + (y - y_0)^2 = r^2$ forma cartezianã a ecuației cercului, ($x^2 + y^2 = r^2$ forma carteziană redusă).

$\begin{cases} x = r\cos\theta \\ y = r\sin\theta \end{cases}$ unde $\theta \in [0,2\pi]$ ecuațiile parametrice ale cercului sub formă redusă

$\begin{cases} x = x_0 + r\cos\theta \\ y = y_0 + r\sin\theta \end{cases}$ unde $\theta \in [0,2\pi]$ ecuațiile parametrice ale cercului sub forma generala

Notând $t = tg\dfrac{\theta}{2}$ obținem:

$$x = \frac{1-t^2}{1+t^2}\cdot r;\quad y = \frac{2t}{1+t^2}\cdot r$$

Cunoscând două puncte (x_n, y_n) şi (x_{n+1}, y_{n+1}) ale cercului avem:

$$\begin{cases} x_n = r\cdot\cos\theta \\ y_n = r\cdot\sin\theta \end{cases}, \quad \begin{cases} x_{n+1} = r\cdot\cos(\theta + d\theta) \\ y_{n+1} = r\cdot\sin(\theta + d\theta) \end{cases}$$

Dezvoltând, obținem:

$$\begin{cases} x_{n+1} = x_n\cdot\cos d\theta - y_n\cdot\sin d\theta \\ y_{n+1} = x_n\cdot\sin d\theta + y_n\cdot\cos d\theta \end{cases},$$

relații recursive ale ecuației cercului

Notând $c = \cos d\theta$ şi $s = \sin d\theta$ obținem $\begin{cases} x_{n+1} = cx_0 - sy_n \\ y_{n+1} = sx_0 + cy_n \end{cases}$, relațiile recursive de ordinul I.

$x^2 + y^2 + ax + by + c = 0$, ecuația generală a cercului

https://biblioteca-digitala.ro / https://unibuc.ro

4.3.2 Conice pe ecuații reduse

4.3.3.1 Elipsa

$\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}=1$, ecuația carteziană a elipsei, unde (x_0, y_0) este centrul elipsei iar a şi b sunt semi axele sale

Punctele $(x_0\text{-}c, y_0)$, (x_0+c, y_0) sunt focarele, unde $c=\sqrt{a^2-b^2}$

$e=\frac{c}{a}$ (excentricitatea) $\in (0,1)$

$\begin{cases} x = x_0 + a\cos\theta \\ y = y_0 + b\sin\theta \end{cases}$ $\theta \in [0,2\pi]$ ecuațiile parametrice ale elipsei

4.3.3.2 Parabola

$y^2 = 2px$ ecuația carteziană a parabolei sub forma redusă

$\begin{cases} x = 2pctg^2\theta \\ y = 2pctg\theta \end{cases}$ $\theta \in (-\pi, \pi)$, ecuațiile parametrice ale parabolei

$y = ax^2 + bx + c$ ecuația generală a parabolei.

4.3.3.3 Hiperbola

$\frac{(x-x_0)^2}{a^2}-\frac{(y-y_0)^2}{b^2}=1$ ecuația carteziană a hiperbolei unde (x_0, y_0) este centrul iar $(x_0\text{-}c, y_0)$, (x_0+c, y_0) sunt focarele.

Aici $c=\sqrt{a^2+b^2}$; iar excentricitatea $e=\frac{c}{a}>1$

$\begin{cases} x = x_0 + a\sec\theta \\ y = y_0 + btg\theta \end{cases}$ $\theta \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$, ecuațiile parametrice ale hiperbolei

$x^2 - y^2 = a^2$ ecuatia unei hiperbole echilatere.

4.3.4 Curbe algebrice clasice şi transcendente (curbe nealgebrice)

Parabola cubică: $y = x^3$

https://biblioteca-digitala.ro / https://unibuc.ro

Parabola semicubică: $y^2 = x^3 \Rightarrow y = \pm x\sqrt{x}, x \geq 0$

Sinusoida $y = \sin x$, în aceeaşi familie: $y = a\sin(kx + \alpha)$

Tangentoida $y = tgx$

Exponenţiala $y = e^x$

Logaritmica $y = \ln x$

4.3.5 Curbe şi suprafeţe

Fie $M(x, y, z)$ un punct în spatiu.

Dacă $x = x(t), y = y(t), z = z(t)$ atunci $M(x, y, z)$ descrie o curbă de ecuaţii :

$$\begin{cases} x = x(t) \\ y = y(t) \\ z = z(t) \end{cases}$$

Dacă $x = x(u, v), y = y(u, v), z = z(u, v)$ atunci $M(x, y, z)$ descrie o suprafaţă de ecuaţii:

$$\begin{cases} x = x(u, v) \\ y = y(u, v) \\ z = z(u, v) \end{cases}$$

Prin eliminare lui u şi v obţinem o ecuaţie de forma: $f(x, y, z) = 0$

Deoarece o curbă este intersecţia a două suprafeţe, ecuaţia unei curbe se poate scrie şi sub forma: $\begin{cases} f(x, y, z) = 0 \\ g(x, y, z) = 0 \end{cases}$

4.4 Bazele matematice ale graficii 2D [6, 46, 68]

Fie $M(x, y) \in 2D$

Ecuaţiile $x' = f(x, y), y' = g(x, y)$ trec $M(x, y)$ la $M'(x', y')$ unde M' este transformatul lui M.

Considerând (x, y) ca o matrice 1×2 luăm:

https://biblioteca-digitala.ro / https://unibuc.ro

$$(x', y') = (x, y)\begin{bmatrix} A & B \\ C & D \end{bmatrix} = (Ax + Cy, Bx + Dy) \Rightarrow \begin{cases} x' = Ax + Cy \\ y' = Bx + Dy \end{cases},$$

se obține o *transformare* care depinde de valorile parametrilor A,B,C şi D.

Translația:

$$T\begin{cases} x' = x + a \\ y' = y + b \end{cases}$$

Observație:

- 1_{R^2} este tanslația obținută pentru $a = b = 0$
- T^{-1} este translație
- $T_1 \circ T_2$ este translație

Rezultă că mulțimea translațiilor formează un grup (T, o).

Observație: Nici un produs cu matricele de tipul $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$ nu conduce la ecuația unei translații.

Pentru a rezolva această situație introducem a treia componentă a unui vector (x, y) obținând $(x, y, 1)$.

Astfel matricea translațiilor are forma $\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ a & b \end{bmatrix}$ deoarece:

$$(x, y, 1)\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ a & b \end{bmatrix} = (x + a, y + b) \Rightarrow x' = x + a, y' = y + b$$

Matricile de dimensiune 3x2 nu sunt inversabile şi de aceea adaugând vectorul $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

obținem matricea: $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix}$ care defineşte o translație pentru că:

https://biblioteca-digitala.ro / https://unibuc.ro

$$(x,y,1)\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & b & 1\end{pmatrix}=(x+a,y+b,1)=(x',y',1)$$

Scalarea:

Este controlată de matricea $T=\begin{pmatrix}A & 0\\ 0 & B\end{pmatrix}$

şi astfel: $(x,y)T=(Ax,By)\Rightarrow x'=Ax, y'=By, A,B>0$

Simetria:

Se obţine din scalare luind A, B$\in$ {-1, 1} cu A şi/sau B=-1

Dacă $T=\begin{pmatrix}-1 & 0\\ 0 & 1\end{pmatrix}\Rightarrow$ simetrie în raport cu $O_y: \begin{cases}x'=-x\\ y'=y\end{cases}$

Dacă $T=\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\Rightarrow$ simetrie în raport cu $O_x: \begin{cases}x'=x\\ y'=-y\end{cases}$

Dacă $T=\begin{pmatrix}-1 & 0\\ 0 & -1\end{pmatrix}\Rightarrow$ simetrie în raport cu $O_z: \begin{cases}x'=-x\\ y'=-y\end{cases}$

Deformarea:

Este controlată de matricea $T=\begin{pmatrix}1 & A\\ B & 1\end{pmatrix}$ cu deformare pe O_x datorată lui A şi deformare pe O_y datorită lui B.

Rotatia: Este dată de formulele $\begin{cases}x'=x\cos\theta-y\sin\theta\\ y'=x\sin\theta+y\cos\theta\end{cases}$

Pentru $\theta=0$ obţinem transformare identică

$\theta\leftarrow-\theta\Rightarrow$ transformarea inversă este rotaţie

$R_1\circ R_2$ este rotaţie

Rezultă că rotaţiile plane formează un grup (R,o)

Observăm că formulele de mai sus se scriu sub forma: $(x',y')=(x,y)\begin{pmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{pmatrix}$

Astfel rotaţia este caracterizată de matricea:

https://biblioteca-digitala.ro / https://unibuc.ro

$$T = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \text{ cu det T=1.}$$

Definiție: Fie G un grup de transformări şi E o expresie ce conține coordonatele unui punct fixat din plan. E este un **invariant** al lui G dacă ea ramâne neschimbată prin înlocuirea acelor coordonate cu cele obținute printr-o transformare din G.
E este ***invariant relativ*** dacă E rămâne neschimbată modulo un factor ce depinde de parametrii transformării.

Observație: Toate transformările prezentate mai sus îl au pe $O(0,0)$ *invariant absolut.*
Concluzie:
Fie matricea: $\begin{pmatrix} A & B & 1 \\ C & D & 1 \\ a & b & E \end{pmatrix}$

- A,B,C,D produc scalări, rotații şi deformări
- a,b produc translații
- E produce schimbarea scării generale

$$(x',y',1) = (x,y,1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & E \end{pmatrix} = (x,y,E) = E\cdot\left(\frac{x}{E},\frac{y}{E},1\right).$$

Astfel $x' = \frac{x}{E},\ y' = \frac{y}{E}$.
Pentru $E>1$ se obține micşorarea scării iar pentru $E<1$ se obține mărirea scării.
Pentru A $\neq$D se obține o schimbare de scală diferită pe cele două axe iar pentru A=D se obține o schimbare uniformă de scală.

Exemplu. Generarea rapidă a cercurilor, elipselor şi hiperbolelor
Ecuația parametrică a cercului de centru (x_0,y_0) este dată de relațiile :
$x=x_0+\rho \cos\theta$
$y=y_0+\rho \sin\theta$

https://biblioteca-digitala.ro / https://unibuc.ro

unde ρ este raza cercului iar θ este un parametru ce ia valori de forma $\theta_k = sk\pi/n$, k=0,...,n, pentru a obține o bună aproximare a cercului prin segmente de dreapta de lungimi egale.
Această reprezentare are dezavantajul că funcțiile sin şi cos se calculează la fiecare pas. Putem evita acest lucru folosind formulele:

$$\begin{cases} \cos(\theta + d\theta) = \cos\theta \cos d\theta - \sin\theta \sin d\theta \\ \sin(\theta + d\theta) = \sin\theta \cos d\theta + \cos\theta \sin d\theta \end{cases}$$

Astfel coordonatele punctelor ce dau aproximarea cercului sunt :

$$x_n = x_0 + \rho \cos\theta_n$$
$$y_n = y_0 + \rho \sin\theta_n$$
$$x_{n+1} = x_0 + \rho \cos\theta_{n+1} = x_0 + \rho \cos(\theta_n + d\theta)$$
$$y_{n+1} = y_0 + \rho \sin\theta_{n+1} = y_0 + \rho \sin(\theta_n + d\theta)$$

Pornind de aici găsim relațiile de recurență:

$$x_n = x_0 + r$$
$$y_n = y_0$$
$$x_{n+1} = x_0 + (x_n - x_0)\cos d\theta - (y_n - y_0)\sin d\theta$$
$$y_{n+1} = y_0 + (x_n - x_0)\sin d\theta + (y_n - y_0)\cos d\theta$$

Analog pentru o elipsă de semiaxe a şi b cu centrul în (x_0, y_0) şi cu axele paralele cu axele de coordonate folosim relațiile de recurență:

$$x_n = x_0 + a$$
$$y_n = y_0$$
$$x_{n+1} = x_0 + (x_n - x_0)\cos d\theta - b/a\,(y_n - y_0)\sin d\theta$$
$$y_{n+1} = y_0 + b/a(x_n - x_0)\sin d\theta + (y_n - y_0)\cos d\theta$$

Dacă elipsa are axele înclinate cu unghiul ϕ față de axele de coordonate folosim transformarea :

$$x = x_0 + (x - x_0)\cos\phi - (y - y_0)\sin\phi$$
$$y = y_0 + (x - x_0)\sin\phi + (y - y_0)\cos\phi$$

unde (x,y) este un punct variabil de pe elipsa de semiaxe a şi b cu centrul în (x_0, y_0) şi cu axele paralele cu axele de coordonate.
Hiperbola raportată la axele ei are ecuațiile parametrice :

$$x = \pm a \sec\theta$$
$$y = \pm b \operatorname{tg}\theta, \text{ unde parametrul } \theta \in [0, \pi/2)$$

Pentru eliminarea calculului repetitiv al funcțiilor trigonometrice folosim formulele:

$$\sec(\theta + d\theta) = 1/(\cos\theta \cos d\theta - \sin\theta \sin d\theta)$$
$$\operatorname{tg}(\theta + d\theta) = (\operatorname{tg}\theta + \operatorname{tg} d\theta)/(1 - \operatorname{tg}\theta \operatorname{tg} d\theta),$$

care ne conduc la următoarele relații de recurență:

https://biblioteca-digitala.ro / https://unibuc.ro

$x_{n+1} = \pm a\, x_n / (b \cos d\theta - y_n \sin d\theta)$
$y_{n+1} = \pm b\, (y_n + b \operatorname{tg} d\theta) / (b - y_n \operatorname{tg} d\theta)$

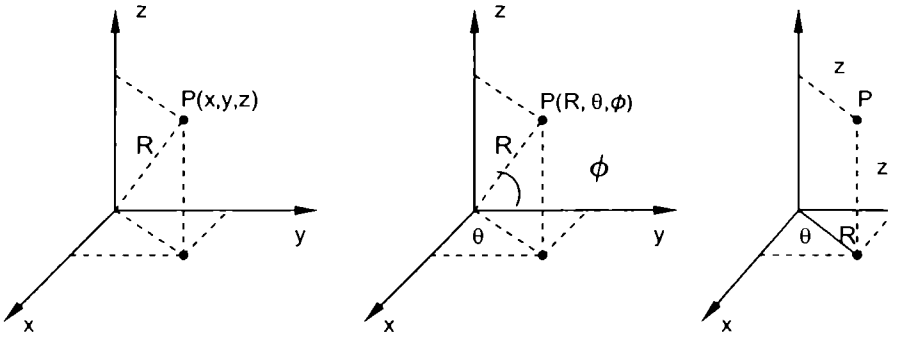
Folosind aceste considerații vom scrie în continuare un program pentru desenarea rapidă a cercurilor. Analog se pot scrie programe pentru desenarea elipselor şi hiperbolelor.

```
#include <graphics.h>
#include <conio.h>
#include <math.h>
#include < iostream.h>
void cerc( int xc, int yc, int r)
{
  float dt=3.141695/(2*r), sint=sin(dt), cost= cos(dt), x,y, xn ,yn;
  x=xc+r;
  y=yc;
moveto(x,y);
int k=4*r;
for (int i=1;i<k;i++) {
                        xn= xc+(x – xc)* cost – (y – yc)*sint;
                        yn= yc+(x – xc)* sint + (y – yc)*cost;
                        lineto (xn ,yn);
                        x= xn;
                    y= yn;
                    }
lineto(xc+r,yc);
}
void main( )
{
 int gd=0, gm;
 detectgraph(&gd, &gm);
 initgraph(&gd,&gm, "c;\\CPP\\bgi");
 cerc(200,200,75);
 getch();
 closegraph( );
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

4.5 Bazele matematice ale graficii 3D [46, 68, 83]

În spaţiu se lucrează cu ***coordonate carteziene, sferice şi cilindrice:***



carteziene sferice cilindrice

Aici $R \geq 0$, $\theta \in [0,2\pi]$ - longitudine

$$\varphi \leftarrow \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] \text{ - latitudine}$$

Trecerea de la coordonate carteziene la cele sferice se face după formulele:

$$R = \sqrt{x^2 + y^2 + z^2}, \theta = \begin{cases} \frac{\pi}{2}, x = 0, y > 0 \\ \frac{3\pi}{2}, x = 0, y < 0 \\ arctg\frac{y}{x} + k\pi, x \neq 0 \end{cases} \quad \text{unde } k = \begin{cases} 0 & x, y > 0 \\ 1, & x < 0 \\ 2 & x > 0, y < 0 \end{cases}$$

$$\varphi = \begin{cases} -\frac{\pi}{2}, x^2 + y^2 = 0, z < 0 \\ \frac{\pi}{2}, x^2 + y^2 = 0, z > 0 \\ arctg\left(\frac{z}{\sqrt{x^2 + y^2}}\right), x^2 + y^2 \neq 0 \end{cases}$$

https://biblioteca-digitala.ro / https://unibuc.ro

Trecerea de la coordonate sferice la cele carteziene se face prin formulele:

$$\begin{cases} x = R\cos\theta\cos\varphi \\ y = R\sin\theta\cos\varphi \\ z = R\sin\varphi \end{cases}$$

Coordonatele cilindrice ale lui P sunt: cota (Z), raza (R), unghiul (θ) cu $Z \in R, R \geq 0, \theta \in [0,2\pi]$

Trecerea de la coordonate carteziene la coordonatele cilindrice:

$$\begin{cases} R = \sqrt{x^2 + y^2} \\ Z = Z \\ \theta \text{ este acelasi ca la coordonatele sferice.} \end{cases}$$

Trecerea de la coordonate cilindrice la cele carteziene:

$$\begin{cases} x = R\cos\theta \\ y = R\sin\theta \\ Z = Z \end{cases}$$

Transformari 3D

Fie F: $R^3 \to R^3$ de forma $(x',y',z') = F(x,y,z,k)$ unde k este un parametru.

Pentru diverse valori ale parametrului k obţinem o mulţime de transformări care formează un grup dacă:

- $(\exists)i$ astfel încât $F(X,i) = X$, $(\forall)X = (x,y,z) \in R^3$
- $(\forall)F(X,k) = X', (\exists)F^{-1}(X',L) = X$ astfel încât ecuatia $\phi(k,L) = i$ se poate rezolva în necunoscuta L.
- Pentru $X' = F(X,k)$ şi $X'' = F(X',L), (\exists)M = \phi(k,L)$ astfel încât $X'' = F(X,M)$ pentru $(\forall)X = (x,y,z) \in R^3$.

Orice $P \in R^3$ are coordonatele carteziene (x,y,z) şi coordonatele omogene $(x,y,z,1)$ ce se pot scrie ca matrice 1×4. Astfel transformările vor fi reprezentate de matrice 4×4 (a patra coloană se adaugă pentru a putea calcula *inversa* matricei).

https://biblioteca-digitala.ro / https://unibuc.ro

Scalarea:

Este dată de matricea $A=\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ pentru că:

$$(x,y,z,1)\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = (ax,by,cz,1) = (x',y',z',1).$$

Dacă $a = b = c \Rightarrow$ scalare uniformă.

Rotația:

- În jurul lui Ox este dată de matricea $R_{Ox} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- În jurul lui Oy este dată de matricea $R_{Oy} = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- În jurul lui Oz este dată de matricea $R_{Oz} = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

În toate cazurile rotația se face în sens pozitiv (trigonometric).

Translația:

Este dată de matricea: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ m & n & p & 1 \end{pmatrix}$

https://biblioteca-digitala.ro / https://unibuc.ro

$$M(x,y,z) \to M'(x+m, y+n, z+p)$$

Distanta pe care s-a făcut deplasarea este $d = \sqrt{m^2 + n^2 + p^2}$

Simetriile:

În raport cu planele xoy, xoz, yoz şi dreptele Ox, Oy, Oz sunt date de matricele:

$$S_{xoy} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$S_{xoz} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}; S_{yoz} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$S_{Ox} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}; \qquad S_{Oy} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$S_{Oz} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Simetria în raport cu $O(0,0,0)$ este dată de matricea

$$S_0 = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

În general pentru simetria față de un punct P se face o translație a sistemului de coordonate astfel încât O să ajungă în P, se ia apoi simetria față de origine şi în final se aplică translația inversă coordonatelor punctului simetric.

https://biblioteca-digitala.ro / https://unibuc.ro

Simetria faţă de un plan oarecare α este dată de o translaţie şi o rotaţie aplicate sistemului dat astfel încât un plan de coordonate să se suprapună peste cel dat, se ia simetria faţă de acest plan de coordonate şi în final se face rototranslaţia inversă.

Simetria faţă de o dreaptă oarecare (d) se obţine într-un mod analog. Se aplică sistemului de referinţă o rotatranslaţie astfel încât o axă de coordonate să se suprapună peste (d), se ia simetria în raport cu această axă şi apoi se aplică rototranslaţia inversă.

Rotaţia în jurul unei drepte $pqIIOx$

Căutăm matricea de rotaţie sub unghiul θ în jurul dreptei $pqIIOx, p(a,b,c), q(d,e,f)$.

Pentru aceasta:

- Facem translaţia T care duce p în O
- Facem rotaţia R de unghiul θ în jurul lui Ox
- Facem translaţia T^{-1} care readuce p la poziţia iniţială

Se obţine:

$$M = T^{-1}RT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & -c & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{pmatrix}$$

Observaţie.

- Dacă T este matrice de translaţie de parametri (a,b,c), T^{-1} se obţine înlocuind a,b,c cu $-a,-b,-c$
- Dacă S este matrice de scalare de parametri (a,b,c), S^{-1} se obţine înlocuind a,b,c cu $\frac{1}{a},\frac{1}{b},\frac{1}{c}$
- Dacă R este matrice de rotaţie de unghi θ, R^{-1} se obţine înlocuind θ cu $-\theta$

Exemplu. [68]

În programul următor prezentăm un exemplu simplu, care surprinde în dinamică rotirea unei case în funcţie de unghiurile şi distanţa de vizualizare. Figura , reprezentând o casă, este memorată prin nişte liste de puncte, care la

https://biblioteca-digitala.ro / https://unibuc.ro

rândul lor sunt asamblate în lista fețelor. Coordonatele utilizator ale colțurilor sunt preluate din fișierul de intrare şi stocate într-un tablou bidimensional. Ele intră în rotiri controlate de utilizator prin taste direcționale. Prin modificarea punctului de vizualizare ,dat prin coordonatele cilindrice (θ, ϕ, R), se tratează fețele ascunse calculând produsul scalar dintre normalele la o suprafață a figurii şi vectorul de vizualizare. Proiecția şi scalarea în 2D a fețelor vizibile permit determinarea imaginii plane corespunzătoare figurii din spațiu.

Programul este următorul:

```
# include <stdio. h>
# include <process.h>
# include < math.h>
# include < alloc.h>
# include < graphycs.h>
# include < conio.h>
# include < iostream.h>
#define PI 3.141592
struct punct {
        int  pct;
        struct  punct  *urm;
         };
           void  calcp( double w1, double w2, double w3, double w4,
           double v1, double v2, double v3, double v4,double &ax,
           double &ay, double &bx, double &by)
{
ax = (v2 – v1)/(w2 – w3); bx = (v1*w2 – v2*w1)/(w2 – w1);
ay = (v4 – v3 )/(w4 – w3); by = (v3*w4 – v4*w3)/(w4 – w3);
}
void adaug(struct punct* &fata, int  pf)
{
struct  punct temp,*nou;
nou = (struct punct*) new(sizeof(struct punct));
nou → pct = pf; nou → urm = NULL;
if ( fata= =NULL) fata = nou;
         else {
         temp = fata
         while ( temp →  urm ! = NULL) temp = temp → urm;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        temp → urm = nou;
        }
}
main ( )
{
FILE *pf;
 double tx[50],ty[50],w1,w2,w3,w4,v1,v2,v3,v4,ax,ay,bx,by;
 int bg[50][50],dt,dm,n,m,i,j,pfi,ad,a1,a2,a3,gata;
 double xr,yr,x0,y0,z0,x,y,z,t,f,r,xd,yd,mx,my,mz,o1,o2,o3, ty,fg;
 double plg[50][3], px,py,x1e;y1e;x2e,y2e,st,sf,ct,cf,p1,p2,p3;
 double q1,q2,q3,prod_scal,v01,v02,v03;
 struct punct *temp, *fata[50];
 char c
 if (( pf = fopen ("td1.dat","rt")) = = NULL) {
                cout <<"Fişier inexistent !!!"<< endl; exit (1);
 fscanf (pf,"%d%d",&n,&m);
 mx=my=mz=0;
 for ( i=0 ,i<n,i++ ){
          fscanf(pf,"%lf%lf%lf",&x,&y,&z);
          plg[i][0] =x; plg[i][1] =y;plg[i][2] =z;
          if ( x>mx) mx=x; if (y>my) my=y; if ( z>mz) mz=z;
          }
 for ( i=0 ; i<m ;i++) {
          fata[i] = NULL;
          do {
              fscanf (pf,"%d", &pfi);
              if ( pfi) adaug ( fata[i],pfi);
              } while (pfi);
           }
 for (i=0; i<n; i++) {
         plg[i][0] = plg[i][0] - mx/2; plg[Ii][1]= plg[i][1] - my/2;
         plg[i][2] = plg[i][2] - mz/2;
         r = 10*sqrt(mx*mx + my*my + mz*mz);
 cout <<"Unghiul t="; cin >>tg; cout <<"Unghiul f="; cin>>fg;
 gata=0; dt= DETECT; initgraph(&dt,&dm,"c:\\cpp\\bgi";
 setbkcolor(WHITE);setcolor(BLUE);
 do {
     t = PI*tg/180; f=PI*fg/180;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
st=sin(t);sf=sin(f);ct=cos(t);cf=cos(f);
o1=r*cf*ct;o2==r*cf*st;o3=r*sf;
w1=1e20;w2=-1e20;w3=1e20;w4=-1e20;
for ( i=0; i<n; i++) {
        for( j=0;j<n;j++) bg[i][j]=1;
       x = plg[i][0] ; y = plg[i][1] ; z = plg[i][2] ;
       x0 = - x*st + y*ct;
       y0 = - x*ct*sf – y*st*sf + z*cf;
       z0 = - x*ct*cf – y*st*cf – z*sf*r;
       xr = r*x0/z0; yr = r*y0/z0; tx[i] = xr; ty[i] = yr;
       if(w1 > xr)  w1 = xr; if (w2 < xr ) w2 = xr; if ( w3 > yr) w3 = yr;
       if ( w4 < yr) w4 = yr ;
       }
v1 = 10 ; v2 = getmaxx ( )-10; v3 = 10; v4 = getmaxy ( )-10;
calcp (w1,w2,w3,w4,v1,v2,v3,v4,ax,ay,bx,by);
for ( j=0; j<n;j++) {
    a1 = fata[j] → pct; ed = a1; a2 = fata[j] → urm→ pct;
    a3 = fata[j]→ urm→ urm→ pct;
    p1 = plg [a2-1][0] – plg[a1-1][0]; p2 = plg[a2-1][1] – plg[a1-1][1];
    p3 = plg[a2-1][2] – plg[a1-1][2]; q1 = plg[a3-1][0] – plg[a1-1][0];
    q2 = plg[a3-1][1] – plg[a1-1][1]; q3 = plg[a3-1][2] – plg[a1-1][2]
   v01 = o1 – plg[a1-1][0]; v02 = o2 – plg[a1-1][1]; v03 = o3 –
                                        plg[a1-1][2];
            prod_scal = v01*(p2*q3 – q2*p3) + v02*(p3*q1 –
        q3*p1) + v03*(p1*q2 –q1*p2);
        if (prod_scal > 0) {
              x1e = ax*tx[a1-1] + bx; y1e = v4 – ay*ty[a1-1] – by;
              xd = x1e; yd = y1e; temp = fata[j];
              while ( temp→ urm!=NULL) {
                     temp = temp→ urm; a2 = temp→ pct;
                    x2e = ax*tx[a2-1] + bx;
                    y2e = v4 – ay*ty[a2-1] – by;
              if ( bg[a1-1][a2-1] && bg[a2-1][a1-1] ){
                    line ((int)x1e, (int)y1e , (int)x2e, (int)y2e);
                    bg[a1-1][a2-1] = 0;
                    }
              a1 = a2; x1e = x2e; y1e = y2e;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                    }
        a2 = ad; x2e = xd; y2e = yd;
        if (bg[a1-1][a2-1] &&bg[a2-1][a1-1])  {
                    line ((int)x1e, (int)y1e,(int)x2e,(int)y2e);
                    bg[a1-1][a2-1] = 0;
                     }
         }
}
do {
   c = getch( );
   switch ( c){
            case 27 : gata =1;
                      break;
            case `\0` :{
                       c= getch ( );
                        switch ( c) {
                              case 72 : {fg+ = 10;cleardevice ( );}break;
                              case 80 : {fg- = 10; cleardevice ( ); }break;
                              case 75 : { tg- = 10;cleardevice ( );}break;
                              case 77 : {tg+ = 10;cleardevice ( );}break;
                               }
                       }
                  }
 } while ( c! = 27 && c! = 72 && c! = 80 && c! = 75 && c! = 77);
 } while ( !gata);
closegraph ( );
}
```

Structura fişierului td1.dat este următoarea:

```
18
11
4   1   0
4   4   0
1   4   0
1   1   0
4   4   2
1   4   2
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
1    1    2
4    1    1
2.5  2    3
2.5  3    3
4    2.25 0.1
4    2.75 0.1
4    2.75 1.5
4    2.25 1.5
3    4    1
2    4    1
2    4    1.75
3    4    1.75
1    4    3    2
1    2    5    8
2    3    6    5
3    4    7    6
1    8    7    4    0
8    5    10   9    0
5    6    10   0
6    7    9    10   0
7    8    9    0
11   12   13   14   0
15   16   17   18   0
```

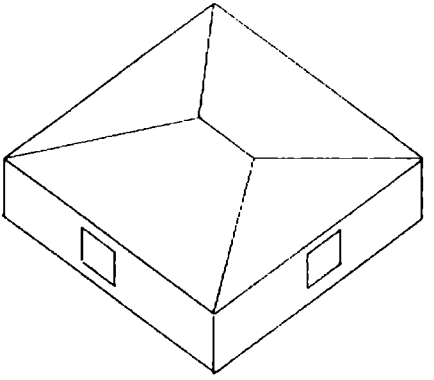


Fig. 2.7 Rotatia unui corp convex

4.6 Obiecte grafice [6, 68, 83]

Un sistem grafic este complet definit prin obiectele grafice elementare ce îl compun şi prin operaţiile ce se pot aplica acestora.
Obiectele sistemelor grafice sunt:
1.**Punctul**, obiect dat prin coordonatele sale (x, y, z) în 3D
2.**Segmentul**, obiect format din mulţimea punctelor de pe o dreaptă cuprinse între două puncte: $P(x_{\min}, y_{\min}, z_{\min}), Q(x_{\max}, y_{\max}, z_{\max})$,
Astfel: $(\forall)X \in PQ, X = (x, y, z)$ avem $x_{\min} \le x \le x_{\max}$, $y_{\min} \le y \le y_{\max}$ şi $z_{\min} \le z \le z_{\max}$.

https://biblioteca-digitala.ro / https://unibuc.ro

3.**Cercul** în R^3 este dat prin 3 puncte necolineare din care două sunt extremităţile unui diametru iar al treilea este pe circumferinţa cercului pentru a defini planul cercului. Cercul, raza şi ecuaţia planului se calculează uşor din aceste 3 puncte.

4.**Elipsa** în R^3 este dată prin 3 puncte necolineare: două focare şi un punct pe circumferinţă. Aceste 3 puncte definesc planul elipsei, distanţa focală şi curba eliptică.

5.**Poliliniile** în $3D$ se definesc prin n varfuri $(x_i, y_i, z_i), i = \overline{1,n}$ astfel încât segmentele ce definesc poliliniile nu sunt toate coplanare.

6.**Curbele spline**. Curbele spline în $3D$ sunt curbe care aproximează o polilinie şi deci este suficient să dăm cele n puncte de la polilinii.

7.**Triunghi şi suprafete poligonale**. Suprafeţele poligonale sunt alăturari de trinunghi în acelaşi plan.

8.**Sfera** se defineşte prin două puncte (centru şi un punct pe sferă) sau două extremităţi ale diametrului secţiunii maximale.

9.**Elipsoid, sferoid şi suprafete cuadrice**:

Elipsoidul este dat de: $\left(\frac{x-x_c}{a}\right)^2 + \left(\frac{y-y_c}{b}\right)^2 + \left(\frac{z-z_c}{c}\right)^2 = 1$ unde (x_c, y_c, z_c) este centrul iar a, b, c sunt parametrii ce reprezintă dimensiunile elipsoidului pe semiaxele $O\,x, O\,y$, respectiv Oz.

Dacă doi parametrii sunt egali atunci elipsoidul se numeşte **sferoid**.

Dacă $a = b = c$ obţinem ***sfera.***

Suprafeţele cuadrice – generalizări ale elipsoidului şi sferoidului, au ecuaţiile:

$$ax^2 + by^2 + cz^2 + dyz + exz + fxy + gx + hy + kz + l = 0$$

10.**Suprafetele generale** în 3D au ecuaţiile $\begin{cases} x = f(u,v) \\ y = g(u,v) \\ z = h(u,v) \end{cases}$, u, v parametrii suprafeţei curbilinii.

Parametrii u şi v pot fi aleşi ca sistem de coordonate iar curbele obţinute pentru u=ct sau pentru v=ct pot fi trasate pe suprafaţă.

Familia de curbe pentru u=ct (şi analog pentru v=ct.) poate fi trasată drept *curbe spline* iar suprafaţa se numeşte *suprafaţă spline* şi trasată cu ajutorul curbelor spline.

https://biblioteca-digitala.ro / https://unibuc.ro

11.**Parabola spline** este o reuniune de arce parabolice obţinute din punctele de control $P_1, P_2, ..., P_n$.

Produsele informatice care oferă grafică dispun de instrumente care elimină discontinuităţile unei curbe, instrumente bazate pe folosirea unui set de *puncte de control.* Curba trasată interactiv va trece prin aceste puncte sau printr-o vecinătate controlată a lor asigurând totodată şi proprietatea de netezire. În cazul nostru primul arc trece prin P_1, P_2, P_3 al doilea prin P_2, P_3, P_4, s.a.m.d ultimul trece prin P_{n-2}, P_{n-1}, P_n.

Pentru a găsi formula unui arc parabolic luăm ecuaţia $y = ax^2 + bx + c$ şi înlocuim în ea coordonatele punctelor din plan prin care trece parabola, obţinând pentru punctele $P_1(x_1, y_1), P_2(x_2, y_2), P_3(x_3, y_3)$.

$$V\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \Rightarrow \begin{pmatrix} a \\ b \\ c \end{pmatrix} = V^{-1}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

unde $V=V(x_1,x_2,x_3)$ este matricea determinantului Vandermonde.

Din P_2, P_3, P_4 obţinem parametrii celui de al doilea arc parabolic a', b', c'. Trasând arcul din prima parabolă de la P_1 la P_2 şi arcul din a doua parabolă de la P_2 la P_3 s.a.m.d obţinem curba spline căutată.

Observăm că în P_2 tangenta la primul arc are panta $m = 2ax_2 + b$ iar la al doilea arc are panta $m' = 2a'x_2 + b'$. Aceste pante nu sunt neapărat egale şi deci curba obţinută nu este netedă.

O altă metodă este să găsim a,b,c pentru arcul de la P_1 la P_2 şi de a păstra panta din P_2. Următorul arc este determinat de P_2 şi P_3 şi de panta din P_2. Analog se determină celalalte arce.

În acest caz ecuaţiile sunt:

$$\begin{cases} y_2 = a'x_2^2 + b'x_2 + c' \\ y_3 = a'x_3^2 + b'x_3 + c' \\ m = 2a'x_2 + b' \end{cases}$$

12.Cubicele spline sunt o extensie a parabolelor spline. Ele sunt secvenţe de cubice şi nu de parabole.

https://biblioteca-digitala.ro / https://unibuc.ro

Ecuația cubicei este: $y = ax^3 + bx^2 + cx + d$

Pentru determinarea parametrilor a,b,c,d sunt necesare punctele de control: P_1, P_2, P_3, P_4.

Obținem $$V\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \Rightarrow \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = V^{-1}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

unde V=V(x_1, x_2, x_3, x_4) este matricea determinantului Vandermonde.

Dupa găsirea lui a,b,c,d se trasează arcul P_1P_2, apoi P_2P_3 cu a', b', c', d' obținute din punctele P_2, P_3, P_4, P_5 etc.

Pentru a avea netezire la trasarea curbei, folosim ecuațiile:

$$\begin{cases} a'x_2^3 + b'x_2^2 + c'x_2 + d' = y_2 \\ a'x_3^3 + b'x_3^2 + c'x_3 + d' = y_3 \\ 3a'x_2^2 + 2b'x_2 + c' = m \\ 6a'x_2 + 2b' = s \end{cases}$$

unde:

din prima derivată $m = 3ax_2^2 + 2bx_2 + c$

din derivata secundă $s = 6ax_2 + 2b$

Observație.

Notând $$\Delta = x_3 - x_2 \ , \ M = \frac{y_3 - y_2}{\Delta} \Rightarrow \begin{cases} a' = \left(\frac{M - m}{\Delta} - \frac{s}{2}\right)/\Delta \\ b' = \frac{s}{2} - 3a'x_2 \\ c' = m - 3a'x_2^2 - 2b'x_2 \\ d' = y_2 - a'x_2^3 - b'x_2^2 - c'x_2 \end{cases}$$

13.Curbe Bezier

În 2D curbele se pot descrie explicit prin $y = f(x)$, prin ecuațiile parametrice $x = f(t,.....), y = g(t,.....)$ sau sub formă implicită $F(x,y) = 0$.

https://biblioteca-digitala.ro / https://unibuc.ro

Pe baza lui F se construiesc două clase de curbe netede folosite în grafica pe calculator şi anume: curbele Bezier si cubicele B-spline

a) Curba Bezier este o curbă definită prin n puncte (puncte de control Bezier) $(x_i, y_i), i = \overline{1,n}$ **şi** dată de ecuaţiile:

$$\begin{cases} x(t) = \sum_{k=0}^{n-1} B_{k,n-1}(t) x_{k+1} \\ y(t) = \sum_{k=0}^{n-1} B_{k,n-1}(t) y_{k+1} \end{cases}$$

unde coeficinţii $B_{k,n-1}(t)$ sunt daţi de funcţiile polinomiale Bernstein

$$B_{k,m}(t) = \frac{m!}{k!(m-k)!} t^k (1-t)^{m-k}, t \in [0,1]$$

Se observă că $x(0) = x_1, x(1) = x_n, y(0) = y_1, y(1) = y_n$ şi astfel curbele pleacă din punctul (x_1, y_1) şi ajung în (x_n, y_n).

Curba Bezier nu trece neapărat prin celelalte puncte, dar acestea se menţin într-o bandă îngustă în apropierea curbei.

Procedura Curba-Bezier are nevoie de valoarea lui n şi de coordonatele celor n puncte.

Coeficienţii funcţiilor polinomiale Bernstein se calculează cu procedura Coef-Bez( ) care îi depune intr-un tablou Bez-Coef $[n]$ după formula:

$$\text{Bez-Coef}(k) = \frac{m!}{k!(m-k)!} = \frac{m(m-1)\ldots\ldots\ldots\ldots(k+1)}{(m-k)!}$$

```
Procedure Coef-Bez (m,Bez-Coef)
array Bez-Coef(n)
for k=0,m
    Bez_Coef(k)←1
    for j=m,k+1,-1
       Bez_Coef (k)←Bez_Coef(k)*j
    repeat
    for j=m-k,2,-1
        Bez_Coef(k)←Bez_Coef (k)/j
    repeat
repeat
return
end
```

https://biblioteca-digitala.ro / https://unibuc.ro

Coeficienţii Bernstein $B_{k,m}(t)$ se calculează cu ajutorul coeficienţilor din Bez-Coef prin funcţia:

```
Function Coef_Bern (nm,k,t,Bez_Coef)
array Bez_Coef (n)
bv←Bez_Coef (k)
for i=1,k
   bv←bv*t
repeat
v=1-t
for i=1,nm-k
    bv←bv*t
repeat
Coef_Bern←bv
return
end
```

Cu aceşti coeficienţi putem calcula coordonatele x şi y ale punctelor de pe curba Bezier folosind şi coordonatele celor n puncte de control, prin procedura Bez_Pct

```
Procedure Bez_Pct (n,xa,ya,t,Bez_Coef,x,y)
array Bez_Coef(n)
x←0; y←0
nm=m-1
for k=0,nm
    bv←Coef_Bern (nm,k,t,Bez_Coef)
    x ← x + xa(k + 1) * bv
    y ← y + ya(k − 1) * bv
repeat
return
end
```

Procedura principală va construi punct cu punct elementele curbei Bezier unind punctele consecutive cu linii.

```
Procedure Curba_Bezier (n,xa,ya)
array xa(n),ya(n)
nm←n-1
call Coef_Bez (nm, Bez_Coef)
call Bez_Pct ( n,xa,ya,o,Bez_Coef,x1,y1 )
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        for i=1, nr_pasi
            t=i/nr_pasi
        call Bez_Pct ( n,xa,ya,t,Bez_Coef,x2,y2 )
        call Trasare_linie ( x1,y1,x2,y2 )
                x1 ← x2; y1 ← y2
        repeat
        return
        end
```

Transcrierea algoritmilor în C++

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <iostream.h>
#define POLY_MAX 100
void initial(float *xmax, float *xmin,float *ymax,float *ymin,
        int *nr_pasi)
{
cout<< "Introdu domeniul lui x:"<<"xmin="<<xmax="<<endl;
cin >>xmin>>xmax
cout<< "Introdu domeniul lui y:ymin="<<"ymax="<<endl;
cin>>ymin>>ymax
cout<c"Introdu nr_pasi: "<<endl;
cin>>nr_pasi;
}
void puncte_de_intrare(float xa[ ], float ya[ ],int *n)
{
int i;
cout<<"Introdu nr. Punctelor de control"<<endl;
cin>>n
for (i=0;i<n;i++){
        cout<<"x["<<i+1<<"]="
        cin>>xa[i]
        cout<<"y["<<i+1<<"]="
        cin>>ya[i]
        }
}
void calcul_Coef_Bez(int nm1, float Bez-Coef[ ])
{
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
int k,j;
for (k=0; k<=nm1;k++){
        Bez-Coef [k]=1;
        for (j=nm1;j>=k+1;j--) Bez-Coef[k]*=j;
        for (j=nm1-k;j>=2;j--) Bez-Coef[k]/=j;}
}
float Calcul_Coef-Bern(int nm1, int k, float t, float Bez-Coef[ ])
{
float bv,v; int i;
bv=Bez-Coef[k];
for (i=1;i<=k;i++) bv*=t;
v=1-t
for (i=1; i<=nm1-k;i++)bv*=v;
return bv;
}

void calcul_Bez-Pct(int n, float xa[], float ya[], float t,  float
Bez_Coef[], float &x,float &y)
{
int k, nm1; float bv;
x=0; y=0; nm1=n-1;
for (k=0; k<=nm1; k++) {
bv=Calcul_Coef-Bern(nm1, k, t, Bez-Coef);
x+=(xa[k]*bv);
Y+=(ya[k]*bv);
  }
}

void Desen-Curba-Bezier(int n, float xa[], float ya[], float Bez-
Coef[], int nr_pasi)
{
int i, nm1; float x1,x2,y1,y2,t;
nm1=n-1;
Calcul-Coef-Bez(nm1, Bez-Coef);
Calcul-Bez-Pct(n,xa,ya,0,Bez-Coef,x1,y1);
for (i=0;i<nr_pasi;i++){
        t=(float)(i+1)/(float)nr_pasi;
Calcul-Bez_Pct(n,xa,ya,t,Bez-Coef,x2,y2);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
line (x1,y1,x2,y2)
x1=x2;
y1=y2;
  }
}
void main( )
{
int n, nr_pasi, mode, driver=DETECT,i;
float xmax,ymax,xmin,ymin,Bez-Coef[POLY_MAX];
float xa[POLY_MAX], ya[POLY_MAX];
initial ( &xmax,&xmin,&ymax,&ymin,&nr_pasi);
puncte_de_intrare(xa,ya,&n)
initgraph(&driver, &mode, "d:\\CPP\\bgi");
setbkcolor(WHITE); setcolor(BLUE);
for (i=0;i<n;i++) {
moveto(xa[i]-4,ya[i]-4);
outtext("*");
 };
Desen_curba_Bezier(n,xa,ya,Bez-Coef,nr_pasi);
getch( );
closegraph( );
}
```

b) Cubicele B-Spline constau din poţiuni de curbe cubice date prin ecuaţiile:

$$\begin{cases} x(t) = a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \end{cases}$$

Cel de-al i-lea cubic se obţine obligând curba să treacă prin (x_i, y_i) pentru t=0, şi prin x_{i+1}, y_{i+1} pentru t=1 şi folosind coordonatele punctelor $i-1, i, i+1, i+2$.

Coeficienţii se obţin cu formulele matriceale:

$$\begin{pmatrix} a_x \\ b_x \\ c_x \\ d_x \end{pmatrix} = \frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}\begin{pmatrix} x_{i-1} \\ x_i \\ x_{i+1} \\ x_{i+2} \end{pmatrix}$$

https://biblioteca-digitala.ro / https://unibuc.ro

şi analog:

$$\begin{pmatrix} a_i \\ b_i \\ c_i \\ d_i \end{pmatrix} = \frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}\begin{pmatrix} y_{i-1} \\ y_i \\ y_{i+1} \\ y_{i+2} \end{pmatrix}$$

Programul C++ pentru trasarea cubicelor B-spline este:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <iostream.h>
struct b_spline{
        double ax,bx,cx,dx;
        double ay,by,cy,dy;
        };
int nr_pasi=100,n;
int xa[100], ya[100];
float b_spline_x(float t,b_spline *bsp)
{
return (bsp→ax*t*t*t+bsp→bx*t*t+bsp→cx*t+bsp→dx)/6.;
}
float b_spline_y(float t,b_spline *bsp)
return (bsp→ay*t*t*t+bsp→by*t*t+bsp→cy*t+bsp→dy)/6.;
 }
void initial( )
{
init mode,  driv=DETECT;
initgraph(&mode, &driv, "d\\cpp\\bgi");
setbkcolor(WHITE);
setcolor(BLUE);
}
void Puncte_de_intrare( )
{
int i;
cout<<"Numarul. Punctelor de control ="; cin>>n
for( i=0; i<n; i++) {
          cout << "x["<<i+1<<"]=";cin>>xa[i+1];
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
            cout << "y["<<i+1<<"]=";cin>>ya[i+1];}
                    }
}
void Calcul_Coef_B_Spline(int i, b_spline *bsp)
{
bsp->ax = -xa[i-1]+3*xa[i+1]+xa[i+2];
bsp->bx = 3*xa[i-1]-6*xa[i]+3*xa[i+1];
bsp->cx = -3*xa[i-1]+3*xa[i+1];
bps->dx = xa[i-1]+4*xa[i]+xa[i+1];
bsp->ay = -ya[i-1]+3*ya[i+1]+ya[i+2];
bsp->by = 3*ya[i-1]-6*ya[i]+3*ya[i+1];
bsp->cy = -3*ya[i-1]+3*ya[i+1];
bps->dy = ya[i-1]+4*ya[i]+ya[i+1];
}
void Desen_Curba(float t1,float t2,int s, b_spline *bsp)
{
int i;
float t,delta_t,x1,y1,x2,y2;
x1=b_spline_x(t1,bsp);y1=b_spline_y(t1,bsp); delta_t=(t1-t2)/s;
for(i=1;i<s; i++){
        t=t1+i* delta_t;
        x2=b_spline_x(t,bsp);
        y2=b_spline_y(t,bsp);
        line(x1,y1,x2,y2);
        x1=x2; y1=y2;
        }
}
void Desen_b_Spline(int nr_pasi, b_spline *bsp)
{
int i;
for(i=2;i<n-1;i++){
        Calcul_Coef_B_Spline(i,bsp);
        Desen_Curba(0,1,nr_pasi,bsp);
        }
}
main()
{
b_spline bsp;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
int i;
Puncte_de_intrare();
initial();
for(i=1; i<=n; i++){
        moveto(xa[i]-4, ya[i]-4);
        outtext("*");
Desen_b_Spline(nr_pasi,&bsp);
getch();
}
```

c) Curbele spline 3D sunt o generalizare a celor 2D.
În 3D avem :

$$x \equiv x_1 = f_1(t) \equiv \sum_{k=0}^{p} a_{1jk}\, s^k$$

$$y \equiv x_2 = f_2(t) \equiv \sum_{k=0}^{p} a_{2jk}\, s^k$$

$$z \equiv x_3 = f_3(t) \equiv \sum_{k=0}^{p} a_{3jk}\, s^k$$

unde t este un parametru global ce ia valori de la 0 la t_{max}. Pentru t dat, $j = [t]$, $s = \{t\} = t - j$.
Deci $s \in [0,1)$ iar $j = 0,1,2,\ldots,$ p este gradul polinomului spline (p = 3 pentru cubice spline). $a_{1jk}, a_{2jk}, a_{3jk}$ sunt date de relațiile :

$$a_{1jk} = \sum_{l=1}^{3} M_{kl} G_{1jl} \;\; ; \; a_{2jk} = \sum_{l=1}^{3} M_{kl} G_{2jl} \;\; ; \; a_{3jk} = \sum_{l=1}^{3} M_{kl} G_{3jl} \;\; ,$$

unde $(M_{kl})_{k=1,\ldots,p,l=1}$, este **matricea spline** iar G_{ijk} este numit **vector geometric .**

Astfel , funcțiile spline globale se exprimă prin :

$$x_i = \sum_{k=0}^{p} \sum_{l=1}^{3} M_{kl} G_{ijl} s^k \;\; , \;\; i = 1..3 \; ;$$

https://biblioteca-digitala.ro / https://unibuc.ro

Prin inversarea ordinii de sumare :

$$x_i = \sum_{l=1}^{3}(\sum_{k=0}^{p} M_{kl} s^k) G_{ijl} \quad \text{sau} \quad x_i = \sum_{l=1}^{3} G_{ijl} B_l(s)$$

unde $B_l(s) = \sum_{k=0}^{p} M_{kl} s^k$ este **funcție amestec** .

Pentru un anumit tip de curbe spline M_{kl} _este constantă şi astfel forma curbei este dată de **vectorul geometric** .

Pentru p = 3 , se obțin cubicele spline care cuprind : cubicele spline Hermite , cubicele spline Bézier şi curbele B-spline . Ele sunt date de matricea M şi vectorul geometric următoare :

Curbele spline Hermite au $M = \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ şi vectorul geometric $G_{ij} = \begin{pmatrix} x_i(j) \\ x_i(j+1) \\ \dfrac{dx_i(j)}{ds} \\ \dfrac{dx_i(j+1)}{ds} \end{pmatrix}$;

Curbele spline Bézier au matricea spline $M = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}$ şi vectorul geometric $G_{ij} = \begin{pmatrix} x_i(j-1) \\ x_i(j) \\ x_i(j+1) \\ x_i(j+2) \end{pmatrix}$.

Acestea pot fi extinse la curbe Bézier polinomiale ce folosesc mai mult de 4 puncte de control.

Dacă avem p+1 puncte de control cu coordonatele (C_{1jk} , C_{2jk} , C_{3jk}) pentru k = 0,..,p atunci ecuațiile se rescriu sub forma :

$$x_i(t) = \sum_{k=0}^{p} \frac{p!}{k!(p-k)!} C_{ijk} s^k (1-s)^{p-k}$$, i = 1,..,3 ; j = [t] iar s = t – j .

https://biblioteca-digitala.ro / https://unibuc.ro

14. Suprafeţe spline. Curbele spline 3D se pot generaliza la suprafeţe spline date de ecuaţiile :

$$x \equiv x_1 = f_1(u,v) = \sum_m \sum_l a_{1jl} b_{lkm} s^l t^m$$
$$y \equiv x_2 = f_2(u,v) = \sum_m \sum_l a_{2jl} b_{lkm} s^l t^m$$
$$z \equiv x_3 = f_3(u,v) = \sum_m \sum_l a_{3jl} b_{lkm} s^l t^m$$

sau mai compact

$$x_i = f_i(u,v) = \sum_l \sum_m C_{ijklm} s^l t^m \quad , \; i = 1,2,3$$

unde C_{ijklm} este produsul tensorial al coeficientilor a_{ijl} şi b_{lkm} , $j = [u]$, $s = u - j$; $k = [v]$, $t = v - k$;
Coeficienţii suprafeţelor spline pot fi exprimaţi în funcţie de vectorii geometrici G_{ijq} şi H_{ikr} astfel :

$$C_{ijklm} = \sum_q \sum_r M_{lq} G_{ijq} H_{ikr} M_{mr}$$

Introducând acesti coeficienţi în forma compactă a lui x_i şi rearanjând sumele obţinem :

$$B_{q,r}(s,t) = \sum_l \sum_m M_{lq} M_{m,r} s^l t^m$$

numite funcţiile **amestec ale suprafeţelor .**
Ecuaţiile suprafeţelor vor fi :

$$x_i(u,v) = \sum_q \sum_r B_{q,r}(s,t) G_{ijq} H_{ikr} \quad , \; i = 1,2,3 .$$

Aceste ecuaţii , împreună cu funcţiile amestec $B_{q,r}(s,t)$ şi cu vectorii geometrici G_{ijq} şi H_{ikr} dau forma suprafeţei , obţinând în funcţie de vectorii geometrici suprafeţe spline bicubice , suprafeţe B-spline, etc.

https://biblioteca-digitala.ro / https://unibuc.ro

4.7 Reprezentarea grafică a obiectelor 3D [6, 68]

Pentru a reprezenta grafic un obiect 3D cu ajutorul calculatorului trebuie găsită o formă de codificare numerică a geometriei obiectului care să furnizeze toate datele necesare. Aceste date despre un obiect 3D se memorează prin şiruri de cifre şi caractere stocate în tablouri numerice sau alfanumerice. Un obiect se poate modela prin puncte, prin linii sau curbe, prin suprafeţe plane arbitrare sau prin volume elementare (modelare volumetrică prin voxeli).

În cazul modelării prin suprafeţe plane trebuie aleasă într-un mod convenabil o mulţime de puncte de pe suprafaţa obiectului memorând într-o listă valorile coordonatelor acestor puncte într-un sistem de coordonate ales. Punctele se numesc **vârfuri** iar lista coordonatelor lor se numeşte **listă de vârfuri**. Vârfurile vor fi unite prin linii sau curbe, obţinând reprezentări simplificate ale obiectului.

Modelarea volumetrică a unui obiect se obţine prin alăturarea unei mulţimi de volume elementare precum conuri, piramide, sfere, etc. , între care se operează intersecţii, reuniuni, etc. pentru a se ajunge la forma dorită.

Pentru a obţine o imagine cât mai fidelă a obiectului modelat se poate asocia fiecărei suprafeţe sau volum elementar, atribute de culoare, transparenţă, reflexie, etc.

Se pot analiza diverse poziţii relative ale componentelor unui obiect sau se pot simula mişcările obiectului în raport cu un reper fix, producând astfel un efect de animaţie.

Pentru a realiza imagini de bună calitate şi animaţii cu obiecte complexe, uneori în timp real, sunt necesare pachete de programe performante şi resurse hardware costisitoare.

Deoarece reprezentarea grafică a unui obiect din 3D înseamnă obţinerea unei imagini bidimensionale ne interesează reprezentarea obiectelor 3D pe un plan numit **plan de proiecţie**. Pe ecran se va afişa imaginea obţinută pe planul de proiecţie.

În grafica pe calculator se folosesc frecvent două tipuri de proiecţie : **proiecţia paralelă şi proiecţia perspectivă**.

În aceste proiecţii, imaginea pe planul de proiecţie a unui obiect 3D se formează ducând drepte de proiecţie dintr-un punct numit **centru de proiecţie** prin fiecare punct al obiectului şi reţinând intersecţiile acestora cu planul de proiecţie.

https://biblioteca-digitala.ro / https://unibuc.ro

Pentru proiecția paralelă, centrul de proiecție se află la infinit și astfel dreptele de proiecție sunt paralele, iar pentru proiecția perspectivă centrul de proiecție se află la distanță finită față de obiect.

După direcția de proiecție relativă la planul de proiecție există două tipuri de proiecții paralele : proiecția paralela **ortogonală** sau **ortografică** la planul de proiecție și proiecția paralelă **oblică** în care direcția de proiecție diferă de normala la planul de proiecție.

Dacă notăm cu α, β, γ unghiurile în sens trigonometric dintre axele Ox, Oy, Oz și axa Ox' din planul de proiecție, din câteva considerații geometrice găsim formulele ce dau coordonatele proiecției axonometrice P(x',y') a punctului M(x,y,z) din 3D. Acestea sunt :

$$x' = x\cos\alpha + y\cos\beta + z\cos\gamma$$
$$y' = x\sin\alpha + y\sin\beta + z\sin\gamma$$

Dacă Oz coincide cu Oy, $\gamma = \frac{\pi}{2}$ și formulele se simplifică.

Un caz particular de proiecție axonometrică îl reprezintă **proiectia cabinet** în care dacă planul de proiecție este xOy, proiecția lui Oz pe acest plan face unghiul de 45^0 cu Ox, unitatea de măsura pe Oz fiind jumătatea din unitatea pe Ox și Oy. Formulele proiecției cabinet a punctului M(x,y,z) în P(x',y') sunt :

$$x' = x + \frac{\sqrt{2}}{4} \cdot z$$
$$y' = y + \frac{\sqrt{2}}{4} \cdot z$$

Dacă direcția de proiecție face unghiuri egale cu cele 3 axe ale sistemului de referință se obține proiecția axonometrică numită **izometrică**.

Proiecția perspectivă creaza imagini similare celor din tehnica fotografică, având dezavantajul că denaturează unghiurile și lungimile, deformând astfel obiectele.

În această proiecție liniile obiectului paralele între ele dar neparalele cu planul de proiecție, par să conveargă într-un punct numit **punct de fugă.**

Dacă observatorul este fix, el vede toate obiectele în interiorul unui **con de vedere** în care directoarea, situată într-un plan normal pe direcția de observare, este ușor eliptică având o deschidere verticală de 15^0 și una orizontală de 17^0.

https://biblioteca-digitala.ro / https://unibuc.ro

În grafica pe calculator conul de vedere se inlocuie cu o **piramida de vedere**, ca şi când observatorul ar observa lumea printr-o fereastră dreptunghiulară decupată, într-un plan opac situată la distanța d de observator. Ecranul este de fapt o directoare a piramidei de vedere care împreună cu poziția observatorului, determină complet această piramidă.

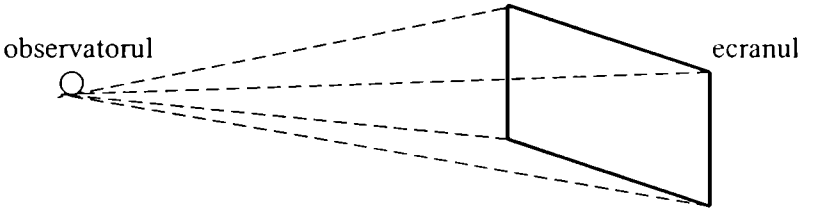


Fig. 2.8 Piramida de vedere

4.8 Aplicații. Programe pentru rezolvarea unor probleme de geometrie în spațiu [6, 68, 83]

Programul foloseşte proiecția oblică a unui obiect 3D pe un plan pe care se formează imaginea propriu-zisă a obiectului, metodă în care direcția de proiecție este diferită de normala la planul de proiecție. Proiecția oblică se poate defini prin punctul din plan M(x',y',0) în care se proiectează M(0,0,1) şi se exprimă analitic prin relații ce depind de distanța ρ de la origine la M' şi de unghiul α făcut de raza OM' cu direcția pozitivă a lui Ox.

Vom avea astfel $x' = \rho \cos\alpha$, $y' = \rho \sin\alpha$, z'=0 .

De aici rezultă că proiecția M'(x',y') a unui punct oarecare M(x,y,z) are coordonatele :

$$\begin{cases} x' = x + \rho \cos\alpha \cdot z \\ y' = y + \rho \sin\alpha \cdot z \\ z' = 0 \end{cases}$$

Pentru a vizualiza un obiect 3D în diverse poziții situat la diferite distanțe de observator se folosesc transformări 3D precum : translație, rotație, mărirea sau micşorarea scării, simetrii, etc.

Pentru a translata un sistem de puncte $M_i(x_i,y_i,z_i)$ din 3D trebuie să precizăm valorile dx,dy,dz pentru deplasarea lor pe direcție Ox, Oy, şi

https://biblioteca-digitala.ro / https://unibuc.ro

respectiv Oz, sistemul de puncte deplasându-se pe distanța

$$d = \sqrt{dx^2 + dy^2 + dz^2}$$

Astfel coordonatele punctelor translatate vor fi :

$x_i'=dx+x_i$,
$y_i'=dy+y_i$,
$z_i'=dz+z_i$

Dacă dorim să rotim un punct M(x,y,z) în planele xOy (axa de rotație Oz), yOz (axa de rotație este Ox), xOz (axa de rotație Oy) cu unghiurile c, a şi respectiv b, sensul de rotație fiind cel al acelor de ceasornic (invers trigonometric) obținem noua poziție M'(x',y',z') dată de relațiile :

x'=xcosc+ysinc, y'=--xsinc+ycosc, z'=z (Oz axa de rotație)
x'=x , y'=ycosa+zsina, z'=-ysina+zcosa (Ox axa de rotație)
x'=-zsinb+xcosb, y'=y, z'=zcosb+xsinb (Oy axa de rotație)

Prin înmulțirea coordonatelor cu diverşi **factori de scară** se modifică distanțele dintre puncte. Printre aceste transformări întâlnim **asemănarea** în care se foloseşte un factor de scară global.

Intr-o asemănare de factor global fgs punctul M(x,y,z) devine M'(x*fgs,y*fgs,z*fgs).

Sistemul de puncte se poate modifica diferit după cele 3 axe ale sistemului de referință, caz în care factorii de scară direcționali sunt să spunem fsx, fsy, fsz. Cu acești factori de scară M(x,y,z) devine într-o asemănare M'(x*fsx,y*fsy,z*fsz).

Vom scrie acum un program pentru construirea unei clase care să ofere suport pentru rezolvarea problemelor de geometrie în spațiu. El va fi salvat în **directorul curent** ca fişier antet graph3D.h .

```
#include<iostream.h>
#include< math.h>
#include<conio.h>
#include<graphics.h>
const float  pi=3.1415;
class punct3D
      { float x,y,z;
        float xe,ye;
        static int xO,yO;
        static float ro,alfa,fact,raport;       //raport=raport spectral
        public:
           punct3D()
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
            {x=y=z=0; xe=0; ye=0;}
          punct3D(float x1,float y1,float z1);
         friend void initgr3D(int  gd=DEFECT, int gm=0, char cale=
       "\Cpp\bgi",float ro1=0.5,float alfa1=pi/4, float fact=1);
          static void setOrig(int x0,int y0)
          {
           xO=x0; yO=y0;}
          static int getOrigX()
           {return xO;}
          static int getOrigY()
           {return yO;}
          static float setfact(float fs)
          { float f=fact;
                  fact=fs;
                  return f;}
           void proiectie();
           punct3D Roy(float beta);
           friend void putpixel(punct3D& P, int color);
           friend void  moveto(punct3D& P);
           friend void line(punct3D& P,punct3D& Q);
           friend void lineto(punct3D& P);
           friend istream& operator >>(istream& s, punct3D& P);
           friend ostream& operator <<(ostream& P, punct3D& P);
      };
   int punct3D::xO, punct3D::yO;
   float punct3D::raport, punct3D::r0,punct3D::alfa, punct3D::fact;
void initgr3D(int gd,int gm,chr* cale,float ro1,float alfa1;float fac-
tor1)
 {int xa,ya;
  initgraph(&gd,&gm,cale);
  getaspectratio(&xa,&ya);
  punct3D::xO=getmaxx()/2;
  punct3D::yO=getmaxy()/2;
  punct3D::raport=float(xa)/ya;
  punct3D::ro=ro1;
  punct3D::alfa=alfa1;
  punct3D::fact=factor1;
 }
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
punct3D::punct3D(float x1,float y1,float z1)
{ x=x1; y=y1; z=z1; proiectie();}
 void punct3D::proiectie()
      {float d=ro*z;
       xe=x+d*cos(alfa);
       ye=ra*(y+d*sin(alfa));
       }
punct3D punct3D::Roy(float beta)
 {
  punct3D Q(-z*sin(beta)+x*cos(beta),y,z*cos(beta)+x*sin(beta));
  Q.proiectie();
  return Q;
 }
void putpixel(punct3D& P,int color)
{putpixel(P.x0+P.fact*P.xe,P.y0-P.fact*P.ye,color);}
 void moveto(punct3D& P)
    {moveto(P.x0+P.fact*P.xe,P.y0-P.fact*P.ye);}
 void line(punct3D& P,punct3D& Q)
   {line(P.x0+P.fact*P.xe,P.y0-P.fact*P.ye,P.x0+P.fact*Q.xe,P.y0-
P.fact*Q.ye);}
 void lineto(punct3D)& P)
 { lineto(P.x0+P.fact*P.xe,P.y0-P.fact*P.ye);}
 istream& operator >>(istream& s,punct3D& P)
 {
  s>>P.x>>P.y>>P.z;
  P.proiectie();
  return s;}
ostream& operator <<(ostream& s,punct3D& P)
 { s<<"("<<P.x<<","<<P.y<<","<<P.z<<")";
  return s;}
```

În continuare vom folosi acest fişier antet pentru rezolvarea unor probleme concrete de geometrie în spaţiu.

1. Program pentru desenarea şi rotirea unei prisme.

```
#include"graph3D.h"
punct3D
P[]={punct3D(50,10,0),punct3D(150,10,0),punct3D(150,10,80),
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
punct3D(50,10,80),punct3D(50,60,0),punct3D(150,60,0),punct3D(150,6
0,80),
            punct3D(50,60,80)};
void desen(punct3D *P)
{ for (int i=0;i<3;i++) {
          line(P[i],P[i+1]); line(P[4+i],P[5+i]); line(P[i],P[4+i]);
          }
line(P[3],P[0]); line(P[7],P[4]); line(P[3],P[7]);
}
void rotirePrisma()
{punct3D Pr[8];
setwritemode(XOR_PUT);
 for (float beta=0; beta<2*pi; beta+=pi/10) {
     for (int i=0;i<8;i++)  Pr[i]=P[i]; Roy(beta);
   desen(Pr);
   getch();
   desen(Pr);}
 }
void main()
{initgr3D();
 punct3D O(0,0,0),X(300,0,0), Y(0,200,0),Z(0,0,200)
 line(O,X); line(O,Y); line(O,Z);
 rotirePrisma();
 getch();
 closegraph();
}
```

2. Program pentru desenarea unei sfere.

```
#include"graph3D.h"
punct3D punctSf(float fi,float t,float raza)
  {punct3D M(raza*cos(fi)*sin(t),raza*sin(fi),raza*cos(fi)*cos(t));
  return M;}
void sfera(int xO,int yO,int raza)
{ punct3D::setOrig(xO,yO);
   punct3D M;
   float pas=pi/10;
   for (float fi=-pi/2;fi<pi/2;fi+=pas) {
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        M=punctSf(fi,0,raza);
        moveto(M);
        for (float t=pas;t<=2*pi; t+=pas) {
         M=punctSf(fi,t,raza);
         lineto(M);
        }
 }
for (float t=-pi/2;t<pi/2;t+=pas) {
       M=punctSf(0,t,raza);
       moveto(M);
       for (float fi=pas;fi<=2*pi;fi+=pas) {
      M=punctSf(fi,t,raza);
      lineto(M);
    }
  }
}void main()
{ initgr3D();
  sfera(200,150,100);
getch();
closegraph();
}
```

4.9 Metode şi funcţii pentru realizare de animaţie [6, 68, 46]

Animaţia este caracteristică perifericelor ce dispun de ştergere selectivă, care presupune ştergerea unui obiect de la o anumită poziţie şi redesenarea şi ştergerea lui în diferite poziţii pe traiectoria de deplasare. Avem animaţie în timp real când viteza mişcării corespunde celei reale şi pseudo animaţie în caz contrar

Algoritmul animaţiei parcurge următoarele etape:

1. desenarea modelului;
2. memorarea porţiunii ce va fi suprascrisă de poziţia iniţială a obiectului animat;
3. desenarea obiectului în poziţia iniţială;
4. calcularea unei poziţii, a unghiului şi formei obiectului în mişcare;
5. restaurarea fundalului în porţiunea din care placă obiectul;
6. actualizarea noilor coordonate ale obiectului;
7. reluarea de la pasul 2) pentru toate poziţiile intermediare.

https://biblioteca-digitala.ro / https://unibuc.ro

Ochiul uman percepe discretizarea mişcării pe paşi dacă succesiunea are loc la un interval mai mare de 1/25 secunde. Deoarece o imagine este vizibilă în paşii 3 şi 4, pentru ca animaţia să aibă loc fără intermitenţe sesizabile, trebuie ca timpul de vizibilitate să fie aproape egal cu timpul total, care este funcţie de numărul de pixeli ce compun obiectul animat. Pentru un număr mic de pixeli desenarea şi ştergerea obiectului nu consumă mult timp şi deci condiţia este îndeplinită.
Se caută astfel numărul maxim de pixeli din imaginea animată astfel încât timpul de vizibilitate/timpul total ≈ 1. Acest număr maxim este funcţie de timpul mediu de citire/scriere pe pixel care este la rândul lui dependent de perifericul grafic folosit.
Viteza animaţiei depinde în mare măsură de durata calculelor noilor poziţii ale obiectului. Dintre poziţiile în care apare obiectul pe ecran, unele sunt poziţii cheie, iar celelalte se calculează prin interpolare pornind de la două poziţii cheie succesive. Această tehnică, numită multiplicarea imaginii presupune calculul coordonatelor $(x_i(t), y_i(t))$, $i = 1,n$ unde n este numărul vârfurilor imaginii animate, iar t este o poziţie oarecare între două poziţii cheie succesive .
Cum animaţia reală nu se poate realiza la viteza şi calitatea cerută de ochiul uman se apelează la trucuri, hard şi soft pentru realizarea unei pseudoanimaţii. Printre acestea întâlnim tehnica manipulării paletei de culori. Aceasta foloseşte o singură imagine compusă din mai multe obiecte ce folosesc coduri diferite de culori. Obiectul animat apare în mai multe poziţii în imagine cu coduri de culori diferite, dar care sunt decodificate iniţial prin paleta de culori, în aceiaşi culoare ca a fondului pe care se deplasează. Rând pe rând, pe sensul de deplasare, culoarea aferentă obiectului animat este relevată comutând intrarea în tabelul de culori pe o culoare diferită de a fondului, apoi revenind pe culoarea fondului.
In programul urmator prezentam simularea deplasării soarelui [68] într-un peisaj cu puţine culori (marea şi cerul). Numărul de culori rămase permite folosirea unui număr mare de poziţii ale soarelui. În acest exemplu, soarele se deplasează pe un arc de elipsă în cele 16 culori. BLUE se foloseşte pentru mare, LIGHT BLUE pentru cer şi YELLOW pentru soare, iar cele 13 culori rămase corespund la 13 „sori" invizibili (de culoarea fondului). Ulterior aceste culori se traduc în YELLOW pentru o cuantă de timp, după care revin la culoarea fondului.

https://biblioteca-digitala.ro / https://unibuc.ro

Programul este următorul:

```
# include <math.h>
# include <graphics.h>
# include <conio.h>
# include < dos.h>
# include <iostream.h>
# define PI  3.141592
static  double  wl,wr,wt,wb;
static  int  vl,vr,vt,vb;
void setwindow ( double   newwl,double newwt, double newwb,double
newwr )
{ wl = newwl; wt = newwt; wb = newwb; wr = newwr; }
`void  setviewport ( int  newvl, int newvt, int newvb, int newvr )
{ vl = newvl; vt = newvt; vb = newvb; vr = newvr; }
void  scalare ( double  *x, double  *y)
{
*x = (vr – vl)/(wr – wl)** x + ( vl*wr – vr*wl)/(wr – wl) ;
*y = (vt – vb)/(wt – wb)* *y + (vb*wt – vt*wb)/(wt – wb);
}
void main ( )
{
int i,a,b;
double x,y,phi;
int  mod, driver = DETECT;
initgraph ( &driver, &mod, "c: \\ cpp \\ bgi");
setpalette ( 15, LIGHTBLUE);
setcolor (15);
rectangle (0,0,getmaxx ( ), 0.8*getmaxy ( ) );
setfillstyle ( SOLID_FIL,15);
floodfill (10,10,15);
a = getmaxx ( )/2; b = 0.8*getmaxy ( );
setpalette (0,BLUE); setcolor (0);
rectangle (0,0.8*getmaxy( ),getmaxx( ),getmaxy( ));
setfillstyle ( SOLID_FILL,0);
for (i=1  ,phi = 7*PI/180. ; i < 14  ; i++ ) {
                        x = a*cos(phi); y = b*sin(phi);
                        setwindow ( -320. , 380. ,0. ,320.);
                        setviewport ( 20,25,360,620);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                    scalare ( &x,&y );
                    setpalette ( i, LIGHTBLUE);
                    setcolor ( i );
                    circle ( x,y,20 );
                    phi+= ( 14.*PI/180.);
                    setfillstyle ( SOLID_FILL, i )
                    floodfill ( x,y,i );
                   }
getch ( );
for ( i=1; i < 14; i-- ) {
                   setpalette ( i, YELLOW );
                   delay  (200);
                   setpalette ( i, LIGHTBLUE);
                   }
closegraph ( );
}
```

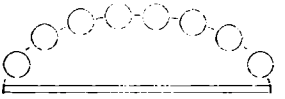


Fig. 2.9

Pe lângă această metodă se mai cunosc următoarele:

- metoda manipulării planelor de biți, aplicată sub forma bufferului dublu sau triplu
- metoda setului de caractere definită de utilizator numita şi tehnica graficii indirecte.

Se poate realiza pseudoanimația [68] nu prin calculul noilor coordonate ale obiectului ci prin decuparea şi mutarea dintr-un loc în altul a unei porțiuni dreptunghiulare de pe ecran care conține obiectul animat. Forma dreptunghiulară se decupează uşor prin colțul stânga-sus şi dreapta-jos. Următorul program ilustrează deplasarea unei biciclete, pedala schimbându-şi poziția pe parcursul deplasării:

```
# include < graphics.h >
# include < alloc.h >
# include < dos.h >
# include < conio .h >
# include < iostream.h >
# define STEP 5
void *ob1, *ob2, *ob3, *ob4;
void drawob(void);
void moveob (void);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
 void main ( )
{
int gmod, gdrive = DETECT;
initgraph ( &gdrive, &gmod, "c: \\ cpp \\bgi ");
setcolor (RED); drawob ( ); getch ( ); moveob ( ); getch ( );
closegraph ( );
}
void drawob (void)
{
void *ped;
circle ( 50,100,25); circle (150,100,25); line (50,100,80,85);
line (80,85,134,85); line (77,82,95,100); line (130,80,150,100);
line (128,80,113,82);line (128,80,116,78); line (72,81,89,81)
line (73,82,90,82);
ob1 = new (imagesize (25,75,175,125));
ob2 = new (imagesize (25,75,175,125));
ob3 = new (imagesize (25,75,175,125));
ob4 = new (imagesize (25,75,175,125));
ped = new (imagesize(85,90,110,110));
circle (95,100,5); line (95,100,130,87);getimage (85,90,110,110,ped);
line (86,100,104,100);line (85,100,87,100);line (103,100,105,100);
getimage (25,75,175,125,ob1);
putimage (85,90,ped,COPY_PUT);
line (88,96,102,104); line (87,96,89,96); line (101,104,103,104);
getimage (25,75,175,125,ob2); putimage (85,90,ped,COPY_PUT);
line (95,95,95,105); line (94,95,96,95); line (94,105,96,105);
getimage (25,75,175,125,ob3); putimage (85,90,ped,COPY_PUT);
line (102,96,88,104); line (101,96,103,96); line (87,104,89,104);
getimage (25,75,175,125,ob4); delete(ped);
}
void moveob (void)
{
int i,c;
for ( i=0, c=0; i< getmaxx( ) – 200,i+=STEP, c++) {
                    switch ( c%4) {
                            case 0: putimage( 25+i,75,ob4,XOR_PUT);
                  putimage(25+STEP+i,75,ob1,XOR_PUT);
                              break;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
                    case 1: putimage(25+i,75,ob1,XOR_PUT);
                    putimage(25+STEP+i,75,ob2,XOR_PUT);
                        break;
                    case 2: putimage(25+i,75,ob2,XOR_PUT);
                    putimage(25+STEP+i,75,ob3,XOR_PUT);
                        break;
                    case 3: putimage(25+i,75,ob3,XOR_PUT);
                    putimage(25+STEP+i,75,ob4,XOR_PUT);
                            break;
                           }
            delay (20);
             }
}
```

Se ştie că **memoria ecran** aparţine plăcii grafice. În ea se păstrează informaţiile care se afişează pe ecran. În realitate imaginea de pe ecran este generată de 50-75 ori/sec.

În modul grafic, memoria ecran cuprinde pentru fiecare pixel un cod reprezentat în biţi prin care se determină culoarea pixelului. Acest mod de codificare face ca memoria ecran să poată conţine imaginea codificată a mai multor ecrane. Zona din memoria ecran necesară memorării unui ecran se numeşte **pagina video.** În funcţie de mărimea memoriei ecran de pe placa grafică se poate memora una sau mai multe pagini video. La fiecare moment o singură pagină video apare pe ecran, numindu-se **pagina vizibilă.**

Dacă placa grafică este de tip EGA(256 KO), VGA sau Hercules, calculatorul poate trimite ieşiri grafice către o pagină din memorie, numită **pagina activă**, fără ca ea să apară pe ecran, urmând ca ulterior conţinutul ei să fie vizualizat la ecran.

Această facilitate permite obţinerea de **animaţie** prin vizualizarea alternativă a paginilor de memorie spre care au fost trimise ieşiri grafice în timp ce alte pagini video sunt afişate pe ecran.

Funcţia **setactivepage** defineşte pagina activă către care sunt îndreptate ieşirile grafice iar funcţia **setvisualpage** permite vizualizarea unei anumite pagini grafice.

Putem grupa tehnicile de realizare a animaţiei in grafica pe calculator in urmatoarele patru clase:

1. Ştergerea desenului si redesenarea lui în culoarea de fond

https://biblioteca-digitala.ro / https://unibuc.ro

2. Afişarea funcţiei **setwritemode(XOR_PUT)** care setează modul de scriere pe ecran pe XOR. Aceasta face ca la primul apel al funcţiilor line, lineto, linerel, drawpoly, rectangle, outtext, outtextxy să se deseneze iar la al doilea apel să se şteargă desenul afişat anterior. În acest caz pentru determinarea valorii unui pixel se face XOR între culoarea veche şi cea curentă a pixelului, şi două operaţii XOR cu aceeaşi valoare reproduc culoarea iniţială : aXORbXORb=a
3. Folosirea funcţiilor **getimage** şi **putimage**. Ultimul parametru al funcţiei putimage poate lua valoarea XOR_PUT şi astfel modul de scriere va fi definit XOR ca şi la funcţia setwritemode.
4. Folosirea funcţiilor de manipulare a paginilor video, pentru modurile garfice cu mai multe pagini video.

Pagina asupra căreia acţionează funcţiile grafice la un anumit moment se numeşte **pagina activă (curentă).** Funcţia setactivepage stabileşte următoarea pagină activă.

Animaţia se obţine prin construcţia imaginii următoare celei curente în timp ce imaginea curentă este afişată în pagina vizibilă şi după aceea pagina activă va fi setată ca pagină vizibilă prin funcţia setvisualpage, iar o altă pagină video va fi setată ca pagină activă în care se va desena o nouă imagine, s.a.m.d .

Vom exemplifica aceste tehnici de realizare a animaţiei scriind pentru fiecare categorie câte un program care roteşte o piramidă patrulateră regulată cu cate 30^0 în jurul înălţimii. [6]

1. Program de animaţie prin ştergerea figurii, redesenând-o în culoarea fondului.

```
#include<iostream.h>
#include<math.h>
#include<conio.h>
#include<graphics.h>
#define PI 3.14
#define N 3
#define LAT 100
#define LAT2  LAT/2
#define H  200
#define D  50
int dy=300, gd,gm,err,xx[N][5],yy[N][5],i,j;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
void pr_cabinet(void);
void desen(int i);
void main()
{
 gd=DETECT;
 initgraph(&gd,&gm,"C:\\CPP\\bgi");
 err=graphresult();
 if (err!=grOk) {
    cout<<"Eroare grafica: "<<grapherrormsg(err))<<endl;
    cout<<"Apasa o tasta pentru oprire : "<<endl;
    getch();
    exit(1);
 }
setlinestyle(DASHED_LINE,0,1);
pr_cabinet();
for (j=0;j<=29;j++)
      for (i=0;i<=N-1;i++)  {
              setcolor(BLUE);
              desen(i);
              delay(200);
              setcolor(getbkcolor());
              desen(i);
             }
  closegraph();
  }
void pr_cabinet(void)
{
 int   x[5]={10,LAT+10,LAT+10,10,LAT2+10, y[5]={D,D,D,D,D+H},
         z[5]={0,0,LAT,LAT,LAT2} , xc=LAT2+10, zc=LAT2, i, j ;
 double s,c,u,r,alfa,x1,z1;
 r=sqrt(2)/4; alfa=PI/2/N;  u=alfa;
 for (j=0;j<=4;j++)  {
        xx[0][j]=(int)(x[j]+r*z[j]);
        yy[0][j]=dy-(int)(y[j]+r*z[j]);
       }
 for (i=1;i<=N-1;i++)  {
       c=cos(u); s=sin(u);
       for (j=0;j<=3;j++)  {
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        x1=xc+(x[j]-xc)*c-(z[j]-zc)*s;
        z1=zc+(x[j]-xc)*s+(z[j]-zc)*c;
       xx[i][j]=(int)(x1+r*z1);
       yy[i][j]=dy-(int)(y[j]+r*z1);
     }
 u=u+alfa;
 xx[i][4]=xx[0][4];  yy[i][4]=yy[0][4];
}
}
void desen(int i)          //desenează piramida din pozitia i
{
 int j;
 moveto(xx[i][0], yy[i][0]);
 for (j=1;j<=3;j++)  lineto(xx[i][j],  yy[i][j]);
 lineto(xx[i][0], yy[i][0]);
 for (j=0;j<=3;j++)  {
      moveto(xx[i][4],yy[i][4]);
      lineto(xx[i][j]), yy[i][j]);
}
}
```

2. Program de animație prin setarea scrierii la ecran pe XOR.

```
#include<graphics.h>
#include<iostream.h>
#include<math.h>
#include<conio.h>
#define PI 3.14
#define N 3
#define LAT 100
#define LAT2  LAT/2
#define H 200
#define D 50
int dy=300,gd,gm,err,xx[N][5],yy[N][5],i,j;
void pr_cabinet(void);
void main()
{
 gd=DETECT;
 initgraph(&gd,&gm,"C:\\CPP\\bgi ");
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
     err=graphresult();
     if (err!=grOk)  {
         cout<<"Eroare grafica :"<<grapherrormsg(err)<<endl;
         cout<<"Apasa o tasta pentru oprire :"<< endl;
        getch();
        exit(1);
      }
    setwritemode(XOR_PUT);setcolor(BLUE);
    setlinestyle(DASHED_LINE,0,1);
    pr_cabinet();
    for (j=0;j<=29;j++)
        for (i=0;i<=N-1;i++)  {
                    desen(i);
                    delay(200);
                    desen(i);
                }
     closegraph();
   }
void pr_cabinet()
{
 int  x[5]={10,LAT+10,LAT+10,10,LAT2+10},  y[5]={D,D,D,D,D+H},
        z[5]={0,0,LAT,LAT,LAT2} , xc=LAT2+10, zc=LAT2, i, j ;
 double s,c,n,r,alfa,x1,z1;
 r=sqrt(2)/4; alfa=PI/2/(double)N;  u=alfa;
 for (j=0;j<=4;j++)  {                          //Proiecția piramidei inițiale
      xx[0][j]=(int)(x[j]+r*z[j]);
      yy[0][j]=dy-(int)(y[j]+r*z[j]);
      }
 for (i=1;i<=N-1;i++)  {
     c=cos(u); s=sin(u);
     for (j=0;j<=3;j++)  {                       //rotește cu alfa vârfurile bazei
       x1=xc+(x[j]-xc)*c-(z[j]-zc)*s;
       z1=zc+(x[j]-xc)*s+(z[j]-zc)*c;
      xx[i][j]=(int)(x1+r*z1);                   //coordonatele proiecției
      yy[i][j]=dy-(int)(y[i]+r*z1);
     }
 u=u+alfa;
 xx[i][4]=xx[0][4];  yy[i][4]=yy[0][4];      //vârful nu se rotește
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
 }
}
void desen()                              //desenează piramida din poziția i
{
 int j;
 moveto(xx[i][0], yy[i][0]);
 for (j=1;j<=3;j++)  lineto(xx[i][j],  yy[i][j]);
 lineto(xx[i][0], yy[i][0]);
 for (j=0;j<=3;j++)  {
      moveto(xx[i][4],yy[i][4]);
      lineto(xx[i][j]), yy[i][j]);
 }
```

3. Program de animație obținut memorând imaginea în getimage() scriind-o apoi cu putimage() și având al 4-lea argument XOR_PUT.

```
#include<alloc.h>
#include<iostream.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
#define PI  3.14
#define  N  3
#define LAT 100
#define LAT2   LAT/2
#define H 200
#define D  50
int dx=155, dy=300, gd,gm,err,xx[5],yy[5],i,j;
void far *mem[N];
void gen_im();
void desen();
void main()
{
 gd=DETECT;
 initgraph(&gd,&gm,"C:\\CPP\\bgi ");
 err=graphresult();
 if (err!=grOk)  {
        cout<<"Eroare grafica :"<<grapherrormsg(err)<<endl;
        cout<<"Apasa o tasta pentru oprire :"<< endl;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        getch();
        exit(1);                    //oprire cu cod de eroare
      }
setlinestyle(DASHED_LINE,0,1);setcolor(BLUE);
gen_im();
delay(50);putimage(0,0,mem[2],XOR_PUT);
for (j=0;j<=28;j++)
      for (i=0;i<=N-1;i++)  {
            putimage(0,0,mem[i],XOR_PUT);     //desenează piramida
            delay(100);
            putimage(0,0,mem[i],XOR_PUT);     //şterge piramida
          }
   closegraph();
}
void gen_im()            //generează proiecţiile vârfurilor pentru 3 poziţii
{
 int x[5]={10,LAT+10,LAT+10,10,LAT2+10},
y[5]={D,D,D,D,D+H},z[5]={0,0,LAT,LAT,LAT2},xc=LAT2+10,zc=LAT2,i,j;
 double s,c,u,r,alfa,x1,z1;
 r=sqrt(2)/4; alfa=PI/2;  u=alfa;
 for (j=0;j<=4;j++)  {
       xx[j]=(int)(x[j]+r*z[j]);
       yy[j]=dy-(int)(y[j]+r*z[j]);
       }
 desen();                          //desenează piramida în poziţia iniţială
 dim=imgesize(0,0,dx,dy);                  //dimensiunea imaginii
 if ((mem[0]=farmalloc(dim))==NULL)  {
          closegraph();
          cout<<"NU este suficient spatiu heap"<<endl;
          exit(1);
         }
 getimage(0,0dx,dy,mem[0]);              //salvează imaginea în mem[0]
 for (i=1;i<=N-1;i++)  {
         c=cos(u);   s=sin(u);
      for (j=0;j<=3;j++)  {               //roteşte cu alfa vârfurile bazei
        x1=xc+(x[j]-xc)*c-(z[j]-zc)*s;
        z1=zc+(x[j]-xc)*s+(z[j]-zc)*c;
        xx[j]=(int)(x1+r*z1);                //coordonatele proiecţiei
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
        yy[j]=dy-(int)(y[j]+r*z1);
      }
 putimage(0,0,mem[i-1],XOR_PUT);          //şterge imaginea
 desen();
  if ((mem[i]=formalloc(dim))==NULL)  {
              closegraph();
              cout<<"Eroare din lipsa de spatiu heap"<<endl;
              exit(1);
             }
 getimage(0,0,dx,dy,mem[i]);                //salvează imaginea în mem[i]
 u=u+alfa;
}
}
void desen()                                  //desenează piramida
{
 int j;
 moveto(xx[0], yy[0]);
 for (j=1;j<=3;j++)  lineto(xx[j],  yy[j]);
 lineto(xx[0], yy[0]);
 for (j=0;j<=3;j++)  {
      moveto(xx[4],yy[4]);
      lineto(xx[j]), yy[j]);
   }
```

4. Program de animație prin funcțiile de manipulare a paginilor video.

```
#include<iostream.h>
#include<alloc.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
#define PIi  3.14
#define  N  3
#define LAT 100
#define LAT2   LAT/2
#define H 75
#define D  50
int dy=200, gd,gm,err,xx[5],yy[5],i,j;
void gen_n_pag_video();
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
void desen();
void main()
{
 gd=VGA; gm=VGALO;                    //VGALO=4 pagini video
 initgraph(&gd,&gm,"C:\\CPP\\bgi ");
 err=graphresult();
 if (err!=grOk)  {
        cout<<"Eroare grafica :"<<grapherrormsg(err)<<endl;
        cout<<"Apasa o tasta pentru oprire :"<< endl;
       getch();
      exit(1);
     }
setlinestyle(DASHED_LINE,0,1);setcolor(BLUE);
gen_n_pag_video();
for (j=0;j<=28;j++)
      for (i=0;i<=N-1;i++)   {
            setvisualpage(i);              //afişează pagina i
           delay(500);
            }
   closegraph();
}
void gen_n_pag_video()
{
 int    x[5]={10,LAT+10,LAT+10,10,LAT2+10},
y[5]={D,D,D,D,D+H},
z[5]={10,10,LAT+10,LAT+10,LAT2+10} , xc=LAT2+10,
zc=LAT2+10, i, j ;
 double s,c,u,r,alfa,x1,z1;
 r=sqrt(2)/4; alfa=PI/2;  u=alfa;
 for (j=0;j<=4;j++)  {
       xx[j]=(int)(x[j]+r*z[j]);
       yy[j]=dy-(int)(y[j]+r*z[j]);
      }
 setactivepage(0);          //activă este pagina 0
 desen();                   //generează pg. 0, adică desenează piramida
 setvisualpage(0);          //afişează pagina 0
 for (i=1;i<=N-1;i++)  {
        c=cos(u);   s=sin(u);
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
      for (j=0;j<=3;j++) {                    //roteşte cu alfa vârfurile bazei
        x1=xc+(x[j]-xc)*c-(z[j]-zc)*s;
        z1=zc+(x[j]-xc)*s+(z[j]-zc)*c;
       xx[j]=(int)(x1+r*z1);
       yy[j]=dy-(int)(y[j]+r*z1);
      }
 setactivepage(i);                            //activă este pagina i
 desen();                                     //generează pagina i
 setvisualpage(i);                            //afişează pagina i
 u=u+alfa;
 }
}
void desen()
{
 int j;
 moveto(xx[0], yy[0]);
 for (j=1;j<=3;j++) lineto(xx[j], yy[j]);
 lineto(xx[0], yy[0]);
 for (j=0;j<=3;j++) {
      moveto(xx[4],yy[4]);
      lineto(xx[j]), yy[j]);
 }
}
```

Observaţii.

1. Funcţiile **graphresult()** şi **grapherrormsg(int);** folosite în aceste 4 programe permit detectarea erorilor ce pot apărea în program. Codul erorii detectate este furnizat de graphresult şi stocat într-o variabilă temporară întreagă. Funcţia grapherrormsg întoarce un mesaj de eroare pentru codul erorii din această variabilă. Funcţia graphresult() întoarce un cod de eroare pentru ultima operaţie grafică raportând o eroare şi resetând nivelul de eroare la grOk.
2. Funcţia **farmalloc(unsigned long)** alocă memorie pentru heapul far(un spaţiu de memorie localizat într-o zonă distinctă a memoriei interne, diferită atât de zona codului programului cât şi de stiva pentru alocarea statică a datelor). Variabilele din această zonă nu pot fi referite direct printr-un nume de variabila, ci indirect printr-o variabilă statică ce conţine adresa heap-ului far de la

https://biblioteca-digitala.ro / https://unibuc.ro

care începe memorarea variabilelor dinamice (din această zonă), variabilă numită **variabila pointer far**. Eliberarea spațiului de memorie alocat cu funcția farmalloc în zona heap se face cu funcția farfree.

Aceste funcții au prototipul în fișierul antet **alloc.h**. Funcția farmalloc întoarce un pointer la blocul de memorie alocat sau valoarea NULL dacă alocarea nu reușeste din lipsă de spațiu suficient pentru blocul ce trebuie alocat.

4.10 Fractali [6, 68, 83]

Se pot imagina figuri geometrice diferite de cele formate din linii,suprafețe sau corpuri cu 3 dimensiuni. Unele dintre ele aparțin unui spațiu cu dimensiuni fracționare între 1 și 2 și având o structură între linie și suprafață. Aici nici lungimea și nici aria nu se poate măsura deoarece marginile figurii nu sunt netede. Spre exemplu țărmul unei insule are lungimea variabilă în funcție de precizia cu care se ia în considerare fiecare fiord, apoi fiecare piatră și fiecare bob de nisip. La fiecare prag de semnificatie corespund lungimi cu totul diferite. Alte exemple pot fi forma norilor, a cetii, a conturului unei păduri a liniilor de coastă, a fulgerelor, a comportamentelor seismice, climaterice, etc. și în general forma oricărui peisaj din natură.

Această clasă de figuri cu dimensiuni fracționare definește fractalii, care descriu sugestiv dinamica evoluției unor sisteme, mișcarea browniană, evoluția unor populații într-un spațiu vital limitat etc. Fractalul a fost descoperit la începutul anilor '80 de matematicianul Michael Barnsley care observând cum fiecare frunză de ferigă se aseamană cu întreaga plantă, a modelat matematic acest fenomen. Dezvoltând această idee a reușit să "recreeze" alte imagini din natură așa cum ar fi: conturul munților, simulări grafice ale cutremurelor, etc., demonstrând că orice imagine din lume se poate reprezenta cu ajutorul unei categorii de fractali. Și aici este respectat principiul cauzalității conform căruia aceleași cauze produc în condiții similare aproximativ aceleași efecte. Între anumite limite curbele își păstrează regularitatea existând însă și intervale în care punctele apar cu totul imprevizibil.

Decupând dintr-o imagine o parte și mărind-o obținem porțiuni care reproduc întregul amintindu-ne că respectă în mare aceiași legitate. Fractalii au următoarele caracteristici:

https://biblioteca-digitala.ro / https://unibuc.ro

a) autosimilaritatea (fiecare parte conține în ea imaginea întregului);
b) iregularitatea (nu are frontiere netede) care conduce la imposibilitatea determinării exacte a lungimii sau suprafeței lor.
Descrierea formală a acestor imagini se face prin ecuații cu diferențe.
Mulțimea stărilor unui sistem dinamic liniar restricționată la un domeniu vizibil se poate obține printr-un program ca cel următor [83]:

```
# include <stdlib.h>
# include <stdio.h>
# include <dos.h>
# include <graphics.h>
# include <iostream.h>
double a[] = { 0.,0.2,-0.15,0.85);
double b[] =  {0,-0.25,0.3,0.05);
double c[] = {0,0.20,0.25,-0.05);
double d[] = {0.15,0.20,0.25,0.85);
double e[] = {0,0,0,0);
double f[] = {0,1.5,0.45,1.5);
double p[] = {320,2600,4900,32767);
int xscale = 25, yscale = 25, xoffset = 0, yoffset = -160;
void main ( )
{
int d=0,m=0;
void  desen(int);
initgraph ( &d,&m," c: \\cpp\\bgi");
desen(4);getchar( );
}
void desen ( int color )
{
int px,py,i,j,k;
double y=0,x=0,newx;
srand(5);
for ( i=1; i<=10000;i++)  {
                 j = rand( );
                 k = ( j<p[0]? 0: (( j<p[1])?1:( j<p[2]? 2:3));
                 color = k + 1;
                newx = a[k]*x + b[k]*y + c[k];
                y = c[k]*x + d[k]*y + f[k]; x = newx;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
            px = x*xscale + xoffset;

            py = y*yscale + yoffset;
            if (( px  >=-320) && ( px < 320)&& (py >= -240) && (py <
240)) putpixel ( px  + 320, py + 240, color);
            }
}
```

Fractalul obținut cu acest program este foarte asemănător unei frunze de ferigă.

Putem obține o dimensiune fracționară și rezolvând grafic ecuația complexă pătratică cu diferențe finite: $Z_{n+1} = Z_n^2 + C$ pornind de la starea inițială Z_0. Pe axe formula devine: $x_{n+1} = x_n^2 - y_n^2 + a$, $y_{n+1} = 2x_n y_n + b$, daca $C=a+bi$.

Mulțimea numerelor complexe pentru care pornind de la origine ($Z_0=0$), mulțimea punctelor Z_n este mărginită, excepție eventual un număr finit de puncte, se numește **mulțime Mandelbrot** (dupa numele parintelui geometriei fractalilor Benoit Mandelbrot).

Dacă îl menținem pe C constant și pornim din stări inițiale diferite vom obține o mulțime de puncte numită **mulțime Julia**.

Programul următor [68] reprezintă o mulțime Iulia pentru C=1.0+0.9i parcurgând două etape; prima pentru a găsi valorile minime și maxime a primelor 1000 puncte ce se vor reprezenta și a doua pentru a reprezenta mai multe puncte cu scalare corespunzătoare și cliping (decupare) automat.

```
# include <stdio.h>
# include <stdlib.h>
# include <dos.h>
# include <conio.h>
# include <graphics.h>
# include <iostream.h>
void main (void)
{
 void desen (int); desen(4);
}
void desen (int color)
{
float x = 0,wr = 0. wl = 0, y = 0,wt = 0, wb = 0, newx, newy, a =1., b =
0.9;
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
int i,xe,ye,vl = 0, vr = 400, vb = 0, vt = 400;
int d =0, m = 0;
clrscr ( )
cout <<" Dați a și b"<<endl; cin >> a >>b;
for ( i = 0 ; i < 1000; i++ ) {
            newx = x*x – y*y + a; newy = 2*x*y + b;
            if ( newx < wl ) wl = newx;
                    else  if ( newx > wr )  wr = newx;
            if ( newy < wb ) wb = newy;
                    else  if ( newy > wt)  wt = newy;
            x = newx; y = newy;
                  }
x = y = 0; clrscr ( ); fflush ( stdin);
cout <<"wl =" << wl << "wr = " << wr << "wt = " << wt << "wb = " <<
wb << endl ;
getchar( )
initgraph ( &d, &m, "c : \\cpp\\bgi");
for ( i = 0; i < 1000; i++ ) {
             newx = x*x – y*y + a; newy = 2*x*y + b;
             x = newx; y = newy;
            xe = ( vr – vl)/( wr – wl )*x + ( vl*wr – vr*wl)/(wr – wl);
            ye = ( vt – vb)/( wt – wb)*y + (vb*wt – vt*wb)/(wt – wb);
            putpixel (xe,ye, i%5);
            }
getch ( ); closegraph ( );
}
```

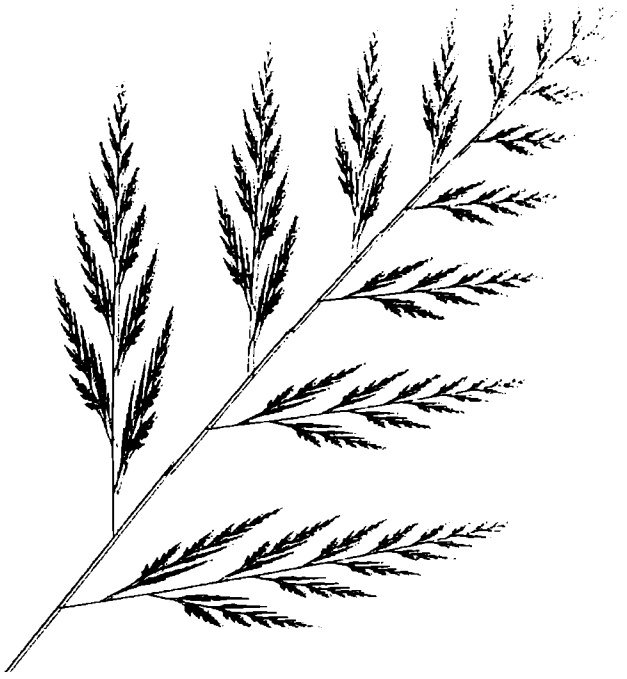
Reducerea unei imagini la sistemul de ecuații care o descriu deschide o nouă pistă în domeniul comprimării, stocării și transmiterii la distanță a unei imagini grafice prin sisteme de funcții iterate. Prin transformarea unei imagini în fractal s-a obținut o rată incredibilă de compresie de peste 10.000 la 1, punându-se bazele tehnologiilor bazate pe transmitere în timp real a imaginilor video în miscare prin linii telefonice normale.
Aproximarea rădăcinilor unei ecuații algebrice sau transcendente f(x) = 0 prin **metoda tangentei** (dată de Newton), aplicabilă când se dispune de un interval ce conține soluția și pe care f' și f" au semn constant, se face prin șirul : $x_{n+1} = x_n - f(x_n)/f'(x_n)$.

https://biblioteca-digitala.ro / https://unibuc.ro

Acest lucru corespunde înlocuirii unor porțiuni mici de curbă cu segmente din tangenta dusă în unul din punctele diviziunii, procesul de convergență fiind vizualizat grafic prin fractali. Deplasarea de la un punct grafic la altul se poate descrie şi prin reguli de producție, caz în care imaginea obținută se numeşte **graftal**.

Remarcăm că geometria fractală, care de multe ori încearcă să imite neregularitățile universului, evidențiază incapacitatea metodelor clasice de modelare în fața diversității haotice a naturii. Omul s-a îndepărtat treptat de formele neregulate construindu-şi o lume în care predomină regularitatea. Este interesant de observat că tocmai prin cele mai regulate şi riguros controlate creații ale sale (calculatoarele electronice), omul şi-a întors din nou fața spre imprevizibil. Acestea i-au permis să imite într-un mediu virtual, realitatea de care s-a depărtat, reuşind să se apropie din nou de natură.

In concluzie trebuie realizat că oricât de fidel am copia un segment din realitatea înconjurătoare nimic nu se compară cu originalul.



https://biblioteca-digitala.ro / https://unibuc.ro

https://biblioteca-digitala.ro / https://unibuc.ro

Partea a III-a

Programare WEB



- HTML
- JavaScript

https://biblioteca-digitala.ro / https://unibuc.ro

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL I Limbajul HTML

1.1 Introducere (Internet, HTML, Java, URL)[28,78,77]

Internet-ul, sau rețeaua mondială de calculatotore, reprezintă un puternic instrument pentru facilitarea afacerilor. Tradițional, Internet-ul a avut patru aplicații principale: poşta electronică, ştiri, conectarea la distanță şi transferul de fişiere. Odată cu amploarea luată de această mare rețea de calculatoare s-au diversificat şi numărul acestor aplicații este în continuă creştere.

Web-ul (World Wide Web sau WWW – pânza de păianjen mondială), instrument care a revoluționat accesul la Internet, este o rețea de calculatoare bazată pe tehnologiile Internet şi care permite utilizatorului unui calculator să acceseze informații aflate pe un alt calculator din rețea. Fiind un sistem client/server, este o colecție imensă de documente (numite pagini) răspândite în toată lumea. Fiecare pagină conține link-uri (legături) la alte pagini. Documentele Web sunt realizate cu ajutorul limbajului HTML (HyperText Markup Language) si includ text, grafică şi indicatori către alte pagini de Web. Utilizând HTML se pot crea documente statice. Cu ajutorul limbajului JavaScript se adaugă dinamism paginilor Web.

JavaScript este un limbaj de programare simplu, de tip script, pentru definirea comportamentului elementelor dintr-o pagină Web. JavaScript poate specifica, în doar câteva rânduri, răspunsurile la acțiuni sau evenimente cum ar fi deschiderea unei pagini, deplasarea mouse-ului într-un anumit punct sau ştergerea unui anumit câmp dintr-un formular.

Internet este rețeaua publică ajunsă la scară planetară. Este alcătuit dintr-un conglomerat de rețele de diverse complexități, de la cele multinaționale la cele regionale sau zonale, toate vorbind o aceeaşi limbă, statuată de familia de protocoale TCP/IP.

Internet a crescut extrem de rapid. Dacă in 1981 erau doar 213 calculatoare gazdă în cadrul rețelei, prin 1990, avea cca 320 000 calculatoare gazdă, ajungând în 1993 la cca 2,1 milioane. Acum, numărul calculatoarelor gazdă a explodat practic, în plină eră a sistemului Web.

Principiul director care a făcut reteaua publică Internet atât de populară este cel al adoptării unor standarde deschise.

https://biblioteca-digitala.ro / https://unibuc.ro

Pe lânga stiva TCP/IP obligatorie, în memoria staţiei de lucru, vor mai fi programe sau " servicii TCP/IP" ca de exemplu: e-mail, ftp, telnet, archie, gopher, WAIS, finger, WWW, etc.

Naşterea, în martie 1989 şi creşterea sistemului *World Wide Web* a schimbat practic felul de comunicare al majorităţi utilizatorilor planetei. Tim Barners-Lee, s-a gândit să uşureze modul de comunicare al celor neiniţiaţi, si in 1992 a proiectat şi implementat un sistem care să permită legături, asocieri multiple între documente. Sistemul constă dintr-un program interactiv numit *browser* sau *provider,* care uşurează "răsfoirea" unui atare păienjeniş de documente, urmărind legăturile dintre acestea *(links).* Aşa a prins viaţa proiectul WWW. Prin ianuarie 1993, deja existau vreo 50 de servere Web sau Web sites. Toate foloseau acelaşi protocol pentru hipertexte, cunoscut sub acronimul HTTP (*HyperText Transfer Protocol*). HTTP este un protocol adica un set de reguli "ştiut" de protagoniştii celui care îl întrebuinţează, iar protagoniştii tipici sunt două programe care dialoghează, clientul pe de o parte şi server-ul Web pe de alta. Viaţa unei tranzacţii sub protocol *http* trece prin 4 faze: conectarea, cererea, răspunsul şi închiderea. În cursul acestor faze clientul caută să contacteze server-ul Web:

- Dacă tentativa eşuează, apare o casetă de dialog care anunta: "conexiunea nu a putut fi stabilită"
- Dacă tentativa reuşeşte, clientul emite o cerere către server-ul Web (alias server-ul *http*), care contine obiectul specific pe care îl cerem, ca şi protocolul pe baza căruia trebuie să raspundă server-ul respectiv.
    - Dacă server-ul *http* prelucreaza cererea, el va pregăti răspunsul,
    - Daca nu, apare mesaj de eroare.

Vom numi "obiecte" aceste piese de informaţie care împreună alcătuiesc răspunsul la cerere. Pe măsura sosirii unui obiect acesta este afişat.

HTML este un limbaj simplu de descriere a documentelor care se referă la alte documente, derivat din SGML (standard generalizat de limbaj pentru hipertexte). Limbajul va avea ca "output" un fişier de tip text, ce poate fi prelucrat cu cel mai simplu editor. Comenzile HTML se insereaza în jurul blocurilor de text dintr-un document WWW, pentru a descrie tipul textului: diferite niveluri de antet, paragraf simplu, header sau footer, etc. Există şi alte comenzi ce permit specificarea legăturilor către alte documente sau către alt text din acelaşi document şi de asemenea comenzi care permit importul altor medii (sunet, imagine, animaţie).

https://biblioteca-digitala.ro / https://unibuc.ro

Clientul browser primeşte documentul de la server-ul WWW, interpretează comenzile HTML, formatează fiecare structura din document în modul cel mai favorabil posibil pentru monitorul respectiv.

Java [77] este un limbaj orientat spre obiecte, destinat mediului de reţea, robust, care oferă un acces securizat la informaţie, neutru din punct de vedere architectural, portabil, prevăzut şi cu mecanismul de *multi threading*; în fine, este şi un limbaj de scop general. Cu el se obtin aplicaţii, denumite în engleză *applets*. Orice *browser* existent azi ştie să interpreteze programe Java, în sensul că printre componentele sale suplimentare (*plug-ins*), conţine şi acest interpretor al codului intermediar sub care "circula" *applet*-urile. Java conferă un cu totul alt nivel de interactiviate Web-ului. Dar nu numai Java este în stare de acest lucru, ci şi JavaScript. Java economiseşte munca de scriere a programului, întocmai ca munca unui stilist care creează modele în industria textilă; şi în programare găsim *obiecte* şi *clase* gata făcute de program, pe care le includem în programele făcute cu Java. În acest mod putem crea lucruri de efect, chiar fară să avem cunostinte de programare. Prin Java putem interacţiona cu mediul şi astfel impresiona pe cei ce vizitează pagina de casa cu o serie de efecte de animaţie sau sonore. Java este considerat un limbaj de nivel înalt de programare, precum C++. Singura diferenţă faţă de acesta este că Java este primul limbaj de programare dezvoltat pentru mediul de reţea. Java posedă o bogată colecţie de rutine pentru protocoalele TCP/IP cum ar fi HTTP sau FTP. *Applet*-ul Java poate înlesni accesul oriunde în Internet, prin sistemul URL (Universe Resource Locator). Compilatorul Java va genera un aşa-zis cod neutru, de fapt un fişier obiect, care nu conţine instructiuni maşină. Din această cauză se spune că este *arhitecture-neutral* , adică independent de arhitectură. Cel care face translaţia din acest cod intermediar în codul maşină propriu platformei este aşa numitul *just-in-time compiler*, un program de interpretare a codului intermediar. Pentru fiecare dintre platforme, acesta va genera instrucţiunile maşină specifice, chiar la momentul încarcarii aplicaţiei compilate cu Java, adică *applet*-ului.

Pentru a putea lucra cu HTML, Java sau JavaScript, trebuie să ştim ce este un URL.

URL sunt iniţialele cuvintelor *Uniform Resource Locator*- mijlocul de localizare a unei resurse tangibile în Internet, adică a unui calculator gazdă din mediul Web. El poate fi considerat drept un fel de căsuţă poştală sau o adresă de post restant.

https://biblioteca-digitala.ro / https://unibuc.ro

Fiecare mesaj URL permite browser-ului să intre în legatură cu un server Web, spre a avea acces la tot felul de fişiere. Pe lângă faptul că dă posibilitatea identificarii unei pagini de casă anume cautată, fiecare marcaj URL adresează o resursă şi, în plus, indică şi metoda prin care se poate accesa resursa respectivă.
Forma generală a marcajului URL este:
metoda de acces://nume-server/specificaţia-de-cale/nume-fisier.html
În mediul Web protocolul standard este:
- **http**, cu varianta sa care asigură transferul securizat, întalnită sub prescurtarea **https**, **s** de la **secure shell**.

Sunt şi alte protocoale, spre exemplu:
- **ftp**, protocolul de transfer al fisierelor (*file transport protocol*);
- **news**, pentru acces in *Usenet newsgroup* (adică la unul din cluburile de discuţii axate pe cele mai diferite domenii);
- **gopher**, acces la un serviciu de tip cautare pe bază de indecşi şi meniuri cu opţiuni selectabile prin cifre şi litere;
- **telnet**, accesul la un calculator sub forma sesiunii clasice de lucru sub UNIX, argumentele cozii telnet referindu-se la denumirea sesiunii sub care se lucrează, la parola de acces (password) şi la ce calculator gazdă (host), sub forma: *user:password@host*;

Prin FTP sunt accesibile milioane de fişiere din Internet. Pe majoritatea maşinilor gazdă, nu este nevoie nici măcar de un cont personal pentru a descărca fişierele căutate. Asta pentru că o practică standard în Internet, numită *FTP anonim*, permite oricărui utilizator să se conecteze la o gazdă FTP, utilizând numele *anonymous*. Dacă se cere o parolă, se va tasta adresa **e-mail**. Printre alte utilizari, FTP anonim este folosit pentru a obţine programe noi pentru calculator.
Un URL de tip FTP are forma următoare: **ftp://gazdă/cale-director**

1.2. Marcaje [28, 30, 32]

Limbajul de Marcare HiperText (*HyperText Markup Language - HTML*) este un set de coduri speciale care se inserează într-un text, pentru a adăuga informaţii despre formatare şi despre legături. HTML se bazează pe *Limbajul Generalizat Standard de Marcare* (*Standard Generalized Markup Language - SGML*). Prin convenţie, toate informaţiile HTML încep cu o paranteză unghiulară deschisă (<) şi se termină cu o paranteză unghiulară închisă (>) - de exemplu, <HTML> -.Acest *control*

https://biblioteca-digitala.ro / https://unibuc.ro

sau *marcaj HTML*, cum mai este numit, comunică unui interpretor HTML (program de navigare) că documentul este scris şi formatat în limbajul HTML standard.

Deşi multe controale HTML sunt unităţi de sine stătătoare, unele din ele sunt *perechi,* având controale de început şi sfârşit. Controlul de început se numeşte *control de deschidere*, iar cel de sfârşit se numeşte *control de închidere.*

Marcajul <HTML> este de tip pereche, deci trebuie adăugat un marcaj de închidere la sfârşitul documentului, care este la fel ca marcajul de deschidere, dar este precedat de /: </HTML>. În aceeaşi manieră, frazele scrise cursiv (cu caractere înclinate sau italice) vor fi trecute între marcajele <I> şi </I>. Orice element situat între marcajul de deschidere şi cel de închidere capătă atributul acelui marcaj.

Cele mai folosite marcaje specifică faptul că se vrea trecerea la un nou *paragraf* sau la o *nouă linie*. Există diverse variante ale acestor controale, dar se pot crea documente Web lizibile şi utile utilizând doar două marcaje: <P> pentru trecerea la un nou paragraf şi
 pentru linie nouă (break). Ca orice marcaj,
 poate apărea oriunde în text, inclusiv la sfârşitul liniei pe care doream să o întrerupem. Controalele HTML nu ţin seama de litere mari sau mici, ceea ce înseamnă că
 şi
 sau chiar
 sunt exact la fel.

Pentru a remedia indentarea la dreapta se poate utiliza *marcajul de informaţie preformatată*: <PRE>. Acesta este şi el de tip pereche, deci acţionează asupra oricarui număr de linii şi trebuie încheiat cu </PRE>.

Orice document HTML este împărţit în două secţiuni: informaţia care ar fi tipărită în antet şi corpul propriu-zis al mesajului. Similar, fişierele HTML sunt de obicei împărţite în două secţiuni: *antetul* (sau *headerul*) care conţine informaţia introductivă de formatare a paginii şi *corpul.* Pentru a delimita fiecare secţiune se utilizează marcajele pereche <HEAD> </HEAD> şi respectiv <BODY> </BODY>.

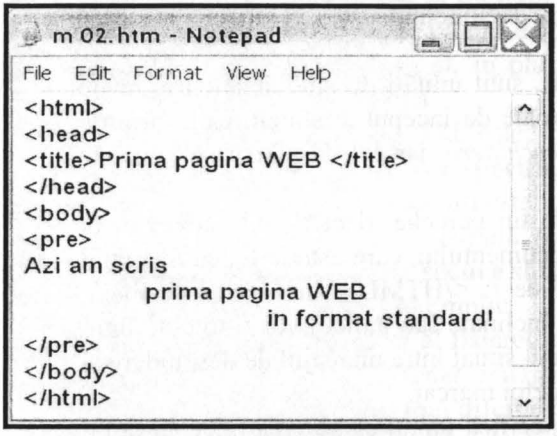
Pentru ca pagina de Web să arate mai elevat ii putem da un titlu adecvat, folosind marcajul <TITLE>. Titlul apare de obicei pe bara de sus a ferestrei afişate de calculator. Marcajul <TITLE> permite să definim exact titlul dorit al documentului. Este un marcaj pereche şi apare în cadrul blocului de informaţii <HEAD> </HEAD>, ca mai jos:

<HEAD>

<TITLE></TITLE>

</HEAD>

https://biblioteca-digitala.ro / https://unibuc.ro



Textul din <TITLE> este utilizat şi ca informaţie de legătură atunci când utilizatorul salvează un document Web într-o *listă de pagini* (*bookmark*) sau *listă favorită*. Astfel, un <TITLE> semnificativ pentru fiecare pagină creată poate fi de un real ajutor.



Aşa cum titlul este plasat în antetul fiecărui document Web, unele informaţii sunt plasate de obicei la sfârşit, de exemplu cele legate de datele de contact ale autorului paginii. Marcajul utilizat pentru asemenea informaţii de contact este de obicei <ADRESS> . Acesta este un marcaj pereche **<ADRESS> informaţii_de_adresă</ADRESS>.**

Informaţiile de formatare discutate până acum permit crearea de documente atractive. Dacă pagina trebuie organizată în secţiuni sau chiar subsecţiuni,fiecare nivel de subtitlu are un control de deschidere şi unul de închidere. Marcajul pentru nivelul cel mai înalt este <H1> iar pentru nivelul cel mai scăzut este <H6>. Pentru a specifica un subtitlu pe nivelul cel mai de sus, se utilizează **<H1>Primul nivel de subtitlu</H1>.**

Un control HTML util pentru organizarea vizuală a documentului este *marcajul de riglă orizontală*: <HR>. Plasat oriunde într-un document Web, acesta produce o linie subţire orizontală afişată pe întrega lăţime a paginii care poate fi folosită de mai multe ori într-un document Web. Utilizat judicios, marcajul <HR> este extrem de util pentru crearea paginilor intersante. O listă a marcajelor de bază HTML se găseşte în Anexă.

https://biblioteca-digitala.ro / https://unibuc.ro

1.3 Stiluri de text [28, 32, 88, 77]

Una din cele mai importante caracteristici in tehnoredactare este utilizarea diverselor fonturi şi stiluri.
Formatarea îngroşată şi înclinată necesită marcaje HTML pereche:
- Marcajul de formatare italică (înclinată) este <I>, cu marcajul pereche </I>;
- Marcajul de formatare îngroşată (bold) este <B>, cu marcajul pereche </B>.

Observaţie: imediat după paranteza unghiulară deschisă a unui marcaj de formatare HTML se scrie codul de formatare; nu sunt permise spaţii.

Alte opţiuni de formatare a textului în documentele Web:
- Marcajul de subliniere este <U>, cu marcajul pereche </U>;
- Marcajul monospatiu este <TT>, cu marcajul pereche </TT>;
- Indicii superiori (superscript) sunt definiţi prin ^{şi}, iar cei inferiori (subscript) prin _{şi};
- Textul poate fi "tăiat" utilizând <STRIKE>, cu marcajul pereche </STRIKE>.

Denumirea de monospaţiu provine de la faptul că fiecare literă dintr-un font de acest tip ocupă exact aceeaşi lăţime, chiar dacă litera respectivă este destul de îngustă. Textul scris cu un font monospaţiu seamănă cu un text scris la maşină.

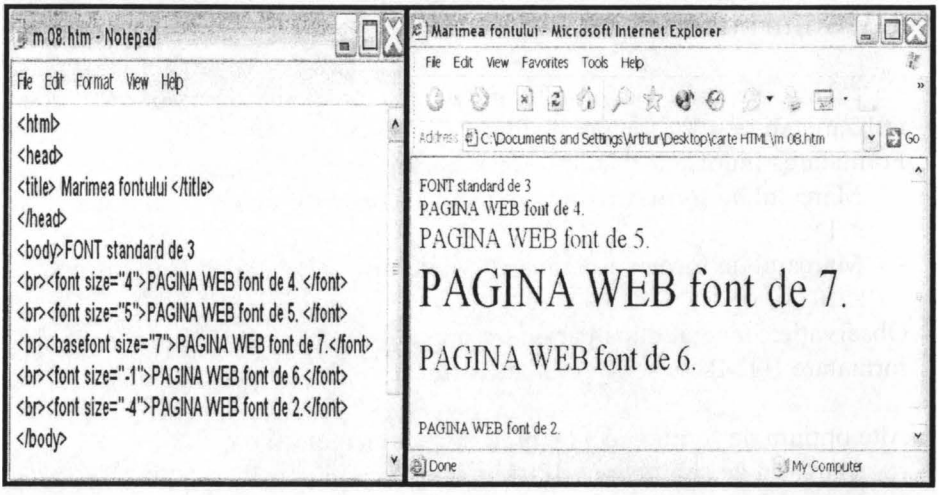
O extensie recentă a limbajului HTML reprezintă una din cele mai atractive proprietăţi: posibiliatea de a modifica dimensiunea, culoarea şi corpul de literă al fontului utilizat. Dimensiunile fonturilor HTML variază de la 1 la 7, 1 fiind cel mai mic, iar 7 cel mai mare. Toate caracteristicile fontului se fac prin diverse atribute ale marcajului <FONT>. Atributele marcajelor HTML se specifică, de obicei, prin perechi de forma nume=valoare. Pentru a modifica dimensiunea unei porţiuni de text se va folosi formatarea:

<FONT SIZE=7>HTML</FONT>

Aici cuvântul HTML va fi afişat cu cea mai mare dimensiune posibilă a programului de navigare.

Alte exemple pot fi văzute în continuare:

https://biblioteca-digitala.ro / https://unibuc.ro



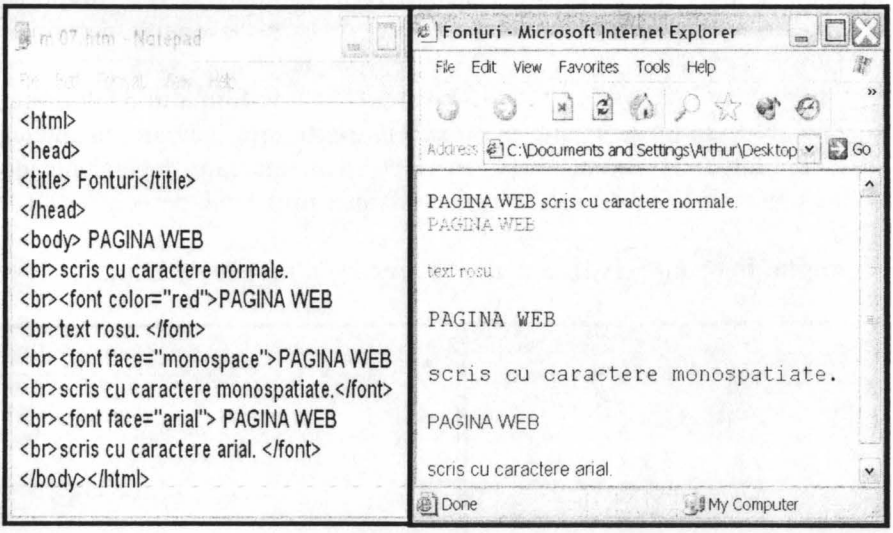
Marcajul de închidere </FONT> nu trebuie să includă atributele marcajului de deschidere (nu a fost nevoie de </FONT SIZE=7> pentru a încheia textul scris cu font mare). Fontul implicit al majorității programelor de navigare este SIZE=3.

În mod similar, se poate specifica o culoare pentru o porțiune de text, prin folosirea atributului **color** al maarcajului <FONT>. Secvența

<FONT COLOR=GREEN>HTML</FONT>

va afișa textul respectiv ("HTML") cu culoare specificată. O proprietate utilă a limbajului de marcare pentru marcaje cu atributeeste posibilitatea specificării mai multor atribute în același marcaj. Dacă se dorește un marcaj mic și albastru, de exemplu, se poate utiliza secvența <FONT SIZE=1 COLOR=BLUE> HTML</FONT>. Nu contează ordinea în care se specifică perechile nume=valoare, deci <FONT COLOR=BLUE SIZE=1 > este identic, din punctul de vedere al programului de navigare, cu <FONT SIZE=1 COLOR= BLUE >.
Un exemplu concludent este următorul:

https://biblioteca-digitala.ro / https://unibuc.ro



Limbajul HTML permite, pe lângă directivele de stil specifice şi aşa numitele stiluri logice care permit definirea unor evidenţieri ale textului. Cele mai cunoscute stiluri logice sunt <EM> </EM> pentru evidenţiere şi <STRONG> </STRONG> pentru evidenţiere puternică.

În standardul HTML sunt specificate multe alte marcaje logice, mai rar folosite , printre care:

<CITE>	</CITE>	Citare bibliografică
<CODE>	</CODE>	Listing de program
<DFN>	</DFN>	Definiţie de cuvânt
<KBD>	</KBD>	Text de la tastatură
<SAMP>	</SAMP>	Exemplu de date introduse de utilizator
<VAR>	</VAR>	Program sau variabilă

Marcajele HTML corespunzătoare acestei secţiuni se află în Anexă.

1.4 Liste si caractere speciale [28, 30, 32, 77]

Unul din cele mai fireşti elemente din documentele HTML cu mai multe pagini este un set de definiţii, referinţe sau indexuri.

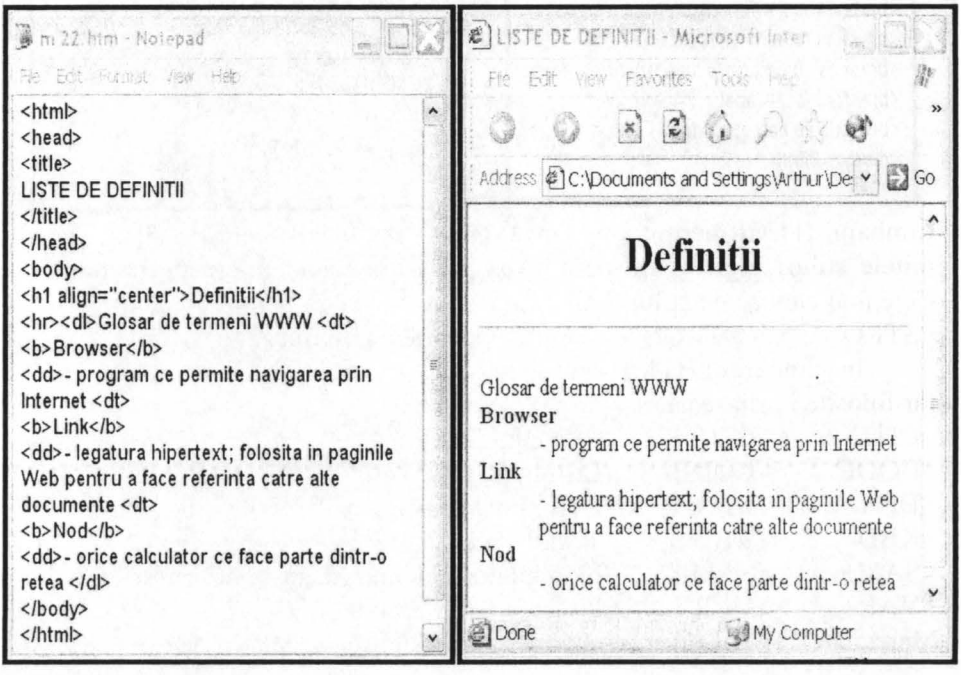
Glosarele sunt exemple clasice în acest sens; cuvintele sunt listate în ordine alfabetică, urmate de definiţii ale termenilor respectivi. În HTML, întreaga secţiune a unui glosar va fi gestionată printr-o *listă de definiţii*, care este inclusă într-o pereche de *marcaje de listă de definiţii*:

https://biblioteca-digitala.ro / https://unibuc.ro

<DL> şi </DL>>. În cadrul acestei perechi de marcaje, o definiţie se compune din două părţi: termenul definit: <DT> şi descrierea definiţiei: <DD>

Conceptul de bază al unei liste este relevat de formatul de definire: o pereche de marcaje în interiorul căreia există alte marcaje cu înţeles special. Marcaje precum <DT> şi <DD> sunt marcaje dependente de context: nu au sens decât dacă apar în interiorul unei perechi <DL> </DL>.

Exemplu. In codul HTML următor se prezintă o listă de definiţii.



1.4.1 Liste neordonate (marcate)

Listele de tip definiţie sunt utile, dar tipul de listă mult mai întâlnit în World Wide Web este lista marcată (numită şi listă neordonată).

Listele neordonate încep cu <UL> şi se încheie cu </UL>, iar fiecare element al listei este definit prin controlul <LI>. Formatul este similar cu cel al listei de tip definiţie.

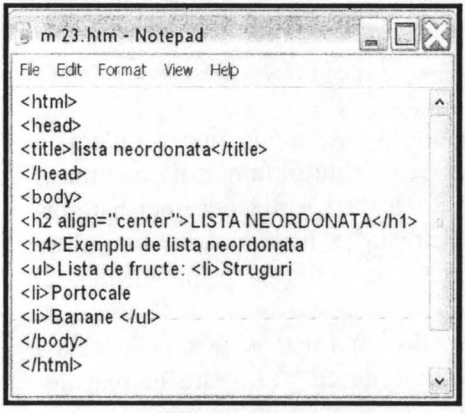
Combinarea celor două tipuri de liste este şi mai utilă. Lista de tip definiţie arată foarte profesional datorită caracterelor îngroşate şi inden-

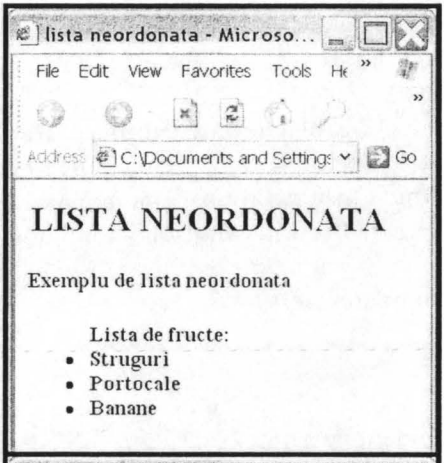
https://biblioteca-digitala.ro / https://unibuc.ro

tării, iar marcajul din dreptul fiecărui element dintr-o listă neordonată este şi el de mare efect. Soluția este includerea unei liste în cealaltă.

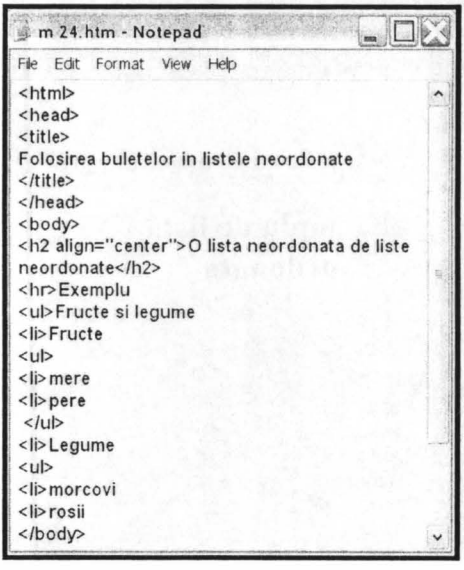
Exemple.

1. Listă neordonată (de fructe) cu elementele pe un singur nivel, se obține ca în codul HTML următor:





2.Liste neordonate (de fructe şi legume) imbricate pe mai multe ni vele, se obțin ca în codul HTML următor:





https://biblioteca-digitala.ro / https://unibuc.ro

Cu atributul *type* al tag-urilor **<ul>** şi **<li>** se poate alege caracterul afişat în faţa fiecărui element al listei. Acesta poate lua valorile:

- "disc" (valoarea prestabilită);
- "circle" (cerc)
- "square" (pătrat)

Observăm în exemplu că listele neordonate pot fi imbricate pe mai multe niveluri.

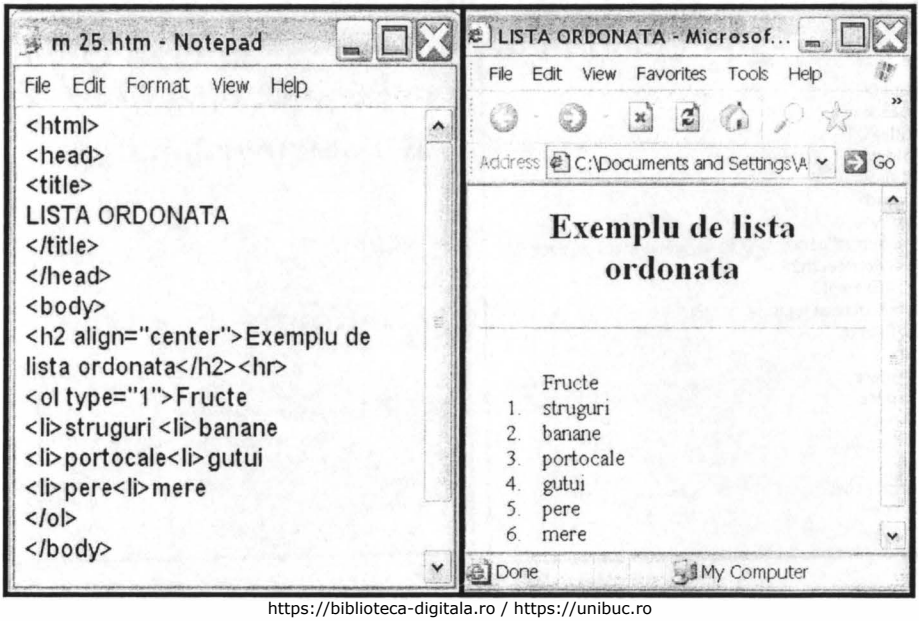
1.4.2 Liste ordonate (numerotate) [28, 32, 77]

Ddacă se doreşte înlocuirea marcajului din dreptul fiecărui element al unei liste neordonate cu număr avem doua soluţii: numerotarea tuturor elementelor sau, mult mai simplu şi mai elegant, utilizarea unei liste ordonate. Marcajul pentru ea este <OL>. Lista se încheie cu marcajul de închidere </OL>. Fiecare element al listei se defineşte printr-un marcaj <LI>.

Se observă că specificarea elementelor unei liste se face prin acelaşi marcaj HTML. Acum se observă mai bine de ce <LI> este marcaj dependent de context. Dacă nu se specifică tipul listei, programul de navigare nu va ştii ce listă să folosească.

Exemple.

1. Listă ordonată de fructe folosind cifrele arabe, se obţine următorul cod HTML:



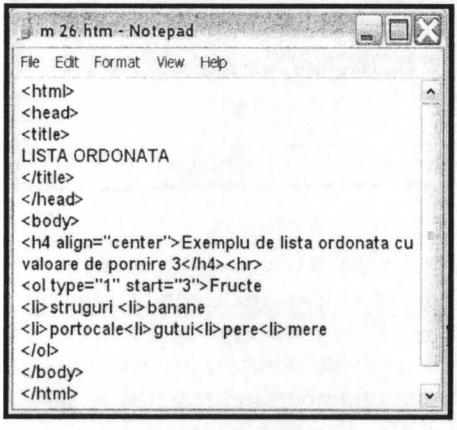
https://biblioteca-digitala.ro / https://unibuc.ro

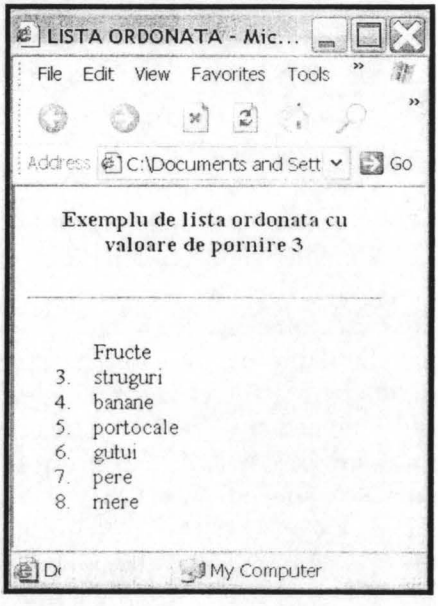
Cu atributul *type* al tag-urilor **<ol>** şi **<li>** se poate alege tipul de caractere afişat în faţa fiecărui element al listei. Acesta poate lua valorile:

- "1" - ordonare de tipul 1, 2, 3 etc. (cifre arabe - opţiune prestabilită);
- "A" - ordonare de tipul A , B , C , D etc. (litere mari);
- "a" - ordonare de tipul a , b , c , d etc. (litere mici);
- "I" - ordonare de tipul I , II , III , IV etc. (cifre romane mari);
- "i" - ordonare de tipul i , ii , iii , iv etc. (cifre romane mici);

Observăm că ordonarea s-a făcut începând cu cifra 1, **valoarea iniţială a secvenţei de ordonare** este dată de Tag-ul **<ol>** atributul *start*. Putem schimba această valoare dând atributului cifra dorită, să zicem 3 pentru exemplul anterior.

2. Listă ordonată cu cifre arabe, începând de la valoarea 3, se obţinre următorul cod HTML:

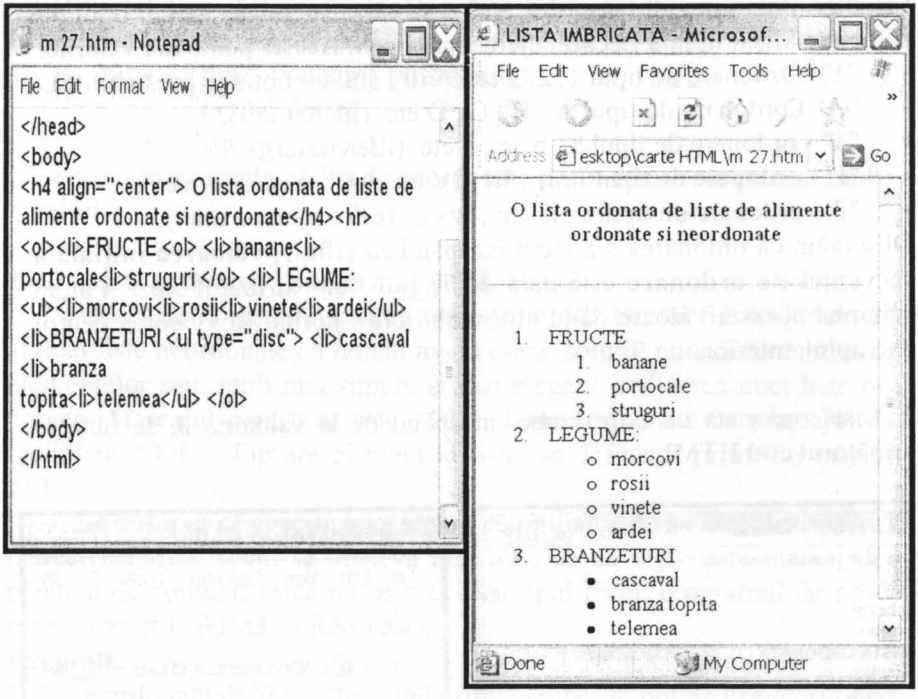




Listele ordonate pot fi imbricate între ele sau pot fi imbricate cu liste neordonate.

3. Liste ordonate imbricate cu liste ordonate şi/sau neordonate se obţin ca în codul HTML următor:

https://biblioteca-digitala.ro / https://unibuc.ro



1.4.3 Caractere speciale [32, 88]

Limbile străine conțin diferite caractere speciale, numite diacritice. Acestea pot fi incluse în codul HTML utilizând marcaje speciale, numite entități sau referințe de entități.

Entitățile pentru caractere speciale nu sunt incluse între paranteze unghiulare (< >), ci încep întotdeauna cu un ampersand (**&**) și se termină cu punct și virgulă (;). Cele mai multe dintre ele sunt mnemonice, după cum se vede în lista de mai jos:

Caractere speciale în HTML

Caracter	**Cod HTML**	**Semnificație**
&	&	ampersand
<	<	mai mic
>	>	mai mare
á	á	*a* mic cu accent ascuțit
à	à	*a* mic cu accent grav
â	â	*a* mic cu accent circumflex
ä	ä	*a* mic cu umlaut

https://biblioteca-digitala.ro / https://unibuc.ro

å	å	*a* mic cu inel
ç	ç	*a* mic cu sedilă
ñ	ñ	*n* mic cu tildă
ø	ø	*o* mic barat
β	ß	*s-z* ligaturat

Pentru a crea o majusculă a unuia din caracterele de mai sus, se scrie prima literă a marcajului de formatare ca literă mare; de exemplu, &Oslash va produce litera Ø, ca în cuvântul CØPENHAGEN (care trebuie tastat C&Oslash ; PENHAGEN).

Pentru a produce o vocală diferită de a cu semn diacritic, se modifică prima literă a marcajului respectiv; de exemplu, cuvântul MALMÖ poate fi specificat corect într-un document HTML prin MALM&Ouml ; .

Aceste caractere nu pot fi afişate de toate programele de navigare şi de aceea este bine să fie verificate înainte de a fi folosite în redactarea paginii Web.

Există unele probleme legate de caracterele internaţionale permise în codul HTML de bază, aşa cum ar fi absenţa unor elemente. In versiunea 4.0 a limbajului HTML, dacă se doreşte să se scrie în spaniolă, există semnul de întrebare invers (¿), care poate fi inserat în text cu **¿** . Dacă specificăm valute, putem codifica lira sterlină (£) prin **&pound** şi, centul (¢) prin **&cent**. De asemenea simbolul de copyright (©) se codifică prin **&copy** şi cel de marcă înregistrată (®) prin **&reg.** Marcajele HTML pentru liste se găsesc în Anexă.

1.5 Spaţii de afişare şi comentarii [28, 30, 88]

O entitate de tip caracter special, des utilizată în proiectarea paginilor Web şi care nu reprezintă nici un caracter şi nu se afişează pe ecran, este spaţiul de afişare(non-breacking space). Specificat prin , acesta permite forţarea spaţiilor multiple între elemente şi asigură faptul că elementele de pe ambele părţi ale acestor spaţii sunt întotdeauna adiacente, indiferent de modul în care fereastra este dimensionată.
Dacă spre exemplu se doreşte ca un cuvânt să fie despărţit printr-un număr de spaţii pe ambele părţi prima soluţie ar fi:

Cuvinte anterioare cuvânt important cuvinte ulterioare.

Acestă soluţie nu funcţionează deoarece spaţiile suplimentare sunt ignorate de către programul de navigare. Soluţia corectă este:

Cuvinte anterioare   cuvânt important

https://biblioteca-digitala.ro / https://unibuc.ro

**   cuvinte ulterioare.**

HTML permite un format specific pentru comentarii în cadrul documentelor. Orice text încadrat de elementele <!- - şi - -> este considerat un comentariu şi este ignorat de programele de navigare Web.

1.6 Referinţe şi legături [28, 32, 78]

Toate marcajele trebuie să înceapă cu o paranteză unghiulară deschisă, urmată imediat de *elementul marcajului*; nu sunt premise spaţii. În interiorul marcajului se pot, însă, specifica alte atribute de forma *atribut=valoare*. Dacă dorim să includem mai multe cuvinte în zona de valoare, trebuie să ne asigurăm că acea porţiune este inclusă între ghilimele, ca de exemplu ALT=" alternative information" .

Nu pot exista spaţii între < şi numele marcajului, dar, în interiorul acestuia, elementele sunt separate prin spaţii.

Marcajul HTML principal pentru referinţe externe este <A>, marcajul *ancoră*, iar perechea sa este </A>. Acesta trebuie să conţină atribute. Fără atribute, marcajul ancoră nu are sens şi nu afectează formatarea informaţiei. Următorul exemplu va avea ca efect afişarea fără formatare a textului:

Se poate vizita <A>Universitatea Bucureşti</A> on line!

Pentru ca această legătură să devină activă, adică să comande programului de navigare să execute ceva, trebuie specificat un atribut de *referinţă hipertext*: HREF="valoare".

Valoare poate fi vidă dacă nu se cunoaşte valoarea respectivă, dar trebuie specificat atributul respectiv pentru ca legătura să apară activă în programul de navigare.

Pentru a transforma textul în legătură Web, el va fi rescris în forma următoare:

Se poate vizita <A HREF =" ">Universitatea Bucureşti</A> on line!

Linia precedentă de cod HTML va fi afişată de un program de explorare Web cu porţiunea dintre marcajele de ancoră colorată în albastru şi subliniată sau evidenţiată într-o altă formă. Informaţia care ar trebui inclusă între ghilimele este URL-ul paginii Web pe care dorim să o referim. De exemplu, URL-ul Universităţii Bucureşti este http://www.fmi.unibuc.ro..

Exemplul următor conţine o legătură hipertext activă, corect specificată, către pagina Universitatii Bucuresti:

https://biblioteca-digitala.ro / https://unibuc.ro

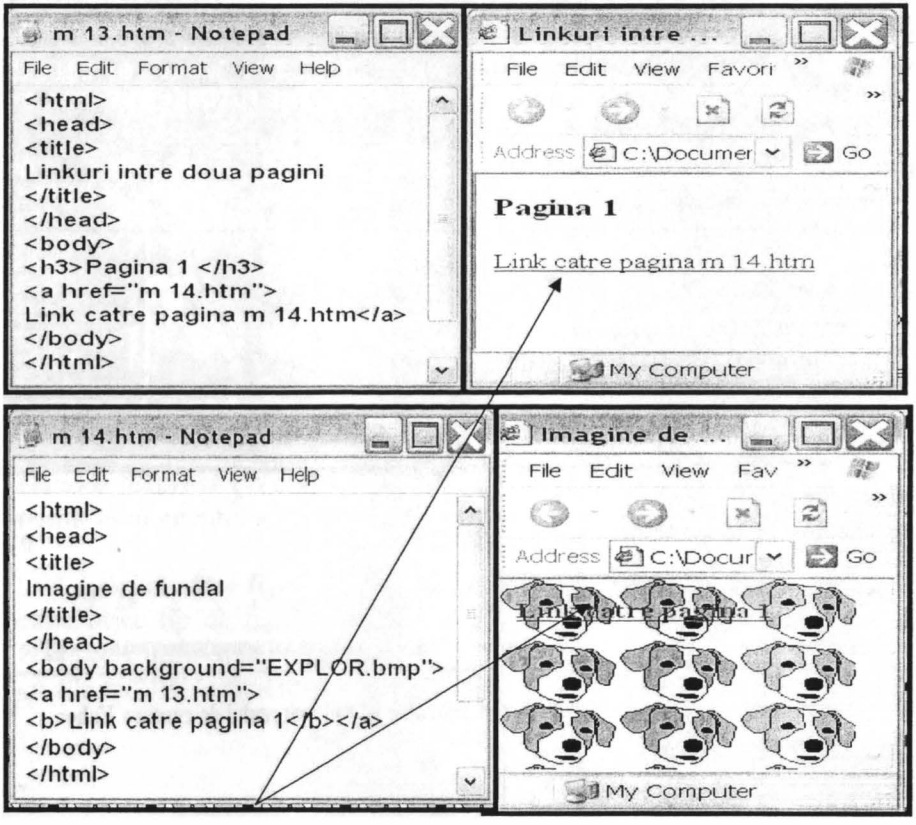
Se poate vizita <A HREF =http://www.fmi.unibuc.ro/**.>Universitatea Bucureşti</A> on line!**

Observatie. Toate apostroafele din codul HTML şi toate ghilimelele trebuie să fie drepte şi nu rotunjite.

Toate ancorele (referinţele hipertext), indiferent de tipul informaţiei indicate, arată la fel într-o pagină Web. Este bine să evităm folosirea etichetelor de genul *Click aici* pentru referinţele hipertext; paginile Web atractive se obţin prin integrarea creativă, semnificativă şi neinsidioasă a legăturilor în text.

Exemple.

1. Legătura către o alta pagină aflată în acelaşi director se face folosind etichetele **<a>...**. (de la "anchor"=ancoră), atributul *href* precizand pagina indicată de legătură, aşa cum se vede pe codurile HTML următoare:



https://biblioteca-digitala.ro / https://unibuc.ro

Se observă cele două link-uri care accesate cu mouse-ul va vor muta dintr-o pagina in alta.

2.Legatură către o pagină aflată pe acelaşi disc local, dar în alt director, se face folosind adresarea relativă, ca în codul HTML următor:

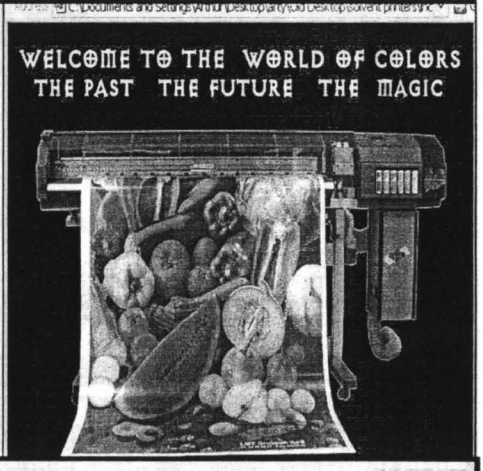
m 15.htm - Notepad

File Edit Format View Help

```
<html>
<head>
<title>
Link catre o pagina aflata pe acelasi disk local dar in director diferit
</title>
</head>
<body>
<h3>Pagina 2 </h3>
<a href="C:\Documents and Settings\Arthur\Desktop\arty\Old Desktop\solvent printers\indhold1.htm">
Link catre pagina indhold1.htm</a>
```

Se observă că pentru a putea ajunge la fişierul dorit de pe hard, s-a dat ***întreaga cale***, adică adresa relativă a fişierului respectiv sub forma:

<a href="C:\Documents and Settings \Arthur\Desktop\arty\Old Desktop\ solvent printers\indhold1.htm">



3.Legătura către un site oarecare se face precizând URL-ul site-ului respectiv.

m 16.htm - Notepad

File Edit Format View Help

```
<html>
<head>
<title>
Link catre site-ul Yahoo
</title>
</head>
<body>
<h3>Link catre site-ul motorului de cautare Yahoo </h3>
<a href="http://www.yahoo.com"><b>YAHOO</b> </a>
</body>
```

Link catre site-ul Yahoo - Microsoft

File Edit View Favorites Tools Help

Address C:\Documents and Settings\Arthur\Desktop Go

Link catre site-ul motorului de cautare Yahoo

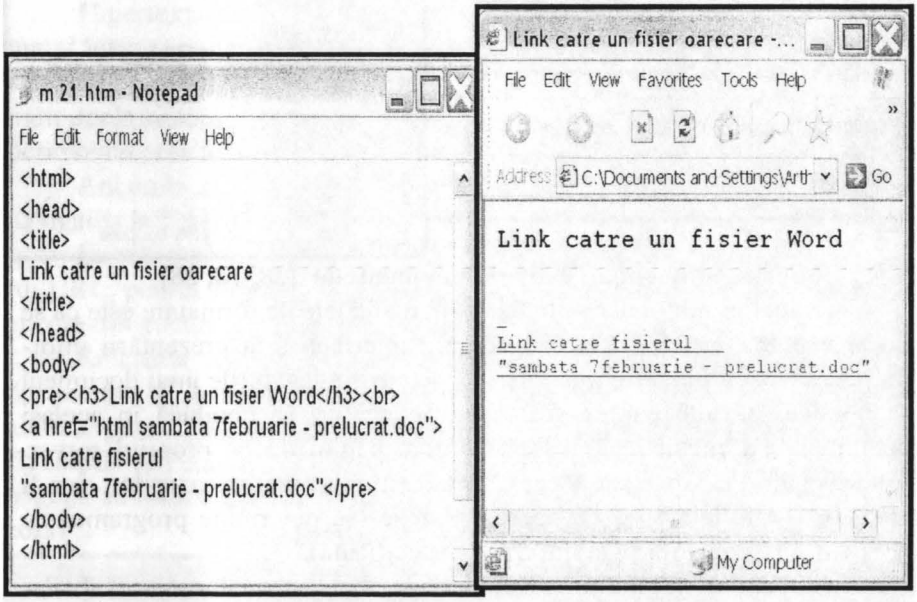
YAHOO

My Computer

https://biblioteca-digitala.ro / https://unibuc.ro

Exemplul anterior încarcă pagina de start a site-ului *YAHOO* utilizand adresa URL www.yahoo.com

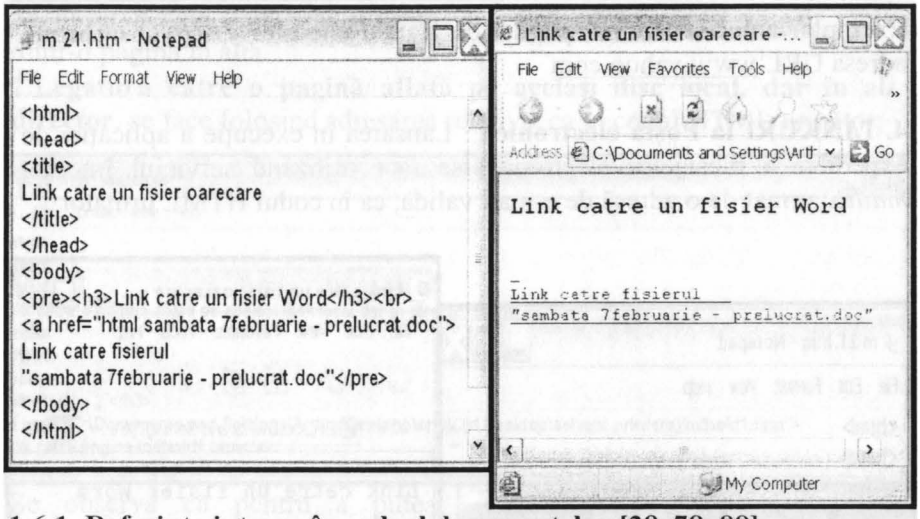
4. LINKURI la Poşta electronică . Lansarea în execuţie a aplicaţiei de expediere a mesajelor electronice se face utilizând serviciul Internet *mailto:* urmat de o adresă de e-mail validă, ca în codul HTML următor:



Observaţie. Exemplul funcţionează dacă pe calculator este instalat *Outlook Express* (Windows) , sau *Pine* (Unix) şi dacă Internetul este pornit în momentul accesării.

5. **Legături către fişiere oarecare**. O pagină Web poate conţine legături către orice tip de fişier, aflat pe orice server. Pentru aceasta în fişierul HTML se utilizează eticheta **<a>**, care are valoarea atributului *href* egală cu adresa URL a fişierului destinaţie.

https://biblioteca-digitala.ro / https://unibuc.ro



1.6.1 Referințe interne în cadrul documentelor [28, 78, 88]

Avantajul major al atributelor din marcajele de formatare este că se poate asigura o excelentă consistență a interfeței și a prezentării informației. De exemplu, este posibil ca jumătate din legăturile unui document să conducă la alte pagini Web, cateva legături să conducă în același document, iar restul legăturilor să refere pagini de pe propriul server. Toate legăturile vor avea aceeași formă (în majoritatea cazurilor, vor fi albastre și subliniate) și cu aceeași funcție (să determine programul de navigare să "sară" direct la informația specificată).

Țintele salturilor interne din documentele Web sunt cunoscute sub numele de *ancore cu nume*. Marcajul HTML pentru un punct-ancoră este o variantă a marcajului ancoră: <A nume=valoare>. Valoarea poate fi orice secvență de caractere, numere și semne de punctuație. Unele programe de navigare impun ca toate caracterele din ancoră să fie litere mici.

Deoarece ancorele sunt destinații în cadrul paginii curente și nu legături către exterior, textul dintre <A NAME> și </A> nu este scos în evidență în nici un fel atunci când este afișat. Totuși, deoarece definiția punctului de destinație este un marcaj de tip ancoră obișnuit, trebuie închis ca orice alt marcaj pereche, pentru a avea un marcaj corespunzător </A> pentru fiecare ancoră cu nume.

Corespondentul unei ancore în HTML este marcajul de formatare care definește *saltul* sau legătura activă din cadrul documentului. Aceasta este o variantă a marcajului <A>; atributul necesar este o altă referință

https://biblioteca-digitala.ro / https://unibuc.ro

hipertext de tip HREF, URL-ul fiind înlocuit de data aceasta cu numele ancorei, prefaţat de simbolul diez(#).

De exemplu, dacă ancora la care se face conectarea este specificată prin <A NAME="glosar">, saltul ar trebui specificat prin <A HREF="#glosar">Un salt la glosar</A>.

O modalitate frecventă în care se pot utiliza ancorele cu nume este crearea unei linii succinte de sumar la începutul documentului.

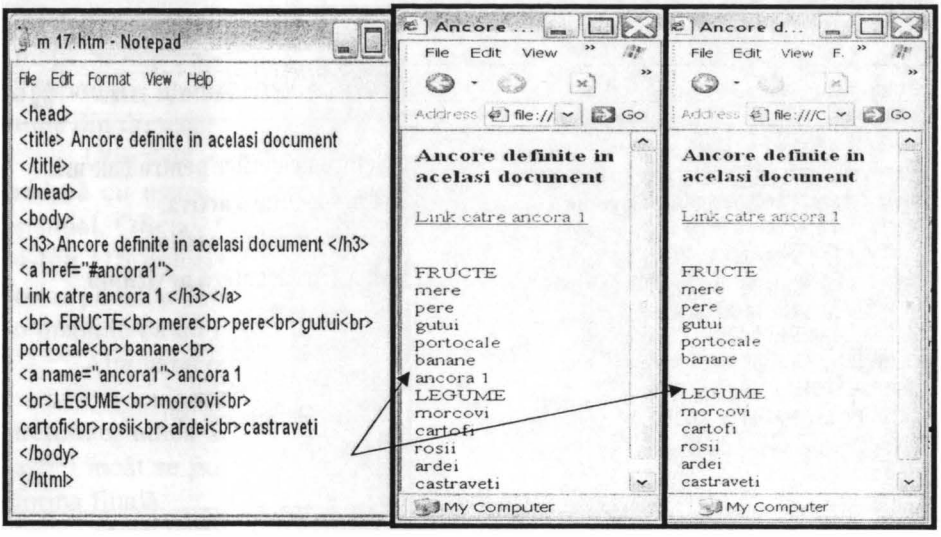
Hipertextul este o extensie logică a stilului actual de scriere, cu remarci între paranteze, definiţii de termini pe margine, note de subsol, citate din articole şi alte elemente similare. Hipertextul reprezintă însă mai mult decât simple legături de la un document la altul, iar limbajul HTML permite crearea tuturor acestor referinţe în cadrul aceleiaşi pagini.

Ancorele şi punctele de salt sunt folosite frecvent pentru a ajuta navigarea în liste de dimensiuni mari, cu informaţii sortate alfabetic.

Există şi notaţia *#ancora*, care se poate adăuga şi la sfârşitul oricărui URL, penrtu a face ca acea legătură să refere direct un anumit punct-ancoră din documentul respectiv.

Referirea ancorelor externe poate fi utilă pentru legături către documente Web mari, care conţin o cantitate mare de informaţii şi care altfel ar putea deruta.

Exemple. 1. Pentru a insera o legătură către "ancora1" definită în aceeaşi pagină, se utilizează eticheta **<a>**, având atributul *href* cu valoarea "#ancora1".



https://biblioteca-digitala.ro / https://unibuc.ro

In exemplul anterior cu lista de fructe şi legume ancora poziţionează pagina la LEGUME.
Se observă că între cele două pagini WEB afişate exista o deosebire. Aceasta provine de la eliminarea numelui ancora 1 din programul de mai sus scriind instrucţiunile corespunzătoare sub forma:

(<a name="ancora1">

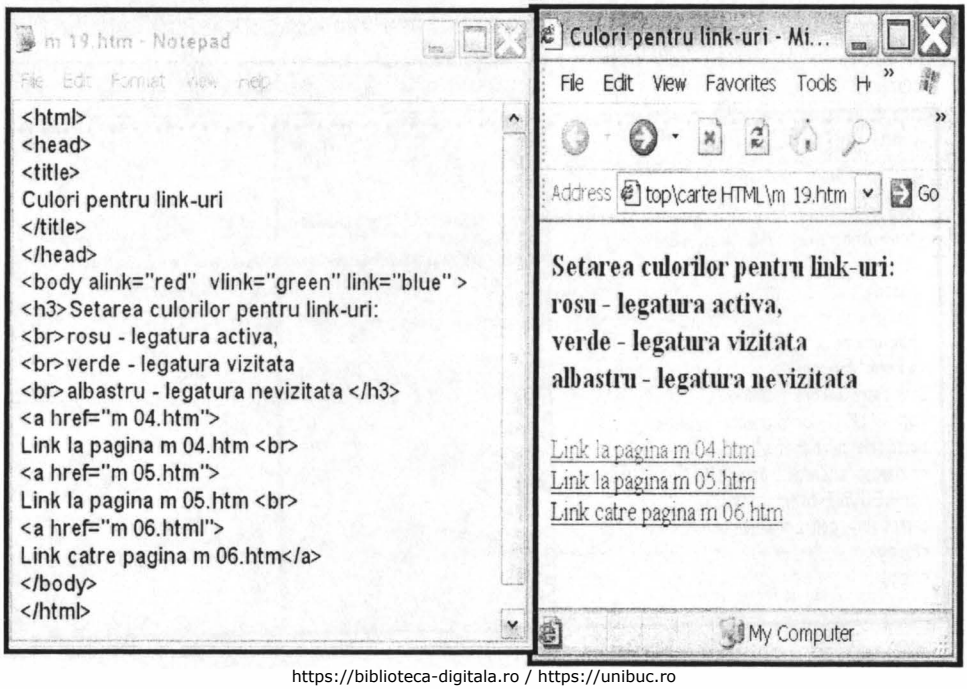
LEGUME
morcovi
)

Observaţie. Legătura către o ancoră definită în altă pagină din acelaşi director, se face tot cu atributul *href* dar care are in acest caz forma "nume_fisier.html#nume_ancora".

2. Colorarea Link-urilor. Linkurilor li se pot atribui culori. În mod prestabilit, legăturile pot primi trei culori, ce pot fi stabilite cu ajutorul a trei atribute ale etichetei **<body>**:

1. *link* spre legăturile nevizitate (nu s-a efectuat nici un clic pe ele);
2. *vlink* spre legăturile vizitate (s-a efectuat cel puţin un clic pe ele);
3. *alink* spre legăturile active (deasupra cărora se află mouse-ul la un moment dat).

Valorile pe care le pot lua aceste atribute sunt culori definite prin nume, sau conform standardului RGB .
Prezentul exemplu evidenţiază colorarea legăturilor:



https://biblioteca-digitala.ro / https://unibuc.ro

O listă completă a marcajelor HTML pentru referințe și legături se află în Anexă.

1.7 Imagini grafice si fragmente audio [77, 78, 88]

Există multe formate grafice acceptabile, dintre care numai două sunt larg înțelese de software-ul de navigare Web, si anume:

- GIF: *Graphics Interchange Format* introdus de Compuserve;
- JPEG: formatul introdus de *Joint Photographic Expert Group.*

Imaginile GIF pot utiliza doar 256 culori, pe când JPEG permite milioane de culori distincte într-o imagine (deși afișarea lor depinde de tipul de display pe care utilizatorul îl are în sistemul său de calcul). Ambele formate grafice comprimă imaginile pentru a reduce dimensiunea fișierelor dar, deoarece comprimarea se face în mod diferit, unele imagini vor avea dimensiuni mult mai mici într-un format decât în celălalt.

Motivul pentru care formatul de imagine GIF este mai mult folosit de către proiectanții de Web nu este paleta mică de culori, ci faptul că aceasta poate fi redusă până la minimul necesar pentru o imagine specifică, ceea ce conduce la o scădere drastică a dimensiunii fișierului.

Imaginile grafice sunt construite din pixeli (puncte distincte de informație din imagine). În cazul unei imagini GIF, fiecare pixel poate avea una din cele maxim 256 culori distincte. Dacă imaginea utilizează doar două culori în loc de 256, dimensiunea imaginii GIF poate fi redusă în mod considerabil, deoarece fiecare pixel necesită un singur bit de informație, comparativ cu 8 biți de informație pentru opțiunea completă de 256 culori. Cu alte cuvinte, dimensiunea fișierului va fi redusă la o optime din dimensiunea inițială.

Utilizând orice editor grafic performant, paleta de culori poate fi redusă cu ușurință astfel încât dimensiunea fișierelor să fie redusă substanțial. Oficial, GIF permite formate de 1 bit (2 culori), 2 biți (4 culori), 4 biți (16 culori), 8 biți (256 culori). Formatul de 1 bit poate lucra cu orice pereche de culori, deci imaginea care este albă pe fond albastru este o imagine cu un singur bit de culoare.

Alte caracteristici utile ale imaginilor GIF sunt posibiliatea de a desemna o culoare drept culoare transparentă și crearea graficelor întrețesute, adică a imaginilor care apar într-o pagină Web linie cu linie, astfel încât se poate vedea cum "ia naștere", de la o formă brută până la forma finală.

https://biblioteca-digitala.ro / https://unibuc.ro

Formatul GIF este utilizat în special pentru butoane, bannere (pa nouri publicitare) şi alte elemente similare, datorită dimensiunii mai mici a fişierelor.

Formatul JPEG este necesar pentru a imita cât mai bine culorile exacte ale imaginii originale.

Limbajul HTML permite includerea imaginilor (stocate) in cadrul paginii WEB. Formatele acceptat de browsere pentru fişierele imagine sunt:

- BMP (BitMap) extensie .bmp (numai cu *Internet Explorer*);
- XBM (X BitMap) extensie .xbm;
- GIF (Graphics Interchange Format) extensie .gif; (8 biţi pentru o culoare, 256 culori posibile)
- JPEG (Joint Photographic Experts Group) extensie .jpeg sau .jpg; (24 biţi pentru o culoare, 16777216 de culori posibile)
- XPM (X PixMap) extensie .xmp;
- TIFF (Tagged Image File Format) extensie .tif sau .tiff;

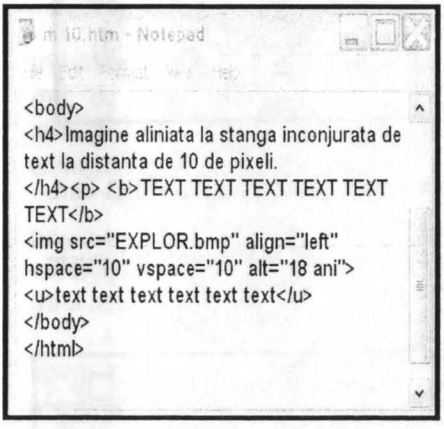
Toate imaginile au o adresă URL ("Uniform Resourse Locator" = identificator unic al resursei) exprimată în funcţie de directorul ce conţine documentul HTML care face referire la imagine.

O imagine poate fi: încadrată, redimensionată, aliniată, fundal pentru o pagină web, link către o altă pagină. Pentru a insera o imagine într-o pagină, se utilizează eticheta **<img>** (de la "image"=imagine)



https://biblioteca-digitala.ro / https://unibuc.ro

cu atributul *src* (de la "source"=sursă, identifică imaginea inserată) şi/sau cu atributul *border* pentru încadrarea în chenar. Valorile acestui atribut sunt numere întregi pozitive, aşa cum se vede în exemplul alăturat.
Dimensiunile prestabilite ale unei imagini pot fi modificate cu atributele width şi height.
Un exemplu de imagine înconjurată de text este următorul:



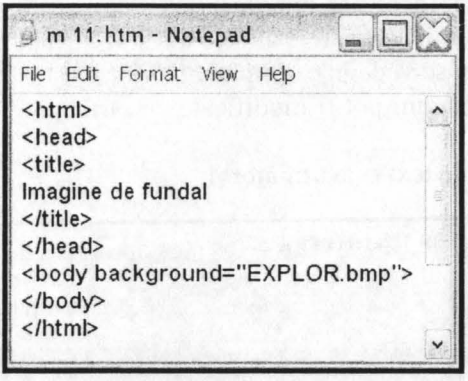


Dacă stabilim pentru atributul *border* al etichetei **<img>** valoarea 0, acest chenar dispare.
O imagine se poate alinia folosind atributul *align* care are valorile:

- "left" - aliniere la stânga;
- "right" - aliniere la dreapta;
- "top" - aliniere deasupra;
- "middle" - aliniere la mijloc;
- "bottom" - aliniere la bază;

Distanţa în pixeli pe orizontală, respectiv verticală, dintre imagine si celelalte elemente din pagină este data de atributele *hspace* şi *vspace*.
Daca se doreste ca in locul imaginii sa apara un text se foloseste atributul *alt* .
O **imagine** poate fi folosita ca **fundal** pentru o pagina Web, folosind atributul *background* al etichetei **<body>**. Imaginea se va multiplica pe orizontală şi verticală până când va umple ecranul, aşa cum se vede pe exemplul următor:

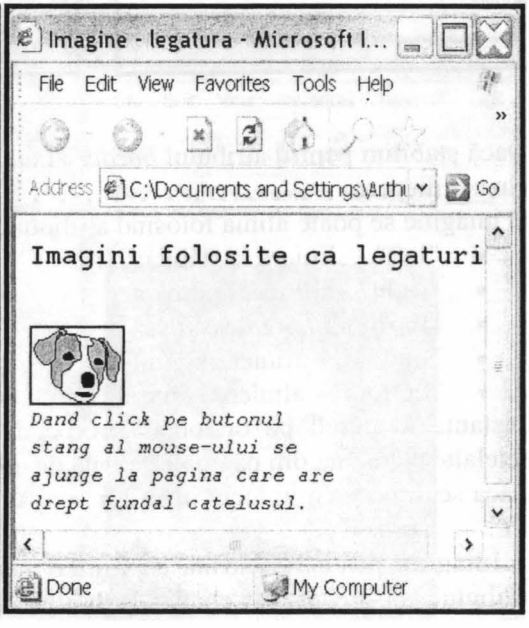
https://biblioteca-digitala.ro / https://unibuc.ro





O imagine poate fi folosită şi ca **legătură** către o altă pagină WEB care va fi accesată automat.
Un exemplu în acest sens ar fi următorul cod HTML, care foloseşte imaginea „EXPLOR.bmp" drept linie către pagina m11.htm :





https://biblioteca-digitala.ro / https://unibuc.ro

Includerea imaginilor într-un document Web este imediată, folosind marcajul de formatare <IMG> (imagine). La fel ca marcajul ancoră <A>, marcajul <IMG> are un singur atribut esențial, SRC=numegrafic și, la fel ca marcajul <HR>, nu are marcaj pereche pentru închidere.
Pentru a include imaginea din fișierul banner.gif, codul HTML ar fi:

<IMG SRC=banner.gif>

Atunci când este vorba despre un fișier GIF, se folosește controlul <IMG> pentru a insera acel fișier în text.

Marcajul <IMG> permite diferite atribute, dar singurul atribut care trebuie obligatoriu să apară este o specificare a fișierului sursă a imaginii, de tipul SRC=numefișier. De exemplu, se poate elimina chenarul albastru din jurul unei imagini grafice care este și hiperlegătură, prin adăugarea unui alt atribut al marcajului <IMG>: BORDER=0.

Un alt atribut al marcajului <IMG> este ALT=text. Ceea ce apare pe post de text alternativ este afișat în situația în care utilizatorul nu poate vizualiza imagini sau optează pentru eliminarea încărcării grficelor, pentru a accelera navigarea în Web.

Atât Internet Explorer cât și Navigator afișează acest text ALT imediat după încărcarea paginii, iar apoi înlocuiesc treptat fiecare element cu imaginea grafică propriu-zisă.

Un al treilea atribut al marcajului de formatare <IMG>, este ALIGN, care oferă un control precis al alinierii. Cele trei alinieri standard sunt ALIGN=top, ALIGN=middle și ALIGN=bottom. În mod implicit, imaginile și materialele adiacente sunt aliniate la baza imaginii.

Cele trei opțiuni principale de aliniere a imaginilor se referă la informația care apare după imaginea în sine. Există un set suplimentar de opțiuni de aliniere care se referă la alinierea imaginii în cadrul ferestrei și nu la cea a materialului adiacent, în raport cu imaginea grafică. Aceste opțiuni suplimentare de aliniere oferă un control mult mai avansat, dar conduc la formatări ceva mai confuze, datorită diferenței dintre alinierea imaginii și alinierea materialului adiacent.

Una din opțiunile noi permite specifiacrea lățimii (WIDTH) și înălțimii (HEIGHT) graficului, înainte ca acesta să fie încărcat, ceea ce face ca documentul să fie redat mult mai rapid pe ecran, chiar înainte ca graficul să fie recepționat. Valorile sunt specificate in pixeli.

Un alt atribut, care a fost menționat și mai devreme, este BORDER (chenar), care poate fi folosit cu mare efect, întru-cât permite specificarea lățimii exacte a chenarului în jurul unei imagini-legătură.

https://biblioteca-digitala.ro / https://unibuc.ro

Alte două atribute utile pentru alinierea imaginilor sunt VSPACE şi HSPACE, care controlează spaţiul vertical şi orizontal în jurul fiecărei imagini grafice.

Ţinând cont că toate imaginile grafice sunt specificate prin acelaşi marcaj HTML de bază, este remarcabilă varietatea care există în cadrul diverselor locaţii din Web. Proiectanţii Web creează diverse înfăţişări ale paginilor lor prin tipurile de imagini pe care le folosesc şi prin combinaţiile unice de grafică, text şi imagini de fond.

Cele mai uzuale imagini provin din urmatoarele surse:

- Imagini noi create chiar de autor;
- Clip art sau alte biblioteci de imagini "la cutie";
- Programe de prelucrare a textelor;
- Fotogarfii scanate;
- Preluarea imaginilor din Internet.

Un rezultat al creşterii interesului pentru proiectarea de pagini World Wide Web este marea varietate de biblioteci Clip art şi de imagini "la cutie", disponibile pe CD-ROM-uri şi discuri flexibile, care există în prezent.

Adobe Photoshop este un editor de imagni extrem de puternic. Prin adăugarea unor extensii provenite de la producători independenţi, cum ar fi excelentul pachet de filtrare Eye Candy, produs de Alien Skin Software, se pot genera grafice de text uimitoare, efectuând doar câteva clicuri cu mouse-ul.

Un alt set valoros de filtre provine de la compania Auto F/X, a cărei suită Photo/Edges permite crearea unei mari varietăţi de chenare deosebite şi interesante pentru fotografii şi alte imagini grafice. Adobe include un set mare de filtre, numit suita Gallery Effects, care permite transformarea imaginilor grafice în picturi, acuarele, gravuri şi în multe alte produse artistice.

Un alt mod de a produce imagini grafice pentru o locaţie Web este utilizarea unui scanner şi a unor imagini artistice existente. O altă modalitate de lucru este scanarea notiţelor sau a desenelor create cu pixuri, creioane, culori sau orice altceva este la îndemână, şi încorporarea acestor imagini în pagina de Web.

Mai există o cale de a prelua imagini, care nu implică simţ artistic sau utilizarea unui scanner. Se pot găsi grafice interesante şi atractive online. Există un număr de locaţii care conţin arhive cu grafice, clip art, imagini de fond şi altele, toate fiind disponibile public.

https://biblioteca-digitala.ro / https://unibuc.ro

Fragmentele audio sunt probabil cele mai distractive şi acestea sunt destul de uşor de introdus în paginile proprii. Aceste înregistrări audio sunt de obicei în formatul care se numeşte micro-law (întâlnit în forma scrisă ca mulaw) şi pot fi introduse butoane sau zone active, la fel ca orice URL.

Utilizatorii care efectuează clic pe fraza un *extras din ultimul meu album* vor descărca un fişier audio, apoi va fi lansat un program de redare audio, pentru a asculta efectiv clipul respectiv.

Un alt mod de a introduce clipuri audio este utilizarea extensiilor HTML EMBED sau BGSOUND.

Trebuie să avem grijă atunci când adăugăm clipuri audio unei locaţii, deoarece aceste fişiere pot creşte foarte mult în dimensiune. Un clip audio de zece secunde poate ajunge la 150K, ceea ce reprezintă un timp destul de mare de descărcare pentru utilizatorii care au acces la Web prin conexiuni telefonice de viteză redusă. Fişierele audio trebuie salvate cu extensia AU sau WAV.

Filmele sunt disponibile în două formate: Quick Time şi MPEG (Motion Picture experts Group – Grupul Experţilor în Cimenatografie). Formatul adecvat pentru Web este însă MPEG, deoarece acesta este aproape universal şi există programe de redare MPEG pentru Mac, PC şi Unix. Fisierele video trebuie salvate cu extensia **.mpg** .

Programele de navigare Web văd extensia MPG şi ştiu să descarce fişierul specificat şi să lanseze un program de redare video.

O altă tehnologie care a apărut recent este aşa-numita tehnologie de medii de tip flux. In acest caz utilizatorul nu aşeaptă până se descarcă întreaga secvenţă video sau adio, deoarece se descarcă atât cât este suficient pentru a începe redarea respectivei secvenţe.

Paginile Web atractive sunt acelea care încorporează în mod inteligent imaginile grafice într-un design general şi care nu se dezintegrează sau devin inutilizabile atunci când utilizatorii nu doresc sau nu pot să încarce toate materialele conţinute în pagină. *Locaţiile Web valoroase se bazează pe conţinut, nu pe înfăţişare.*

Marcajele HTML pentru imagini se află în Anexă.

https://biblioteca-digitala.ro / https://unibuc.ro

1.8 Tabele si cadre [28, 81, 88]

Tabelele reprezintă o extensie importantă a limbajului HTML, avându-şi originea în laboratoarele de dezvoltare ale corporaţiei Netscape Communications. Dar, spre deosebire de tabelele din alte procesoare de text, tabelele HTML pot fi destul de solicitante, deoarece ele sunt soluţia optimă pentru materialele adiacente dintr-o pagină, de exemplu text pe mai multe coloane.

Deşi tabelele oferă o mulţime de posibilităţi interesante, au şi un dezavantaj: sunt destul de greu de construit. Trebuie specificaţi parametrii tabelului, apoi parametrii fiecărei linii iar, în final, fiecare celulă trebuie înconjurată de marcajele <TD> </TD> pentru date. Marcajul specific tabelelor este <TABLE>.

În orice element al unui tabel se pot include grafice, text şi aproape orice altceva, inclusiv alte tabele. Fiecare linie poate avea o aliniere proprie, specificată prin ALIGN=, ca parte a marcajului, iar marcajul TABLE însuşi poate avea o mulţime de opţiuni.

Controlul lăţimii tabelului astfel încât conţinutul să nu apară înghesuit se realizeaza prin atributele WIDTH şi CELLPADDING.

Atributul WIDTH specifică lăţimea exactă a tabelului de pe ecran, indiferent de conţinut, lăţime dată fie sub forma unui număr de pixeli, fie ca procent din lăţimea totală a ferestrei curente de vizualizare.

Cealaltă modalitate de a lăţi tabelul este specificarea unui factor CELLPADDING. Există două atribute, care, la prima vedere, par similare, dar care servesc unor scopuri imporatnte şi distincte în redactarea tabelului. CELLPADDING indică spaţiul, în pixeli, dintre marginea interioară a chenarului unei celule şi materialul conţinut în celulă, pe când CELLSPACING se referă la lăţimea linilor de caroiaj dintre celulele de date.

În cadrul unui tabel, pe lângă specificarea linilor prin <TR> şi a elementelor individuale de date prin <TD>, se pot specifica şi antete de coloană prin marcajul <TH>, care înlocuie marcajul <TD> în definiţia linei. <TH> este aproape identic cu <TD>, cu două excepţii importante: textul din cadrul unui marcaj <TH> trebuie scris cu aldine şi centrat orizontal în celulă.

Alinierea orizontală a celulelor de date se poate face cu ajutorul opţiunii ALIGN, pentru care există variantele: ALIGN=left (la stânga), ALIGN=center (centrat)şi ALIGN=right (la dreapta). VALIGN permite

https://biblioteca-digitala.ro / https://unibuc.ro

să specificăm alinierea pe verticală. Opțiunile pentru VALIGN sunt top (sus), middle (centrat) și bottom (jos). De exemplu, VALIGN=top face ca toate celulele dintr-o linie să conțină informația aliniată în partea superioară.

Culorile dintr-o anumită celulă de date se specifică în marcajul <TD> prin atributul BGCOLOR ca în exemplul următor: **<TD BGCOLOR=yellow>text pe fond galben</TD>**

Această definiție va genera o celulă cu fond galben și textul scris, în mod predefinit, cu negru.

Atributele COLSPAN si ROWSPAN permit extinderea unei celule de date pe mai multe coloane respectiv linii ale tabelului.

Cea mai recentă specificație HTML detaliază modul de definire a culorilor de fond la nivel de celule, utilizând BGCOLOR, dar Internet Explorer adaugă:

- marcajul BACKGROUND, care permite grafice de fond în interiorul celulelor din tabele;
- marcajul BORDERCOLOR, care permite controlul detaliat al culorii chenarului;
- marcajele BORDERCOLORLIGHT și BORDERCOLORDARK definesc cele două culori folosite pentru chenar;
- BACKGROUND=fișier-grafic, care permite specificarea unei imagini grafice de fond în cadrul celulelor individuale de date.

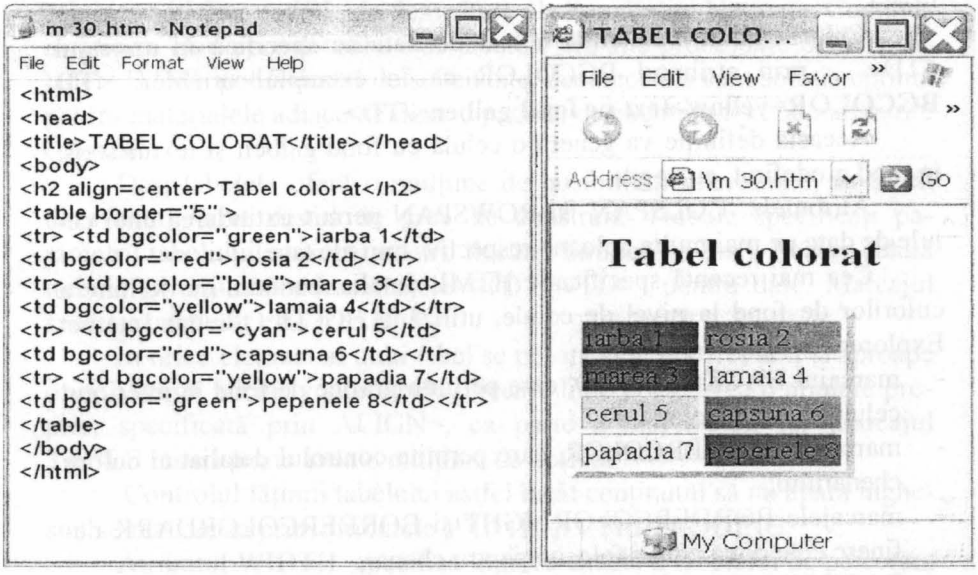
Controlul mai precis al chenarelor din jurul tabelelor și al caroiajului dintre celulele de date se realizează prin atributul RULES care are următoarele valori:

.VOID	Elimină toate muchiile exterioare ale tabelului
ABOVE	Afișează o muchie în partea superioară a cadrului tabelului
BELOW	Afișează o muchie în partea inferioară a cadrului tabelului
HSIDES	Afișează câte o muchie în partea superioară și inferioară a cadrului tabelului
LHS	Afișează o muchie în partea stângă a cadrului tabelului
RHS	Afișează o muchie în partea dreaptă a cadrului tabelului
VSIDES	Afișează o muchie în partea din stânga și din dreapta a cadrului tabelului
BOX	Afișează câte o muchie pe toate laturile cadrului tabelului
BORDER	Afișează câte o muchei pe toate laturile cadrului tabelului

https://biblioteca-digitala.ro / https://unibuc.ro

Exemple.

1. Un program HTML care afişează un tabel colorat în care celulele conţin diverse obiecte colorate în culoarea lor naturală :



2. Următoarea tabelă, care prezintă câteva diferenţe între versiunile 1.0, 2.0 şi 3.0 ale HTML va fi descrisă în limbajul HTML :

Diferenţe între diverse versiuni HTML

Facilităţi	HTML 1.0	HTML 2.0	HTML 3.0
Hărţi şi imagini active		X	X
Ecuaţii			X
Formulare		X	X
Hiperlinkuri	X	X	X
Imagini	X	X	X
Liste	X	X	X
Bare de instrumente			X
Tabele			X

Pentru a obţine această tabelă vom folosi următorul cod HTML
<HTML><HEAD><TITLE> Pagină cu o tabelă </TITLE></HEAD>
<BODY>

https://biblioteca-digitala.ro / https://unibuc.ro

<TABLE border = ALL Rules = ALL>
<CAPTION> Diferențe între diverse versiuni HTML </CAPTION>
<COL ALIGN = LEFT>
<COL ALIGN = CENTER>
<COL ALIGN = CENTER>
<COL ALIGN = CENTER>
<TR><TH> Facilități <TH> HTML 1.0 <TH> HTML 2.0 <TH> HTML 3.0
<TR><TH> Hărți și imagini active <TD><TD> X <TD> X
<TR><TH> Ecuații <TD><TD><TD> X
<TR><TH> Formulare <TD><TD> X <TD> X
<TR><TH> Hiperlinkuri <TD> X <TD> X <TD> X
<TR><TH> Imagini <TD> X <TD> X <TD> X
<TR><TH> Liste <TD> X <TD> X <TD> X
<TR><TH> Bare de instrumente <TD><TD><TD> X
<TR><TH> Tabele <TD><TD><TD> X
</TABLE>
</BODY>
</HTML>

In anexă se găsesc listate toate marcajele necesare realizării de tabele în HTML.

Cadrele răspund la întrebarea ce se întâmplă dacă fiecare celulă dintr-un tabel ar fi o pagină Web.

Spre deosebire de multe marcaje întâlnite până acum, cadrele reprezintă o afirmație de tip totul sau nimic: cadrele individuale se specifică prin marcajul <FRAME>, care este la rândul său inserat într-un specificator <FRAMESET>, ce indică volumul de spațiu alocat fiecărui "panou" de informație.

Argumentul principal al documentelor bazate pe cadre este următorul: în locul unei singure pagini care să definească toată informația afișată utilizatorului, aceasta este împărțită în pagini multiple, fiecare având propria sa porțiune de date.

Cel mai important marcaj pentru cadre este <FRAMESET>. Acesta creează un set de cadre: un grup de cadre în care se va împărți pagina Web. În plus față de opțiunea ROWS care permite împărțirea în panouri orizonatle, se poate folosi COLS pentru a specifica panouri verticale. Există trei valori diferite ce pot fi utilizate în cazul acestor atribute:

- un număr simplu, ce specifică dimensiunea dorită de pixeli

https://biblioteca-digitala.ro / https://unibuc.ro

- asteriscul (*) specifică restul de spațiu;
- un procent din lățime, prin notația n%.

Există câteva atribute care pot fi specificate la definirea cadrelor, cel mai important fiind atributul NAME=. Fiecărui cadru specific i se poate da un nume unic, similar cu <A NAME=>, care poate fi folosit ca mijloc de a controla care din ferestre să fie afectată de anumite acțiuni.

Deoarece nu toți utilizatori sunt în măsură să vizualizeze o pagină bazată pe cadre, este necesară o secțiune în cadrul paginii principale cu cadre, care conține opțiunea NOFRAMES. Dacă programul de navigare ştie să interpreteze cadre, va ignora ce se găseşte în această secțiune, iar dacă nu, materialul cuprins în zona NOFRAMES va fi singurul care va fi înțeles şi afişat

Revenind la tabele, modificarea cea mai de efect care poate fi făcută, este posibilitatea de a face textul să "curgă" în jurul tabelului, în loc de a se afişa un tabel de sine stătător, cu text deasupra şi dedesubt. Acest lucru este realizat de atributul ALIGN din cadrul marcajului <TABLE>. ALIGN=left face ca tabelul să fie aliniat în partea stângă a paginii, iar textul să fie scris în partea dreaptă a tabelului, iar ALIGN=right deplasează tabelul în partea dreaptă a paginii şi determină "curgerea" textului în partea stângă. Atributul ALIGN=center centrează tabelul pe orizontală, dar nu permite "curgerea" textului pe nici una dinter părțile laterale.

Atributele HSPACE şi VSPACE ne permit să specificăm pixeli suplimentari pentru chenar şi pentru a mări spațiul dintre tabel şi textul înconjurător, atât la stânga si la dreapta (pe orizontală cu HSPACE), cât şi deasupra şi dedesubt (pe verticală cu VSPACE). Aceste atribute sunt utilizate exact la fel ca în cazul celor pentru marcajele <IMG> şi a celor pentru includerea imaginilor.

Un alt element al marcajului <TABLE> este <CAPTION>, care permite specificarea unui text explicativ acociat tabelului. Există patru opțiuni de aliniere diferite pentru textul explicativ: top (în partea superioară), bottom (în partea inferioară), left (în partea stângă) şi right (în partea dreaptă). Dacă dorim ca textul explicativ asociat tabelului să fie poziționat într-o parte a sa, putem utiliza ALIGN=left sau ALIGN=right.

Ca şi în cazul tabelelor, există aspecte referitoare la *proiectarea paginilor pe baza cadrelor* care ar putea îmbunătății aspectul paginii Web.

https://biblioteca-digitala.ro / https://unibuc.ro

Pentru a scăpa de elementele nedorite, precum chenarul din jurul fiecărui cadru dintr-o pagină Web care le utilizează, in cazul programului Netscape Navigator se adaugă atributul BORDER=0 la marcajul <FRAMESET>. Internet Explorer necesită adăugarea atributelor FRAMEBORDER=0 şi FRAMESPACING=0 la marcajul <FRAMESET> pentru a determina eliminarea chenarului. In cazul ambelor programe de navigare se pot specifica atribute redundante, deoarece programul de navigare va interpreta numai atributele pe care le înțelege.

Pentru deplasarea conținutului paginii față de marginile cadrului, se combină atributele MARGINHEIGHT şi MARGINWIDTH, care specifică, în pixeli sau procentual, cât de departe de marginea superioară şi marginea din stânga este poziționat conținutul. Cele două atribute au acelaşi efect în cazul ambelor programe de navigare.

O altă opțiune necesara unui cadru este specificare atributului NORESIZE, atunci când nu se doreşte ca utilizatorul să tragă de chenar pentru a-l deplasa in cadru, redimensionând totul după cum consideră că este mai bine. Combinat cu atributul SCROLLING=no, se poate elimina o parte a imaginii sau a paginii, nelăsând utilizatorului nici o posibilitate de a viziona întregul. Aceste două atribute trebuie specificate în cadrul marcajului <FRAME> şi nu în cadrul marcajului<FRAMESET>.

Pentru a "indica" evenimentele unui anumit cadru, se foloseşte marcajul <FRAME> cu atributul NAME= pentru a specifica numele cadrului, iar apoi, în paginile de navigare, TARGET= ca parte HREF pentru a determina ca aceste evenimente să afecteze cadrul specificat şi nu pe cel în care se lucrează. Pot fi specificate şi alte valori pentru atributul TARGET, valori care asigură un control mai riguros asupra evenimentelor. Printre acestea avem:

_blank încarcă documentul într-o fereastră nouă, fără nume
_self încarcă documentul în fereastra curentă
_parent încarcă documentul în fereastra părinte(numai în cazul în care pe ecran există mai mult de o fereastră)
_top încarcă documentul în fereastra din partea superioară, anulând în felul acesta toate celelalte cadre care ar putea fi pe ecran

Utilizarea valorilor speciale TARGET îmbunătățeste considerabil un proiect de pagină bazat pe cadre şi oferă, spre exemplu, o ferastră de navigare care se păstrează chiar şi atunci când utilizatorul vizitează alte pagini ale locației.

https://biblioteca-digitala.ro / https://unibuc.ro

Una din facilitățile cele mai interesante pe care Microsoft le-a introdus în limbajul HTML prin intermediul popularului său porgram de navigare Internet Explorer este conceptul de **cadre interne**, adică ferestre de cadre care sunt complet înconjurate de o fereastră exterioară.

Un cadru intern, adică o fereastră de cadru înconjurată complet de o fereastră exterioară este specificat prin intermediul marcajului <IFRAME> într-un mod asemănător modului în care se specifică marcajul <FRAME>.

Există o serie de opțiuni ale marcajului <IFRAME> printre care FRAMEBORDER, care poate avea valoarea 0 sau 1, în funcție de dorința de a avea sau nu chenar, MARGINWIDTH și MARGINHEIGHT care asigură controlul fin al spațierii dintre marginile cadrului intern și conținutul acestuia și SCROLLING care poate fi yes (da) no (nu) sau auto (automat), ca și în cazul marcajului <FRAME>.

Unui cadru intern i se poate atribui un nume prin intermediul atributului NAME și pot fi specificate legături către acesta cu ajutorul atributului TARGET, ca și în cazul cadrelor obișnuite.

Marcaje HTML pentru relizarea de cadre se află în Anexă.

1.9 Formulare [81, 77, 88]

Formularele asigură construirea unor pagini Web care permit utilizatorilor să introducă efectiv informații și să le transmită serverului. Formularele pot varia de la o simplă casetă de text, pentru introducerea unui șir de caractere pe post de cheie de căutare – element caracteristic tuturor motoarelor de căutare din Web – până la o structură complexă, cu multiple secțiuni, care oferă facilități puternice de transmisie a datelor.

Toate formularele din Web sunt asemănătoare, dar există numai două căi prin care informațiile pot fi transmise de la programul de navigare Web la serverul situat în celălalt capăt al rețelei. Dacă sunt transmise informațiile dintr-un formular, și URL-ul de destinație include informațiile introduse, metoda se numește "get" (METHOD=GET), iar formularul este unul de tip "get". Alternativa este aceea în care atunci când sunt transmise informațiile, URL-ul de destinație arată perfect normal, fără alte informații adăugate la sfârșit. În această situație, avem de-a face cu metoda numită "post" sau cu un formular de tip "post".

Formularele HTML sunt delimitate de marcajul <FORM>, care se specifică prin <FORM ACTION=*url* METHOD=*metoda*> și prin marca-

https://biblioteca-digitala.ro / https://unibuc.ro

jul de terminare </FORM>. *Url*-ul indică fișierul sau aplicația aflată la distanță care urmează să prelucreze informațiile, iar *metoda* se specifică fie prin GET, fie prin POST.

În cadrul marcajului <FORM>, pagina Web poate conține orice informații HTML standard de formatare, grafice, legături către alte pagini și o mulțime de noi marcaje, specifice formularelor. În cea mai mare parte, toate câmpurile unui formular, destinate introducerii datelor, se specifică prin <INPUT> și prin diferite atribute. Aceste noi și variate marcaje permit definirea unei mari varietăți de elemente specifice formularelor asa cum ar fi:

<INPUT	Câmp de intrare pentru date de tip text sau pentru alte tipuri
TYPE=*opt*	Tipul câmpului de intrare <INPUT>
NAME=*nume*	Numele simbolic al valorii câmpului
VALUE=*valoare*	Conținutul prestabilit al câmpului de tip text
CHECKED=*opt*	Buton/casetă validată în mod prestabilit
SIZE=*x*	Numărul de caractere al casetei de text afișată
MAXLENGHT=*x*	Numărul maxim de caractere acceptate
<SELECT>	Grup de casete de validare
<TEXTAREA>	Câmp de intrare de tip text pe linii multiple

Programele de navigare Web curente permit lucrul cu nouă tipuri diferite de <INPUT>, fiecare dintre acestea determinând un tip diferit de output. Tipurile de intrări-utilizator sunt:

- TEXT este câmpul de intrare de tip șir de caractere; se admit și semnele speciale care încep cu caracterul ampersand, &. Mărimea câmpului este dată de SIZE, maxim 20 de caractere; dacă aici apar două valori numerice, a doua se referă la numărul rândurilor;
- PASSWORD (parolă) este un câmp de text, iar datele introduse de utilizator sunt afișate prin caracterul * sau alte simboluri, din motive de securitate. Opțiunea MAXLENGTH poate fi utilizată pentru a specifica numărul maxim de caractere permis pentru parolă;
- CHECKBOX (casetă de validare) asigură o singură casetă (negrupată) de validare; opțiunea CHECKED permite specificarea faptului că această casetă urmează sau nu să fie marcată în mod prestabilit. Opțiunea VALUE specifică textul asociat casetei de

https://biblioteca-digitala.ro / https://unibuc.ro

validare. Ea este de forma unui pătrățel în care apare un semn de bifare, dacă este selectată;

- HIDDEN (ascuns) permite traansmiterea informațiilor programului care prelucrează datele-utilizator, fără ca utilizatorul să le vadă pe ecran. Este un tip util, în mod deosebit în cazul în care pagina care conține formularul HTML este generată automat de un script CGI;
- FILE (fişier) oferă posibilitatea de a permite utilizatorilor să transmită serverului un fişier. Utilizatorii pot introduce numele fişierului sau pot executa un clic pe butonul Browse pentru a-l selecta din calculator;
- RADIO afişează un buton cu interblocare; diferitele butoane cu interblocare care au acelaşi NAME=valoare sunt grupate în mod automat, astfel încât un singur buton din grup poate fi selectat. El este de forma unui cerculeț în care apare un punct, atunci când este selectat.

Cele mai importante tipuri <INPUT> sunt:

- SUBMIT (transmite), care determină apariția în formular a unui buton, care, atunci când este apăsat, transmite conținutul întregului formular la serverul aflat la distanță;
- IMAGE, care este identic cu SUBMIT, numai că în locul unui buton, permite specificarea unei imagini grafice pentru transmisia datelor sau un buton de ENTER;
- RESET, permite utilizatorilor să şteargă conținutul tuturor câmpurilor formularului.

<SELECT> este un meniu derulant de opțiuni, având marcajul de încheiere </SELECT> şi parametrul <OPTION> care defineşte fiecare opțiune a meniului. Trebuie specificat un parametru NAME (nume) care identifică în mod unic întregul meniu definit de marcajul <SELECT>. De fapt, toate marcajele dedicate formularelor trebuie să aibă specificat propriul parametru NAME şi toate numele trebuie să fie unice în cadrul fiecărui formular individual.

În cadrul marcajului <SELECT> poate fi specificat, de asemenea, un parametru SIZE, care indică numărul articolelor de meniu care vor fi afişate odată şî parametrul MULTIPLE, care specifică dreptul utilizatorului de a selecta mai mult de o opțiune. Dacă există o valoare prestabilită, trebuie adăugat SELECTED (selectată, ca opțiune prestabilită) la marcajul <OPTION> pentru a indica acea valoare.

https://biblioteca-digitala.ro / https://unibuc.ro

Marcajul <TEXTAREA> permite construirea unei casete de intrare pe linii multiple. Ca şi marcajul <SELECT>, <TEXTAREA> solicită un nume unic, specificat prin NAME=. Marcajul <TEXTAREA> permite definirea dimensiunii casetei de intrare de tip text, cu ajutorul atributelor ROWS (linii) şi COLS (coloane), care precizează numărul de linii din cadrul casetei şi, respectiv, lungimea liniilor. Marcajul <TEXTAREA> are marcajul de terminare </TEXTAREA>.
Marcajul <TEXTAREA> are sintaxa următoare:
<TEXTAREA NAME="nume intern" ROWS=valoare COLS=valoare> valoare asumată </TEXTAREA>

Se pot exploata formularele de pe alte situri pentru a duplica intrările lor de date în propria pagină. De exemplu, se poate include o zonă de cautare Yahoo!, care să permită comutare rapidă la Yahoo!. Conectându-ne la pagina gazdă Yahoo!, putem vizualiza programul sursă şi, implicit, secvenţa complexă de linii care defineşte formularul de căutare. Aceasta se face printr-o secvenţă de cod HTML de forma:

```
<FORM ACTION ="http://search.yahoo.com/search" METHOD=get>
<INPUT SIZE=30 NAME=p>
<INPUT TYPE=SUBMIT VALUE=Search>
<FONT SIZE=-1>
<A HREF=" http://search.yahoo.com/search /options"> options</A>
</FONT>
</FORM>
```

Aceste linii definesc caseta de căutare care este afişată în partea superioară a paginii gazdă Yahoo!.

Revenind la marcajul INPUT, o altă valoare pe care o poate lua atributul TYPE este TYPE=BUTTON. Tipul BUTTON este un buton de uz general pentru o pagină Web, nici măcar asociat cu un anumit script, în timp ce evenimentul *click* trebuie definit de un script local.

Butonul de intrare poate fi un mecanism puternic de asistenţă pentru utilizatori, în explorarea unor locaţii Web complexe sau în afişarea unei ferestre cu o secvenţă ciclică de reclame, ori ca parte componentă a unui joc. Un alt artificiu util este:

```
<INPUT TYPE="button" VALUE=" Închide" onClick= "window.close()">
```

Dacă se execută clic pe butonul *Închide*, fereastra dispare. Funcţia JavaScript *window.close()* închide fereastra permitand utilizatorilor să se concentreze asupra paginii Web principale.
Vom ilustra crearea unui formular pe următorul exemplu:

https://biblioteca-digitala.ro / https://unibuc.ro

<HTML><HEAD><TITLE> UN EXEMPLU DE FORMULAR </TITLE></HEAD>
<BODY>
<H1> UN FORMULAR HTML </H1>
<FORM ACTION = „http://widget.com/cgi_bin/widgetorda" METHOD = POST>
Nume <INPUT NAME = „client" SIZE = 46 ><P>
Adresa <INPUT NAME = „adresa" SIZE = 40 ><P>
Oraş <INPUT NAME = „oraş" SIZE = 20 >
Ţara <INPUT NAME = „ţara" SIZE = 10 ><P>
Carte de credit #<INPUT NAME = „card" SIZE = 10 >
Expiră <INPUT NAME = „expiră" SIZE = 4>
M/C <INPUT NAME = „cc" TYPE = RADIO VALUE = „mastercard" >
Visa <INPUT NAME = „cc" TYPE = RADIO VALUE = „visacard" >
<P>
<INPUT TYPE = SUBMIT VALUE = „etichetă"><P>
Mulţumim că aţi ales banca noastră !
</FORM></BODY></HTML>
Pagina formatată (formularul) arată astfel:

UN FORMULAR HTML

Nume

Adresa

Oras Tara

Carte de credit # Expira M/C Visa

eticheta

Multumim ca ati ales banca noastra !

În acest formular sunt folosite 3 tipuri de casete şi anume:
1. Caseta ce urmează cuvântul **Nume** care are lungimea de 46 caractere. Şirul de caractere introdus de utilizator se înregistrează în variabila **client**.

https://biblioteca-digitala.ro / https://unibuc.ro

Marcajul <P> anunță navigatorul să afișeze ceea ce urmează pe o linie nouă. Cu el putem controla aspectul formularului pe ecran. Analog există o casetă de lungime 40 de caractere pentru adresa utilizatorului, informație ce va fi memorată în variabila **adresa**. De asemenea avem o linie în formular pe care se introduce orașul și țara, informații memorate în variabilele **oraș**, respectiv **țara**. Dacă cele două casete nu încap pe o linie se trece la linia următoare.

2. După casetele ce selectează numărul cărții de credit și data expirării acesteia urmează un alt tip de casetă numit **buton radio** deoarece el permite alegerea între mai multe alternative. Selecția și/sau deselecția unui astfel de buton se face printr-un click cu mouse-ul pe el sau cu tastatura. În cazul nostru alternativele sunt: M = mastercard și C = visacard. Parametrii VALUE arată ce buton radio a fost acționat. În funcție de opțiunea aleasă pentru cartea de credit, variabila **cc** va avea ca valoare șirul „mastercard" sau șirul „visacard".

3. Un al treilea tip de casetă este butonul SUBMIT, care după selecție, trimite calculatorului pe care se află formularul, toată informația introdusă de utilizator. Șirul parametrului VALUE este eticheta butonului, care se afișează.

Formularele se pot transmite folosind butonul SUBMIT, dând un click pe o hartă activă sau dând <CR> pentru un formular ce conține un singur câmp de tip TEXT.

O listă completă de marcaje HTML pentru formulare se găsește în Anexă.

1.10 Hărți de imagini [28, 78, 88]

Există două căi de creare a unei hărți de imagine: pe server sau la client.

Pentru definirea hărții de imagine pe server, codul HTML care include imaginea este împachetat într-un HREF ce indică un fișier MAP și marcajul <IMG> este extins pentru a include atributul important ISMAP. Fișierul MAP, conținut de serverul Web, specifică toate regiunile semnificative ale imaginii și le asociază unor anumite URL-uri. ISMAP indică faptul că imaginea inclusă este o hartă de imagine; execuția unui clic în diferite puncte ale imaginii determină rezultate diferite.

Formatul specific al fișierelor de tip map și chiar testarea hărților de imagini la server sunt dependente de tipul de server pe care se lucrează. Anumite servere cer indicarea unui fișier map oriunde în sistem, altele

https://biblioteca-digitala.ro / https://unibuc.ro

solicită ca aceste fişiere să se afle într-un director special de fişiere map cu acces paratjat, iar altele nu permit utilizarea fişierelor map. Pentru evitarea situaţiilor de acest gen, a apărut o nouă posibilitate, mai bună: hărţile imagine la client. Prin deplasarea prelucrărilor la client, se eliberează resursele serverului, creşte experienţa utilizatorului şi se obţine un control mai bun asupra hărţii în sine.

Pentru a defini o hartă de imagine la client, mai întâi se defineşte un set de regiuni pe imaginea respectivă, set de regiuni pe care le-am dori active. Marcajul care defineşte regiunea unei hărţi specificate este <MAP>, iar regiunile particulare pentru fiecare punct din cadrul imaginii sunt specificate cu <AREA>.

Atât în cazul hărţilor de imagini la server, cât şi în cazul celor la client, se poate alege dintr-o paletă largă de forme geometrice pentru regiuni. Prin definiţie, lista coordonatelor hărţii este citită numai până când este identificat punctul în care se execută clic, astfel încât două regiuni identice specificate în cadrul marcajului <MAP> nu determină un conflict.

1.11 Noi marcaje în HTML 4.0 [81, 88]

În specificaţia HTML 4.0 au fost adăugate trei marcaje noi: <Q> utilizat în cazul citatelor mici, care nu impun delimitator de paragraph, <INS> şi <DEL>, care permit marcarea anumitor modificări de editare efectuate în document. Desigur că nu se ştie cum va fi formatat textul, dar este de presupus că vor fi modificări ale atributelor de culoare, sublinieri sau alte formatări, atunci când textul va fi afişat utilizatorului. Aceste marcaje acceptă două atribute: CITE=url, care va indica un document ce explică de ce a fost modificată pagina curentă şi DATETIME=valoare, care specifică data şi ora exactă a modificării documentului.

Formatul argumentului DATETIME este următorul:

YYYY-MM-DDThh:mm:ssTZD

cu semnificatiile evidente ale componentelor (TZD este descriptorul orei locale).

Un marcaj important în HTML este marcajul <OBJECT>. Până acum marcajul <APPLET> era folosit pentru includerea programelor Java. În prezent marcajul <APPLET> este înlocuit de marcajul <OBJECT>, deoarece marcajul <OBJECT> asigură mai multe funcţii decât includerea microaplicaţiilor Java. Prin marcajul <OBJECT> pot fi

https://biblioteca-digitala.ro / https://unibuc.ro

incluse orice fel de informații caracteristice paginilor Web, de la imagini, programe până la alte pagini realizate în HTML.

Există un număr mare de atribute diferite care pot fi specificate în cadrul acestui marcaj, inclusiv toate atributele marcajului IMG (align, height, width, border, hspace, vspace, usemap, name) şi multe altele. Printre acestea este şi atributul CLASSID folosit sub forma:

<OBJECT CLASSID=unURL></OBJECT>

Există tipuri de informații pe care este posibil ca programele de navigare să nu le poată executa, vizualiza sau reda - de exemplu o imagine MPEG pe un sistem non-grafic. În aceste cazuri, în locul utilizării marcajului <ALT>, obiectele pot dispune de o valoare alternativă de tip text, specificată între marcajele <OBJECT> şi </OBJECT>.

Marcajul <OBJECT> şi valorile sale prestabilite a fost realizat încât dacă dispunem de o secvență de obiecte programul de navigare va trece de la cel exterior la cel interior, încercând să găsească ceva de afişat pe ecran, aşa cum se vede pe următorul exemplu:

```
<OBJECT TITLE="Satelitul pământului LUNA"
            CLASSID=unURL.py >
  <OBJECT DATA="Luna.mpeg"
            TYPE="apllication/mpeg">
    <OBJECT SRC="Luna.gif">
                  O imagine a satelitului LUNA
    </OBJECT>
  </OBJECT>
</OBJECT>
```

În acest caz, programul de navigare va încerca să execute programul Python (sufixul ".py") dacă este posibil; dacă nu este posibil, va încerca să afişeze un film MPEG şi dacă acesta nu este disponibil, va afişa imaginea "Luna.gif". Dacă nu poate realiza nici acest lucru, textul "**O imagine a satelitului LUNA**" este afişat pe ecran.

Există un marcaj HTML care permite specificarea parametrilor de execuție (run-time)pentru programul invocat. Se numeşte <PARAM> şi trebuie specificat în cadrul marcajului <OBJECT> în forma NAME= şi VALUE=.

De asemenea, se poate specifica direct tipul de cod pentru a fi siguri că numai programele de navigare care pot reda acel tip de informații le vor încărca. Aceasta se face sub forma:

```
<OBJECT CLASSID="java:program.start"
```

https://biblioteca-digitala.ro / https://unibuc.ro

CODETYPE="application/octet-stream">
</OBJECT>

Deoarece tipurile de cod sunt tipuri MIME, acest exemplu specifică un fişier executabil generic (un flux de octeţi – octet-stream). În exemplul precedent există un tip alternativ specificat ca "application/mpeg". În mod analog se specifică fişierele GIF, prin "application/gif" , cele JPEG prin "application/jpeg" şi aşa mai departe.

Pentru a inlocui un marcaj <IMG> cu unul <OBJECT> procedam ca in exemplul urmator:

<IMG SRC=desen.gif ALT="**Un desen**"
HEIGHT=100 WIDTH=100 HSPACE=5 VSPACE=5>

Exact aceeaşi imagine poate fi inclusă în pagină astfel:

<OBJECT DATA=desen.gif
TYPE="image/gif" HEIGHT=100 WIDTH=100
HSPACE=5 VSPACE=5>**Un desen**</OBJECT>

Cu ajutorul marcajului <OBJECT> se poate include efectiv un alt program HTML în propria pagină utilizând formatul:

<OBJECT DATA="header.html"></OBEJCT>

Se poate de asemenea include orice formular HTML din orice locaţie Web.

Există şi o altă modalitate de a include documente HTML în propria pagină Web. Includerile de fişiere pe server (SSI – Server-Side Includes) ne permit să dispunem de acestea ca parte componentă a paginii. În mod tipic, includerile de fişiere pe server se specifică sub formă de comentarii:

<!- -#include file="header.html"- ->

De multe ori, serverele solicită ca fişerele care dispun de material SSI să aibă un alt sufix, de exemplu ".shtml".

Pentru a include documente HTML în pagina Web este mai bine să utilizăm SSI decât marcajul <OBJECT>, dacă ambele se găsesc pe acelaşi server, dar, pentru a include materiale de la o altă locaţie Web, marcajul <OBJECT> este cea mai bună opţiune posibilă.

1.12 Pagini HTML dinamice şi JavaScript [32, 70, 84]

Combinarea tabelelor cu cadrele şi cu alte elemente HTML oferă o gamă largă de instrumente pentru realizarea de pagini WEB, dar toate sunt limitate la lucrul într-o singură dimensiune: totul se petrece pe o

https://biblioteca-digitala.ro / https://unibuc.ro

coală de hârtie. Marcajele permit obținerea unei dispuneri sofisticate a elementelor pe pagină, dar acele elemente nu pot fi plasate unele peste altele.

Cu JavaScript, paginile de casă devin mai inteligente, deoarece răspund la diferitele acțiuni efectuate cu mouse-ul, cu tastatura sau pentru că au fost prescrise o serie de acțiuni prin instrucțiunile adăugate în cadrul fișierului cu extensia *.html*, pe partea de client, cât și în privința scenariilor CGI, scrise de către administratorul Web, pe partea de server.

Scenariile CGI (*script languages*) sunt de fapt programe simple, scrise într-unul din limbajele care vor fi amintite mai jos, care sunt rulate la serverul Web.

Există un standard care reglementează felul în care aceste programe externe sunt folosite de catre servere. Acesta se numește **CGI** (**Common Gateway Interface**). Semnificatia celor 3 cuvinte este urmatoarea: cuvântul **common** spune că oricine, de la orice platformă, poate transfera informație care să fie prelucrată de un atare scenariu CGI, cuvântul **gateway**, definește o legătură și în același timp o poartă, o cale de acces între server, pe de o parte, și celelalte aplicații (clienți) și scenariu, pe de altă parte iar cuvântul **interface**, se referă de fapt la comunicarea cu omul folosind o interfață, un tipic, prin intermediul căruia utilizatorul interacționează cu informația afișată pe ecran de către aceste scenarii.

Dintre limbajele pentru scenarii amintim: shell-ul Bourne, shell-ul C (Unix), C/C++, perl (Practical Extraction and Report Language), Visual Basic, VBScript și, nu în ultimul rând, JavaScript.

https://biblioteca-digitala.ro / https://unibuc.ro

https://biblioteca-digitala.ro / https://unibuc.ro

CAPITOLUL 2. JavaScript

2.1 Introducere [70, 84]

JavaScript a adăugat o mai multă interactivitate şi funcţionalitate paginilor de casă. Până la apariţia sa, singurul mijloc îl ofereau unul din limbajele de tip script: Perl, oricare din cele trei shell-uri Unix (C-shell, Bourne-shell, Korn-shell) sau VBScript (Visual Basic Script). Toate limbajele de tip script (care se mai numesc şi "limbaje pentru scenarii") respectă o interfaţă comună, cunoscută sub prescurtarea CGI (Common Gateway Interface).

JavaScript este mai simplu decât Java, mai puţin dotat decât acesta, şi totuşi se bazează pe Java. Este mai apropiat de acesta decât de marcajele HTML. Dacă Java permite scrierea de aplicaţii de sine stătătoare (stand alone) sau "compilate", JavaScript nu permite acest lucru pe partea de client. JavaScript permite cunoscătorului HTML să trateze cu secvenţe de program adecvate, diverse evenimente iniţiate de către utilizator cum ar fi: efectuarea unui clic cu mouse-ul, "survolarea" cu mouse-ul a anumitor zone ale unei "hărţi" afişate, completarea unui formular sau acţionarea unui buton, conturat pe ecran; organizează ferestre în mai multe cadre, fiecare punctând spre un document din spaţiul Web. Într-un cuvânt, face răsfoirea unui volum mare de informaţii mult mai umană pentru utilizator.

Fiind pregătit pentru arhitectura client-server, pe ecranul unde rulează clientul poate apărea un formular cu felurite câmpuri, unde utilizatorul precizează diferite informaţii sau alege anumite opţiuni. La efectuarea unui clic pe butonul de tip *submit*, răspunsurile utilizatorului pentru acest formular sunt expediate la server, care le prelucrează.

JavaScript are o sintaxă asemănătoare limbajului C. Prima atenţionare care trebuie făcută este că, spre deosebire de HTML, de altfel ca şi C++, JavaScript este case sensitive, adică face distincţia între literele mari şi cele mici.

Instrucţiunile scrise cu JavaScript vor fi incluse în documentul HTML în interiorul marcajului <SCRIPT>. Acesta este un marcaj pereche şi are marcajul corespunzător de terminare </SCRIPT>.
Vom exemplifica folosirea acestui marcaj prin următorul document tip *html:*

```
<HTML>
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
<HEAD>
<TITLE>Document fără cod în JavaScript</TITLE>
</HEAD>

<BODY>
<P>Acesta este un text</P>
</BODY>
</HTML>
```

Să adăugăm la el un prim program scris în JavaScript.

```
<HTML>
<HEAD>
<TITLE>Document cod  JavaScript</TITLE>
<BODY>

<SCRIPT>function filtrezDenumire(txtElement) {
  if (txtEl. length != 5) {
        alert("Se introduce un cuvând din exact 5 litere")
 }
}
</SCRIPT>

<FORM>
<INPUT TYPE="text" NAME="denumire"
  onBlur="filtrezDenumire(denumire.value)"></P>
</FORM>
</BODY>
</HTML>
```

În al doilea exemplu se defineşte un formular cu un singur câmp de tip text. Apare însă un atom nou în marcajul <INPUT>, şi anume *onBlur="filtrezDenumire(denumire. value)". Dupa ce se* se termină de introdus informaţia în acest câmp şi este acţionată tasta ENTER, programul de vizitare client tratează această acţiune drept un eveniment. La apariţia evenimentului **onBlur,** adică la părăsirea acelei resurse, programul este pregătit să răspundă, deoarece a fost prevăzută o secvenţă de tratare pentru un astfel de caz. Este vorba despre funcţia **filtrezDenumire()** care constă dintr-un grup de acţiuni. Informaţia precizată de vizitatorul paginii de casă trebuie să fie filtrată prin verificarea numărului de caractere, care

https://biblioteca-digitala.ro / https://unibuc.ro

trebuie să fie exact 5. Programul focalizează câmpul şi cere să se introducă neapărat 5 caractere (litere, cifre, semne speciale). Dacă vizitatorul nu a tastat exact acest număr de caractere (exclusiv ENTER), funcţia îi va cere printr-un mesaj, să se conformeze întocmai.

În cadrul marcajului <SCRIPT>, apare "codul" acestei funcţii, adică programul scris în limbajul JavaScript. În prima linie se observă cuvântul **function apoi** denumirea funcţiei, **filtrezDenumire** şi, între paranteze rotunde, parametrii formali (aici este doar unul, şi anume **txtEl).** Urmeaza apoi corpul propriu-zis al funcţiei, adica acţiunile considerate necesar a fi prevăzute, când este prelucrat conţinutul câmpului **denumire.**

În linia a doua se remarcă instructţiunea if din javaScript:

```
if (txtEl.length != 5) {
        alert("Se introduce un cuvând din exact 5 litere")
}
```

Mesajul "Se introduce un cuvând din exact 5 litere" apare într-o căsuţă de dialog datorită instrucţiunii **alert("mesaj")**.

Mai sunt necesare câteva precizări: **denumire.value** este argumentul functiei **filtrezDenumire** care se găseşte în marcajul <INPUT>, adică acolo unde funcţia este apelată. Valoarea aceasta se transmite parametrului formal **txt.El**.

Perechea de marcaje <SCRIPT> ... </SCRIPT> poate figura şi în alt loc din cadrul unui document de tipul html asa cum se vede inrmătoarele două exemple .

Mai întâi scriem un program care să afişeze în fereastra documentului mesajul **Trecem la JavaScript! ,** secvenţă care trebuie plasată nu în corpul documentului,<BODY>, ci în antetul său, <HEAD>. Se obţine programul următor:

```
<HTML>
<!-- SCOP: marcajul script în antetul documentului html -->
<HEAD>
<TITLE>Exemplu</TITLE>
<SCRIPT>
document.write("Trecem la JavaScript!")
</SCRIPT>
</HEAD>
<BODY>
```

https://biblioteca-digitala.ro / https://unibuc.ro

<P>Se va observa mesajul Trecem la JavaScript ! transmis cu JavaScript</P>
</BODY>
</HTML>

Acum, cel care apare mai întâi este mesajul **Trecem la JavaScript** şi apoi, mai jos, în aceeaşi fereastră, se va observa mesajul **Trecem la JavaScript!** Secvenţa *&#033* produce semnul exclamării.

Prezentăm acum un exemplu din care se vede că programul scris cu Javascript poate apărea în cele două părţi ale documentului, în interiorul marcajelor <SCRIPT>. Exemplul de mai jos trasează o linie despărţitoare centrată pe 33%, urmată de un text format dintr-un antet de nivelul 1 "Exemplu" şi dintr-un paragraf ". Aici am scris cu JavaScript şi nu cu HTML".

Documentul care încapsulează programul JavaScript este următorul:

```
<HTML>
<!-- SCOP: marcaje script în cele două părţi ale unui document -->
<HEAD>
<TITLE>Exemplu </TITLE>
<SCRIPT>
function bareta() {
  document.write("<HR ALIGN='center'WIDTH=33%>")
}
function scriuCeva(nivel, antet, sir-normal){
document.write("<H" + nivel + ">" + antet + "</H" + nivel + "><P>"+
sir-normal)
}
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT>
document.write(bareta(), scriuCeva(l, "Exemplu", "Aici am scris cu
JavaScript şi nu cu HTML"))
</SCRIPT>
</BODY>
</HTML>
```

Partea principală a programului apare în <BODY>, în timp ce funcţiile apar în <HEAD>. Prima parte interpretată dintr-un document *html* este partea <HEAD>. Deci aici vor fi definite întotdeauna funcţiile.

https://biblioteca-digitala.ro / https://unibuc.ro

În exemplul de mai sus sunt definite două functii: **bareta()** şi **scriuCeva()**. Prima nu are nici un parametru, în timp ce a doua are trei parametri. Este de remarcat felul în care sunt anunţate acestea în definiţiile respective din partea de antet a documentului. Argumentul 1, primul din lista de apel, se pasează parametrului **nivel,** de asemenea aflat pe prima poziţie în lista definiţiei funcţiei.
Textele scrise între ghilimele: **"Exemplu"** şi respectiv **"Aici am scris cu JavaScript şi nu cu HTML",** vor fi preluate de către ceilalţi 2 parametri formali, **antet** şi respectiv şir **normal.**

În instrucţiunea care alcătuieşte corpul funcţiei **scriuCeva()** observam urmatoarele:.mai întâi vedem că unul din parametrii formali, **nivel,** apare de două ori. El poate apărea ori de câte ori este necesar. Semnul + acţionează aici pe post de operator de alipire, alăturare *(concatenare),* pentru a se forma în final textul *<Hl>Exemplu</Hl>,* corect din punctul de vedere al HTML. Elementele acestui mesaj se alipesc părticică cu părticică în cadrul instrucţiunii **document.write(mesaj)** din funcţia **scriuCeva()**. Aceste elemente se scriu obligatoriu între ghilimele.

Spaţiile dintre operatorul + şi aceste elemente nu sunt obligatorii. Dacă instrucţiunea nu încape pe un rând, se foloseşte ENTER şi se continuă pe rândul următor, indiferent de unde. Toate aceste caractere, cum sunt ENTER, blanc, TAB, definesc aşa-zisele spaţii albe *(white spaces),* ignorate de către interpretorul JavaScript.

Linia care alcătuieşte partea principală a programului foloseşte aceeaşi metodă **write**, deoarece în cadrul documentului ea este una dintre metodele acceptate în JavaScript de către acesta.

2.2 Structura programelor JavaScript [70, 77, 84]

Un program în JavaScript este format din: partea principală (main), din secvenţe de tratare a apariţiei anumitor evenimente şi din funcţii. Evenimentele sunt provocate de către mişcarea mouse-lui sau a tastelor, deci în genere, de către vizitator.
Partea principală este alcătuită din instrucţiuni ce apar în cadrul marcajului <SCRIPT> şi care nu intră în corpul unei functii. De fapt, ca şi în limbajul C/C++, main este funcţia principală, numai că acum lipseşte void sau int main() { ... }, cum se scria în C/C++. Este recomandabil a forma aceasta parte principală cât mai redusă posibil.

https://biblioteca-digitala.ro / https://unibuc.ro

În exemplul următor va fi făcută o singură modificare a exemplului precedent şi anume va fi rescrisă funcţia **main** din două instrucţiuni care, de fapt, sunt apeluri de funcţii.

```
<HTML>
<!--SCOP: alt mod de a scrie main-ul -->
<HEAD>
<TITLE>Exemplu</TITLE>
<SCRIPT>
function bareta() {
  document,write("<HR ALIGN='center'WIDTH=33%>")
}
function scriuCeva(nivel, antet, sir-normal)
  document.write("<H" + nivel + ">" + antet + "</H" + nivel +
"><P>" + sir-normal)
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT>
bareta() // aşez bareta în fereastră şi apoi chem funcţia scriuCeva
scriuCeva(l, "Exemplu", "Aici am scris aceste cuvinte cu JavaSeript
şi nu cu HTML.")
</SCRIPT>
</BODY>
</HTML>
```

Secvenţele de tratare a evenimentelor sunt "trezite" şi puse să răspundă la acţiunile pe care vizitatorul paginii de casă le întreprinde.

Evenimentul **onBlur,** prezent mai sus se produce când este părăsită resursa, deoarece se trece la o alta. La apăsarea tastei ENTER se părăseşte un câmp si drept urmare intră pe rol secvenţa de instrucţiuni prevăzută aici, care va examina continutul câmpului şi va proceda în consecinţă.

Documentele de tip htmi devin "inteligente", ca urmare a acestei filozofii de a răspunde la tot felul de acţiuni, efectuate de către vizitatorul paginii de casă.

Prezentăm acum un alt tip de eveniment, produs prin efectuarea unui clic pe orice buton al mouse-ului, care va cere vizitatorului să opteze pentru o anumită viteză:

```
<HTML>
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
<!-- SCOP: prezentarea evenimentului onClick-->
<HEAD>
 <TITLE>Exemplu</TITLE>
 <SCRIPT>
function v100(){
  alert("S.a ales viteza de 100 km/h!")
}
function v120()
  alert("S.a ales viteza de 120 km/h!")
}
 </SCRIPT>
 </HEAD>
<BODY>
 <FORM NAME="Selecție">
 <P>
 Viteza de <INPUT TYPE=radio NAME=viteza1
ONCLICK="v100()" VALUE=100>
 <P>
 Viteza de 120 km/h <INPUT TYPE=radio NAME=viteza2
VALUE=120 ONCLICK="v120()">
<P>
<INPUT TYPE=reset NAME=init VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

Aici partea principală a programului lipseşte. Apar însă două butoane radio, fiecare cu funcția sa. La selectarea uneia dintre viteze, se declanşează interpretarea funcției respective.

Fiecare marcaj <INPUT> este însoțit, de atributul **ONCLICK="nume-de-functie()".** Fiecare funcție are în corpul său doar instrucțiunea **alert. Alert(mesaj)** conduce la afişarea de către programul de vizitare, a unei ferestre care conține mesajul în cauză. Butonul **reset** din acelaşi formular intitulat *selecție* şi cele două butoane radio, servesc la eliminarea selecțiilor efectuate cu mouse-ul.

Funcțiile sunt scurte secvențe de program ce pot fi apelate de către alte funcții, deci şi de partea principală a programului, care poate fi socotită tot o funcție. Funcția are un nume care este precedat de cuvântul

https://biblioteca-digitala.ro / https://unibuc.ro

function. Dupa nume urmeaza, intre o pereche de paranteze rotunde, lista parametrilor formali, cărora li se atribuie în mod poziţional valorile reale ale argumentelor din lista de apel.
Putem atribui valori şi invers, dinspre funcţie spre secvenţa care a apelat-o. Exemplul următor exemplifica acest lucru:

```
<HTML>
<!-- SCOP: functie ce intoarce un rezultat -->
<!-- caculul recursiv al factorialului -->

<HEAD>
<TITLE>Exemplu</TITLE>
<SCRIPT>
function factorial(n)  {
  if ((n == 0) | | (n == 1))
              return 1
  else {
        fact = n*factorial(n-1)
        return fact
   }
}
</SCRIPT>
</HEAD>

<BODY>
 <SCRIPT>
 for( i = 0; i < 10; i ++)  {
   document.write( i +"!="+factorial(i))
   document.write("< BR>")
}
</SCRIPT>
</BODY>
</HTML>
```

Acest document de tip html conţine doar codul scris în JavaScript. Partea principală apelează functia **factorial(n)** care întoarce valoarea factorialului de n, *n* fiind parametrul de intrare. Din programul principal se comandă valori pentru *n* cuprinse între 0 şi 9 inclusiv. In linia:

document.write((i +"!="+factorial(i)*)*

https://biblioteca-digitala.ro / https://unibuc.ro

are loc o alipire între: valoarea variabilei **i** care controlează bucla for şi "!=", care se alipeşte în dreapta valorii i. În fine, în partea dreaptă a acestui sir de caractere apare valoarea factorialului respectiv.

2.3 Elementele de bază ale limbajului [70, 84]

Variabilele, comentariile şi constantele *se declara ca si in C/C++ cu urmatoarele nuantari:*

-- comentariile in JavaScript acceptă şi forma canonică a limbajului C de comentariu/* **text pe mai multe rânduri consecutive** */. , oferind o oarecare relaxare, în sensul că textul explicativ poate fi continuat pe oricâte linii de text ca in exemplul urmator:

/* Acesta este un comentariu în JavaScript care poate fi folosit şi pentru programele scrise cu C. */

-- datele scrise între ghilimele se numesc ***literali***. Un literal este aşadar scris ca un **"grup de cuvinte încadrat de ghilimele terminale".**

Literalii pot fi definiţi şi astfel: **'grup de cuvinte încadrat de apostroafe'.** Nu se admit mixturi de genul: "text". sau 'text". Pentru a include ghilimelele în cadrul textului, se recurge la caracterul special \ , deci se va scrie \". Apostroafele sunt admise între ghilimele, şi reciproc.

-- denumirile variabilelor nu vor fi admise să înceapă cu cifre sau semne speciale. Ele pot începe însă cu semnul "_" sau cu o literă alfabetică.

--fiecare variabilă are un domeniu al său unde este recunoscută. De exemplu, variabila *fact* din funcţia **factorial(n)** este recunoscută doar în cadrul acoladelor funcţiei, adică în corpul ei. Se spune că *fact* este o variabilă locală. În secvenţa de mai jos, o funcţie *f* este apelată din partea principală. La rându-i, *f* apelează *g*:

```
<head>var v=100, ... f (v,...) ... </head>
<body>function f (x,...){ var v = 200 ...g(v,...)...}
...function g(w,... .){...}</body>
```

Variabila locală *v* definită în **main** este redefinită local în corpul functiei *f*. Aici *v* va avea valoarea 200, în loc de 100, cât are în partea principală. Schimbarea valorii acestei variabile *v* din *f* nu va afecta variabila *v* din **main.** Cele două variabile *v* sunt **variabile locale**.

Există însă şi **variabile globale**. Pentru a declara o variabilă *v* globală, deci recunoscută în toate funcţiile acelui program, ar trebui tran-

https://biblioteca-digitala.ro / https://unibuc.ro

scrisă secventa de mai sus în felul următor:

```
<head>var v= 100 ... f (v,...) ... </head>
<body>function f(x,...){ g(x,...)...}...function g(y,...) { v = 25...
                                                        }</body>
```

Acum *v* este o variabilă globală. Singurul loc unde ea se modifică este abia în corpul funcției g.

-- considerarea unei variabile locale sau globale nu este mentionată explicit, ci rezultă din context. În primul rând ea va avea aceeaşi denumire, apoi trebuie să apară în mai multe locuri din program, adică să se lucreze eu ea.

-- deoarece JavaScript este un limbaj în care nu se pot declara (ca în C++/ C) tipurile variabilelor folosite, la momentul interpretării se stabileşte din context daca tipul datei coincide cu operatorul sau funcția care i se aplică asa cum se vede pe urmatoarea secventa:

```
var a = 100              // aici a este o variabila numerica
x=100+a                  // x va lua valoarea 200

a="scripturi"            // a poate fi redefinit ca variabila de tip sir
x="100" + a              // aici  se vor obtine 100 scripturi
```

În continuare este prezentat un mic program JavaScript, în care apare şi o parte de dialog. Este de fapt un formular în care vizitatorul va specifica parametrul razaCerc şi în alt câmp va primi răspunsul, adică lungimea sa. Urmează listingul:

```
<HTML>
<!-- SCOP: calcule interactve-->
<HEAD>
<TITLE>Exemplu</TITLE>
<SCRIPT>
//lungimea cercului este calculată cu formula obişnuită 2*Pi*r
 function calcul(obiect){
  obiect.LungimeCerc.value
  3.14*  obiect.razaCerc.value*  2
  }
</SCRIPT>
<HEAD>
<BODY>
<FORM NAME="calculCerc">
Precizati raza cercului <INPUT TYPE=text NAME=razaCerc SIZE=25>
```

https://biblioteca-digitala.ro / https://unibuc.ro

```
<P>
Lungimea este <INPUT TYPE=text NAME=LungimeCerc SIZE=30>
<P>
<INPUT TYPE=button NAME=buton VALUE="Declansati calcu-
lul" ONCLICK="calcul(this.form)">
</FORM>
</BODY>
</HTML>
```

După ce a fost precizată valoarea razei cercului, când se efectuează un clic pe butonul "Declanşaţi calculul", în caseta de dedesubt cu denumirea internă LungimeCerc *se va anunţa valoarea calculată.*

-- JavaScript foloseşte si tipul de date logic. Valorile literalilor booleeni sunt: "false" (valoare falsa) şi "true" (adevărată).

-- dintre constantele speciale folosite în JavaScript menţionăm: secvenţele care încep cu bara inversă denumite şi *secvenţe escape:* **\b** (înapoi un caracter, *backspace),* **\f** (avans la linia nouă, *line feed),* **\n** (o linie nouă, *new line),* **\r** (întoarce carul, *carriage return),* **\t** (caracterul de tabulare orizontală, *tab)* şi \" (inserează ghilimele în şirul literal).

Expresiile şi operatorii sunt ca si in C++ cu urmatoarele nuantari:

-- se pot efectua şi atriburi multiple de forma: $v=v_1=...=v_n$**=expresie,** unde $v_1,...,v_n$ sunt variabile in care principiul de executare este evident. Atribuirea este un operator care figurează într-o expresie. Ca orice expresie, aceasta are o valoare rezultată în urma evaluării. În cazul atribuirilor, valoarea rezultată este valoarea atribuită variabilei care este prima din stânga într-un şir de atribuiri.

-- operatorii relaţionali sunt operatori binari, care întorc o valoare logică, true dacă relaţia este adevărată, sau false, dacă relaţia nu este adevărată.

-- pentru conversii forţate *şir -> număr* se folosesc două funcţii: **parseInt()** şi **parseFloat()**. Fiecare va converti un şir într-un număr, prima, într-unul de tip întreg, a doua, într-unul de tip real, fireşte, dacă acest lucru este posibil. Secvenţa de mai jos exemplifică acest lucru:

```
var a = "1000"                    //definim literalul 1000
b = parseInt(a)          //b va avea valoarea întreagă 1000
var a = "Java"
b = parseInt(a)            //se va solda cu mesaj de eroare
var a = "123.45"
```

https://biblioteca-digitala.ro / https://unibuc.ro

b = parseFloat(a) //b va avea valoarea numerică 123,45

-- datele octale, adică cele scrise în baza sistemului de numerație 8, vor începe obligatoriu cu cifra 0. Deci, numărul 0123 va fi considerat numărul 123 în baza 8 şi nu în baza 10.De fapt, valoarea zecimală a lui 0123 va fi $1x8^2 + 2x\ 8^1 + 3 = 64 + 16 = 83$.

-- in JavaScript se poate amâna interpretarea unei expresii, până când vizitatorul va specifica expresia şi apoi va apela la serviciile funcției **eval()** pentru a calcula cât este, aşa cum rezultă din următorul program amplasat în contextul unui document html:

```
<HTML>
<!-- SCOP: specificarea dinamică a expresiilor unui vizitator -->
<HEAD>
<TITLE>Exemplu</TITLE>
<SCRIPT>
 // eval preia şirul de valori din câmpul expresie şi după o palpare
// a sa dă rezulatul în câmpul soluție
function calcul(obiect) {
obiect. soluție. value = eval(obiect.expresie.value)
}
</SCRIPT>
</HEAD>

<BODY>
<FORM NAME="evalDinamic">

Furnizați expresia prin valori numerice <INPUT TYPE=text NAME=expresie SIZE=40>
<P>
Rezultatul este <INPUT TYPE=text NAME=soluție SIZE=20
<P>
<INPUT TYPE=button NAME=buton VALUE="Declanşați calculul" ONCLICK="calcul(this. form)">
</FORM>
</BODY>

 </HTML>
```

https://biblioteca-digitala.ro / https://unibuc.ro

-- Limbajul JavaScript nu operează cu tipuri, dar operează cu obiecte. Dacă am avea următoarea instrucțiunea de atribuire: **i = 75 + "five",** analizorul JavaScript decelează semnul lui *two,* îi aplică implicit o conversie coercitivă din șir în număr, obținând 5, îl adună la valoarea 75 și variabila *i* va avea valoarea 80. Se spune despre o astfel de construcție că este una de **coerciție**, dar astfel de trucuri nu sunt recomandate. Limbajul nu are cum decela decât la momentul interpretării cu cine are de-a face. Regula de coerciție se aplică forțat de context.

Nici regula de tip *casting* nu se vede în JavaScript în mod explicit, ca în C. Scriind *parselnt("123")* se forțează o construcție de tip **cast.** Dar dacă acest șir este format undeva în mod dinamic într-o variabilă șir, după care se aplică funcția **parselnt,** pot apărea erori care nu se depanează ușor.

Instrucțiunile limbajului JavaScript sunt cele de la C++ având aceiași sintaxă și semantică .

2.4 Obiectele limbajului JavaScript [32, 70, 77]

Obiectul are o serie de caracteristici (proprietăți) si o serie de funcții. Există și obiecte al căror conținut nu se schimbă numite obiecte statice. Mai există și obiecte create de programator, care, ca și unele dintre obiectele JavaScript, trebuie instanțiate.

JavaScript este un limbaj "bazat" și nu "orientat" pe obiecte, așa cum este C++ care face parte din generația a 4-a (4GL). În programarea orientată pe obiect întâlnim o mulțime de limbaje, unele orientate pe clase, altele, numai pe obiecte, altele, pe ambele. Conceptul, destul de vechi a apărut prin 1967 la Palo Alto, în cadrul companiei XEROX, prin limbajul Smaltalk

Programarea orientată pe obiect a coincis cu perioada limbajelor din generatia a patra. Ea a adus cu sine o economie de timp pentru rescrierea codului, prin reutilizarea codurilor scrise, conducând în final la creșterea eficienței muncii de concepere. Dacă în perioada limbajelor 3GL, FORTRAN-ul alături de COBOL și PL/I alcătuiau fondul de limbaje foarte populare, în perioada 4GL întâlnim: C++, CLOS (*Common LISP Object System),* Object Pascal, Eiffel (un fel de Pascal foarte reușit); apoi treptat trec *pe* obiecte și Cobol-ul, și Basic-ul *(Microsoft Visual Basic* sau *Microsoft VB for Applications,* un dialect VB folosit în MS Excel)

https://biblioteca-digitala.ro / https://unibuc.ro

etc. Nu putem încheia enumerarea fără Java, primul limbaj orientat pe obiecte pentru mediul de rețea.

Un *obiect* apare ca urmare a procesului de instanțiere a unui șablon de date, numit **clasă**. Acest șablon poate fi considerat drept un sertar, în care se găsesc o serie de despărțituri - unele pentru păstrarea datelor, altele pentru diverse funcții folosite în tratarea acestor date.

În programarea orientată pe obiect (OOP - Object Oriented Programming) aceste funcții se numesc **metode**. De pildă, autobuzul este un obiect, avionul este altul, autoturismul este altul, etc.. Toate derivă din clasa vehicule, care la rândul lor derivă din clasa mijloace. Există mijloace care nu sunt deplasabile pe teren precum racheta și altele deplasabile pe pământ precum cele de mai sus. Se observă deci o ierarhie a obiectelor. Putem considera șablonul, de exemplu al unui avion, drept proiectul său de construcție. Se spune despre un obiect că posedă o serie de proprietăți, așa cum ar fi autoturismul care are o anumită greutate, un anumit număr de cai putere, dezvoltă o anumită viteză, etc..

Obiectele conțin o serie de funcții (metode), de asemenea încapsulate. **Exemplu** obiectul *Math* din JavaScript al carui conținut se referă, la anumite valori caracteristice, (cărora li se spune proprietăți) sau funcții frecvent folosite. Dintre proprietățile sale enumerăm: *Math.E* - valoarea numărului e, *Math.ln2- v*aloarea logaritmului neperian al cifrei 2, (adică 0,693), *Math.ln10*-valoarea pentru ln10, (deci 2,302); *Math.PI* (π=3.14); etc. Dar *Math* conține și o serie de metode pentru calculul funcțiilor trigonometice directe precum: *Math.sin, Math.cos, Math.tan;* sau inverse precum: *Math.asin, Math.acos, Math.atan, Math.atan2*; pentru rotunjirea valorilor: *Math.ceil* (de exemplu ceil(123,67) va da 124) sau *Math.floor* (de exemplu, floor(123,67) va da 123). Mai întâlnim: *Math.log, Math.min(v1,v2), Math.max(v1,v2), Math.random* pentru generarea numerelor aleatoare cu distribuția uniformă. Există și *Math.round* (de exemplu, round(66,45) dă 66, dar round(66, 51) va da 67) etc.

Se observă că proprietățile ca si metodele sunt unite prin intermediul operatorului punct (.) de clasa respectivă. În limbajul JavaScript se preferă denumirea de obiect în locul celei de clasă.

0 definiție a obiectului, ar putea fi următoarea: obiectul este o entitate, o structură abstractă de date, constituită din proprietăți și metode. Proprietățile se pot asimila datelor (constante sau variabile), iar metodele, funcțiilor.

Spunem că JavaScript nu este decât bazat pe obiecte, nu și orien-

https://biblioteca-digitala.ro / https://unibuc.ro

tat spre ele pentru că obiectelor acestui limbaj le lipsesc 3 caracteristici ale teoriei OOP *(Object Oriented Programming)*: moştenirea, încapsularea şi abstractizarea.

Corporaţia Netscape Communication a definit propriul concept, cunoscut sub numele **JavaScript Object Model** (JOM). JOM conţine totalitatea obiectelor, proprietăţilor, metodelor dar şi variabilelor şi funcţiilor din universul *Netscape*. Obiectele Netscape nu trebuie confundate cu cele ale limbajului JavaScript. Aceeaşi corporaţie a lansat programul de navigare Netscape Navigator, pentru partea client. Tot ea a creat şi o serie de versiuni pentru partea de server. Este vorba despre: Netscape Entreprise System (platforma Unix), Netscape LiveWire (platforma Windows/NT) sau Netscape Server System 3.0. Fiecare dintre aceste categorii de programe operează cu o serie de obiecte specifice.

In cele ce urmează numim partea *client*, pe scurt *Navigator*, şi partea *server*, pe scurt *LiveWire*.

Pe partea *Navigator* se pot folosi o serie de obiecte conţinute în însuşi acest *JavaScript Object Model*, dar se pot defini şi altele proprii. Pe osatura unui document *HTML* se folosesc facilităţile limbajului JavaScript şi se pot afişa o serie de ferestre, cadre, casete de dialog, unde să apară tot felul de mesaje, resurse grafice. Se pot valida sau evalua expresii în care apar şi variabile, nu numai date numerice sau literali, se pot chiar prelucra date sosite de la LiveWire.

Pe partea *LiveWire*, în JavaScript există posibilitatea de a scrie scenarii în care să fie procesate toate solicitările sosite de la diferiţi clienti (diferite programe de vizitare care recurg la JavaScript), care trimit informaţii cerute prin aceste scenarii, sau de a accesa server-e de baze de date, pentru fişiere sau care conţin biblioteci de aplicaţii scrise în C, Java etc. Se pot chiar adapta diverse aplicaţii de tip *applet* scrise în limbajul Java.

2.5 Definirea şi folosirea propriilor obiecte în JavaScript[70]

JavaScript dispune de o serie de obiecte ale sale care organizează o serie de date, dar şi funcţii (metode) ce operează asupra acestor date. Aceste obiecte - unele deja existente în JavaScript, altele create de utilizator - pot fi cuplate, între ele sau legate de elementele *HTML* dând posibilitatea efectuării diverselor calcule matematice sau creării diferitelor resurse: text, ferestre, formulare, cadre etc. Sintaxa generală pentru

https://biblioteca-digitala.ro / https://unibuc.ro

folosirea unei anumite proprietăți a unui obiect este:

numeObiect.numeProprietate.

In JavaScript un obiect propriu se crează parecurgând două *etape:* definirea obiectului şi crearea(instanțierea) obiectului.

Exemplu. Definim obiectul Autoturism cu funcția aoto() care are 3 parametri: tip, viteză, an fabricație, prin secvența:

```
function auto (stop, iViteza, iAn)
{this.Tip=sTip;
this.Viteza=iViteza;
this.AnFab=iAn;
}
```

La definirea obiectului se poate recurge, pentru denumiri la câte o literă de început care anunță tipul variabilei: s - şir, i - număr întreg. Cuvântul rezervat **this.** este un operator cu care se face referire la obiectul curent.

Instanțierea obiectelor în JavaScript se face prin operatorul new (întocmai ca în C++).

Pentru obiectul Autoturism definit mai sus putem acum instanția oricate obiecte de acest tip, sub următoarea formă:

```
AutoturismA=new auto("Cielo",180, 1995)
AutoturismB=new auto("Dacia", 120, 2003)
AutoturismC=new auto("Ford", 200, 2000)
```

Putem crea oricâte astfel de obiecte. Instrucțiunile de atribuire de mai sus seamănă cu cele ale unui constructor din C++. Operatorul new va aloca resursele (de memorie) necesare pentru întreaga structură de date a obiectului.

Adăugăm şi o metodă obiectului Autoturism care calculează numărul de km parcurşi de acesta. Parametrul intern al obiectului afectat de această funcție în sensul creşterii sale este Lungime . Rescriem funcția **auto(),** ca să anunțăm şi metoda **Parcurs** ca membră a acestui obiect:

```
function auto(sTip, iViteza, iAn)
{
 this.Tip                = sTip,.
 this.Viteza             = iViteza;
 this. AnFab             = iAn:
 this.Lungime            = 0;
 this.Parcurs            = Parcurs:
}
```

https://biblioteca-digitala.ro / https://unibuc.ro

Codul metodei **Parcurs** ar putea fi de forma:

```
function Parcurs(iKm) {
this.Lungime          =this.Lungime +i Km;
```

2.6 Obiecte specifice limbajului JavaScript [70, 77, 84]

Obiectui Date este destinat manevrării datelor calendaristice dar şi orei, pentru afişarea în cadrul paginii de casă. Acest obiect nu posedă proprietăţi, în schimb are vreo 20 de metode, şi nu se pot prelucra date anterioare anului 1970.

Un obiect *Date* se creează astfel: **dataDeAzi = new Date;**

Metodele sunt apelate prin constructia:

nume-obiect-tip-Date.nume-metoda.

Obiectele statice sunt cele care nu se schimba. **Math** este un astfel de obiect, deoarece el nu se schimbă, cum se întâmplă în cazul obiectului *Date.* Orice obiect ce apartine lui *Math* nu va fi static şi deci va trebui creat. *Obiectele statice nu trebuie instanţiate.* Astfel prin:

```
var numarAleator = Math.random()
```

am atribuit variabilei numarAleator o valoare aleatoare generată cu metoda random a obiectului *Math.* Algoritmul ascuns în random asigură generarea unui număr aleator cu distribuţia uniformă în mulţimea [0,1). Dacă dorim să generăm numere aleatoare între doua limite, *nl* şi *n2,* vom scrie:

```
var numarAleator = n1 + (n2-n1)* Math.random()
```

Orice proprietate a unui obiect static poate fi accesată prin construcţia: **obiect-static. proprietate**

Scopul (adica domeniul unde denumirea unui obiect este recunoscută) unui obiect diferă în funcţie de aplicaţie. Pentru partea Navigator, obiectul este recunoscut în pagina curentă de casă, în care a existat scenariul cu obiectul respectiv. În contextul LiveWire, un obiect poate fi recunoscut şi în alte pagini de casă, nu numai în pagina curentă.

Obiectul string este de asemenea un obiect static si se referă la şiruri de caractere ce pot fi afişate cu diferite caracteristici tipografice. Singura proprietate a acestui obiect este numită **length,** adică lungime, şi cuprinde numărul caracterelor (litere, cifre, semne speciale) pe care le conţine obiectul la un moment dat. Metodele obiectului string şi semnificaţiile lor sunt [70]:

https://biblioteca-digitala.ro / https://unibuc.ro

anchor()	Va crea un text pe post de ancoră
big()	Şirul va fi afişat cu un corp de literă mai mare
blink()	Şirul căruia i se aplică va pulsa
bold()	Şirul va apărea cu caractere îngroşate
charAt()	Preia un caracter din şir care fusese adresat prin intermediul unui index
fixed()	Are efectul similar cu al marcajului <TT>
fontsize()	Stabileşte mărimea corpului de literă cu care va fi afişat pe ecran şirul respectiv
indexOf()	Preia valoarea offset-ului acelui caracter care este căutat într-un şir, plecând de la valoarea fromIndex
italics()	Similar marcajului <I>
lastIndexOf()	Se caută ultima apariţie a unui caracter într-un şir. Valoarea acelui index va fi conţinută de acest parametru
link()	Creează o legătură de tip hipertext. Similar cu <A HREF...> spre o resursă de tip URL
small()	Afişează şirul cu corp mic de literă
strike()	Afişează şirul barat cu o liniuţă
sub()	Şirul va fi afişat ca indice inferior
substring()	Preia un şir parţial dintr-un şir.Parametrii ceruţi de această metodă sunt indexA şi indexB. Ambii fixează limitele acelui şir redus
sup()	Afişează acel şir ca indice superior
toLowerCase()	Converteşte literele alfabetice mari ale unui şir în litere mici
toUpperCase()	Converteşte literele alfabetice mici ale unui şir în litere mari

Obiectul numit **masiv** *(array)* este un grup de doua sau mai multe *elemente omogene*. Prin intermediul unui indice, se poate face referire la unul din aceste elemente. Ca si in C++ in JavaScript se consideră că indicele porneşte de la 0. Spre deosebire de C/C++ sau Java, JavaScript nu obligă specificarea tipului datelor.

Exemplu. Fie obiectul *elev* cu următoarele proprietăţi: nume, adresă, vârsta, sex, clasă, şcoală, boli. Ultimul parametru nu este un şir de literali,

https://biblioteca-digitala.ro / https://unibuc.ro

pentru că aceasta ar ocupa foarte mult, ci un nume care ascunde în el ce boli a făcut elevul (1=pojar, 2=scarlatina, 4=oreion, 8=rujeola, 16=vărsat de vânt). Deci un cod 15 ar însemna că elevul a contractat în copilărie primele 4 boli; un cod=23, echivalează cu pojar+scarlatina+oreion+vărsat de vânt etc. Funcția de tip *constructor* a acestui obiect ar fi definită astfel:

```
function elev(sNume, sAdresa, iVarsta, iSex, sClasa, sScoala, sBoli
        this.nume       = sNume;
        this.adresa     = sAdresa;
        this.varsta     = iVarsta;
        this.sex        = iSex;
        this.Clasa      = sClasa:
        this.Scoala     = sScoala;
        this. Boli      = sBoli;
```

Fiecare proprietate poate fi inițializată cu valori referitoare la diverși subiecți. Ca să ne referim la una dintre persoane și, în cadrul ei, la una dintre proprietăți, vom crea două obiecte de tip Elev prin secvența următoare:

```
for (i=0; i<2; i++)
{this(i)=newElev()
this(0)(0)="Ion Bianca"
this(0)(1)="Str. Azurului nr 13"
this(0)(2)=17
this(0)(3)="f"
this(0)(4)="X"
this(0)(5)="Liceul Odobescu"
this(1)(0)="Marin Ion"
this(1)(1)="Str. Lujerului nr. 5"
this(1)(2)=19
this(1)(3)="m"
this(0))(4)="XII"
this(0)(5)="Liceul Basarab"
}
```

Putem accesa proprietățile unui obiect creat sub forma:

```
document.write("Varsta lui "+this(0)(0)+"este"+this(0)(2)+"<BR>")
```

Acest mod de programare este destul de greoi deoarece nu apare deloc numele explicit al obiectului. Se folosește operatorul this atât pentru a referi obiectul cât și pentru a accesa diverse proprietăți ale acestuia, obiectul apărând doar ca o listă de elemente.

https://biblioteca-digitala.ro / https://unibuc.ro

Se remarcă totuşi că limbajul JavaScript admite şi maxime multidimensionale în care primul indice se referă la obiect iar al doilea la proprietăţile acestuia.

2.7 Funcţii proprii ale lui JavaScript [70, 32, 84]

Funcţia eval() are formatul **eval(şir)**.

Scopul funcţiei **eval()** este să analizeze corectitudinea sintactică a unui şir. Şirul poate fi orice expresie în care pot apărea şi denumiri de variabile din acel program, care au fost analizate deja; se admit şi denumiri de proprietăţi ale obiectelor. Funcţia nu serveşte doar la analize sintactice ci şi la calcularea valorii expresiilor numerice . Şirul poate fi format din una sau mai multe instrucţiuni.

Exemple:

- alert(eval("13+20")); afişează 33;
- a=7;
  5*eval(3*++a);
  alert(a); afişează 120;
- alert(eval("2"+"5")); afişează 25;

Funcţia parseInt() are formatul : **parseInt(sir, baza-de-numeratie)**

Şirul *şir* trebuie să fie format doar din cifre, ale sistemului zecimal, octal sau hexazecimal. El nu trebuie să aibă parte fractionară. Dacă aceasta din urmă există totuşi, ea va fi trunchiată. În cazul în care faza de *parsing* eşuează va fi anunţat un cod de eroare care este 0 pentru platforma Windows şi NaN pentru platforma Unix. Aceasta se întâmplă şi când primul caracter din şir nu este conform contextului.

Funcţia parseFloat() are formatul **parseFloat(sir, baza-de-numeratie)**

Se admit, în afara numerelor cu parte fracţionară (exemplu 10.234 sau 135. sau .135) şi numere scrise după notaţia *stiinţifică*, adică cu exponent (exemplu: -12.45e-24). Este valabilă şi aici observaţia referitoare la codul de eroare.

Funcţia escape() are formatul **escape (sir)**

Are rostul de a converti un şir de caractere ASCII în coduri admise de *ISO.Latin-1*.

De multe ori, într-o adresă de tip URL trebuie transmis un caracter de tabulare, tab (fie un / sau \ etc.). Nu există altă modalitate decât folosirea acestei funcţii, dinspre limbajul JavaScript.

https://biblioteca-digitala.ro / https://unibuc.ro

Funcția unescape() are formatul **unescape (sir)**
Este inversă funcției anterioare. Va converti un șir de caractere *ISO-Latin-1* în coduri ASCII.
Funcția number are formatul **number (sir)**
Rolul ei este de a converti șirul în valoare numerică. În cazul în care șirul nu se poate converti către o valoare numerică, returnează constanta de eroare NaN (Not a Number).
Exemple.

alert(number("13")+number("5.2")); afișează 18.2
alert(number("2x")); afișează NaN

Funcția isNaN
Returnează **true** dacă șirul nu poate fi convertit către o valoare numerică și **false** în caz contrar.
Exemple.

alert(isNaN("23")); returnează false
alert(isNaN("23a")); afișează true.

2.8 Obiectele specifice părții client [70, 81, 84]

2.8.1 Obiectul window

Browser-ul afișează informațiile cu ajutorul ferestrelor. El poate afișa mai multe ferestre simultan. Obiectul **window** este de nivelul cel mai înalt și are rolul de a permite lucrul cu ferestrele. Datorită faptului că acest obiect are nivelul cel mai înalt, nu mai este necesar să fie folosit **window** înaintea metodelor sau proprietăților obiectului. Instanța unui obiect **window** este o fereastră afișată de browser. Ca orice obiect, **window** are anumite metode și proprietăți.

2.8.1.1 Metode

1. Metoda **open(fișier_html, nume, parametri)** are rolul de a deschide o fereastră. Parametrii sunt:

- fișier_html – adresa URL a fișerului html care este încărcat de fereastră;
- nume – numele ferstrei;
- parametri – mai mulți parametri care au rolul de a specifica modul în care va fi afișată fereastra pe ecran. Aceștia sunt:
  - **top** – distanța în pixeli față de partea de sus a ecranului la care va fi afișată fereastra;

https://biblioteca-digitala.ro / https://unibuc.ro

- **left** - distanța în pixeli față de partea din dreapta ecranului la care va fi afișată fereastra;
- **width** – lățimea ferestrei în pixeli;
- **height** – înălțimea ferestrei în pixeli
- **resizable** – poate lua una din valorile **yes** sau **no**. Dacă ia valoarea **yes**, utilizatorul poate modifica dimensiunile ferestrei;
- **menubar** - poate lua una din valorile **yes** sau **no**. Dacă ia valoarea **yes**, fereastra afișează meniul browser-ului;
- **toolbar** - poate lua una din valorile **yes** sau **no**. Dacă ia valoarea **yes**, fereastra afișează bara cu instrumente a browser-ului;
- **status** - poate lua una din valorile **yes** sau **no**. Dacă ia valoarea **yes**, fereastra afișează bara de stare;
- **location** - poate lua una din valorile **yes** sau **no**. Dacă ia valoarea **yes**, fereastra afișează edit-ul care conține adresa fișierului html;
- **copyhistory** - poate lua una din valorile **yes** sau **no**. Dacă ia valoarea **yes**, copiază istoria apelurilor ferestrei părinte.

2. Metoda **close()** are ca efect închiderea ferestrei curente.

În exemplul următor, la apăsarea unui buton se deschide o fereastră, iar la apăsarea altui buton fereastra se închide:

```
<HTML>
<!-- SCOP: metodele open şi close-->
<HEAD>
<TITLE>Exemplu</TITLE>
</HEAD>
<SCRIPT>
fereastra=0;
function deschid() {
 fereastra=window.open("C:dw.html","fer") }
function inchid() {
 fereastra.close() }
</SCRIPT>
<BODY>
<INPUT type="button" value="Deschide" onclick="deschid()">
<INPUT type="button" value="Inchide" onclick="inchid()">
</BODY>
</HTML>
```

https://biblioteca-digitala.ro / https://unibuc.ro

Dacă numele ferestrei nu este precizat, metoda close() închide fereastra curentă. Totuşi, înainte de a închide efectiv ferastra, va apărea o casetă de dialog prin care se solicită utilizatorului să confirme dacă vrea să închidă fereastra.

3. Metoda **focus()** are rolul de a activa o ferastră. O fereastră activă este vizibilă pe ecran şi recepţionează evenimentele de la mouse (eventual tastatură). O fereastră poate fi activată printr-un click asupra ei, dar şi prin metoda focus().

4. Metoda **onblur()** are rolul de a dezactiva o ferastră. O fereastră inactivă poate fi aşezată pe bara de task-uri, dar poate fi şi pe ecran, caz în care poate fi acoperită de suprafaţa altei ferestre. O ferastră poate fi dezactivată dacă se execută click pe suprafaţa altei ferestre, dar şi cu ajutorul metodei onblur(), caz în care ea va fi prezentă doar pe bara de task-uri.

2.8.1.2 Proprietăţi

O proprietate deosebit de utilă este **status**. Valoarea reţinută de această proprietate va fi afişată pe bara de stare.

Alte proprietăţi importante sunt: **document**, **frame**, **history**, **location**. Acestea sunt, la rândul lor obiecte.

Obiectul document

Este de mare complexitate şi conţine mai multe proprietăţi şi metode. Printre acestea avem proprietatea de tip obiect **all**. Ea este un vector al tuturor elementelor care se găsesc pe pagină. Microsoft numeşte o astfel de proprietate Colecţie, Fiecare element este tratat ca obiect. Printre elementele (obiectele) care fac parte din vector găsim: **HTML**, **HEAD**, **BODY**, **TITLE** (chiar dacă nu este trecut tag-ul respectiv), **STYLE**, **SCRIPT**, câte un element pentru fiecare paragraf(**P**), imagine(**IMG**), etc. Fiecare element (obiect) conţinut are o anumită proprietate, numită **tagName**, care reţine tipul. La rândul ei **all** are mai multe proprietăţi şi metode, printre care:

- **length** – numărul de componente ale vectorului;
- metoda **tags(tip)** – returnează un vector al elementelor de un anumit tip. Fiind vector, are proprietatea **length**, care permite adresarea tuturor elementelor;
- metoda **item(id)** – returnează elementul (obiectul) de **id** dat;
- **anchors** – vector ce reţine elementele de tip ancoră (**A**);
- **images** – vector care reţine toate elementele de tip imagine (**IMG**);

https://biblioteca-digitala.ro / https://unibuc.ro

- **bgColor** – proprietate care reține culoarea de fond a documentului;
- **fgColor** – proprietate care reține culoarea font-urilor;
- **linkColor** – proprietate care reține culoarea de scriere a legăturilor;
- metoda **write()** – se folosește pentru scrierea integrală a unui fișier HTML
- etc.

https://biblioteca-digitala.ro / https://unibuc.ro

ANEXA [70, 84] Marcajele limbajului HTML

Marcaje de bază		
Marcaj HTML	**Marcaj de închidere**	**Semnificație**
<ADDRESS>	</ADDRESS>	Informații despre adresă și despre autor
<BODY>	</BODY>	Corpul paginii HTML
 		Linie nouă
<HEAD>	</HEAD>	Informație de formatare HTML
<H*n*>	</H*n*>	Nivelul de subtitlu al documentului (*n*= 1-6)
<HR>		Riglă orizontală
<HTML>	</HTML>	Definește un fișier în format Web
<P>		Paragraf nou
<PRE>	</PRE>	Informație preformatată

Marcaje logice		
Marcaj HTML	**Marcaj de închidere**	**Semnificație**
<CITE>	</CITE>	Citare bibliografică
<CODE>	</CODE>	Listing de program
<DFN>	</DFN>	Definiție de cuvânt
<KBD>	</KBD>	Text de la tastatură
<SAMP>	</SAMP>	Exemplu de date introduse de utilizator
<VAR>	</VAR>	Program sau variabilă

Marcaje de formatare a textelor		
Marcaj HTML	**Marcaj de închidere**	**Semnificație**
<B>	</B>	Afișează text îngroșat
<I>	</I>	Afișează text înclinat
<U>	</U>	Afișează text subliniat
<TT>	</TT>	Text cu font monospațiu
<CITE>	</CITE>	Citare bibliografică
<CODE>	</CODE>	Listing de program
<DFN>	</DFN>	Definiție de cuvânt
<EM>	</EM>	Stil logic de evidențiere

https://biblioteca-digitala.ro / https://unibuc.ro

<KBD>	</KBD>	Text de la tastatură
<SAMP>	</SAMP>	Exemplu de date introduse de utilizator
<STRONG>	</STRONG>	Evidențiere logică puternică
<VAR>	</VAR>	Program sau variabilă
<BASEFONT		Specifică dimensiunea implicită a fontului din pagină
<FONT>	</FONT>	Specifică atribute ale textului încadrat
SIZE=n		Dimensiunea textului este 1:7, cu 7 cea mai mare
FACE="a,b"		Specifică fontul: a, dacă este disponibil, sau b
COLOR=s		Culoarea textului: fie un nume de culoare, fie o valoare RGB

Marcaje pentru liste		
Marcaj HTML	**Marcaj de închidere**	**Semnificație**
<DD>		Descriere de definiție
<DL>	</DL>	Listă de tip definiție
<DT>		Termen de definiție
<LI>		Element de listă
<OL>	</OL>	Listă ordonată (numerotată)
TYPE=tip		Tipul numerotării. Valori posibile: A, a, I, I, 1
START=x		Numărul de început al listei numerotate
<UL>	</UL>	Listă neordonată (marcată)
TYPE=forma		Forma marcajului. Valori posibile: circle, square, disc
<!--	-->	Comentariu în codul HTML

Marcaje pentru referinte		
Marcaj HTML	**Marcaj de închidere**	**Semnificația**
<A	</A>	Marcaj de tip ancoră
HREF=*#nume*		Referință către o ancoră internă
NAME=*nume*		Definiția unei ancore interne

https://biblioteca-digitala.ro / https://unibuc.ro

Marcaje pentru imagini		
Marcaj HTML	**Marcaj de închidere**	**Semnificația**
<IMG		Marcaj de includere a imaginilor
SRC=url		Sursa fișierului grafic
ALT=text		Alternativă text pentru afișare, daca este cazul
ALIGN=aliniere		Aliniere imagine în pagină/alinierea materialului în jurul imaginii. Valori posibile: top, middle, bottom, left, right
HEIGHT=x		Înălțimea imaginii, în pixeli
WIDTH=x		Lățimea imaginii. În pixeli
BORDER=x		Chenarul din jurul imaginii
HSPACE=x		Spațiu suplimentar orizontal în jurul imaginii
VSPACE=x		Spațiu suplimentar vertical în jurul imaginii
<BR		Linie nouă
CLEAR=opt		Forțează trecerea la marginea specificată. Valori posibile: left, right, all

Marcaje pentru tabele și cadre		
Marcaj HTML	**Marcaj de închidere**	**Semnificație**
<TABLE	</TABLE>	Tabel Web
BORDER=*x*		Chenar în jurul tabelului
CELLPADING=*x*		Spațiu suplimentar în interiorul celulelor tabelului
CELLSPACING=x		Spațiu suplimentar între celulele tabelului
WIDTH=x		Lățimea tabelului
FRAME=*val*		Control detaliat al celulelor din tabel

https://biblioteca-digitala.ro / https://unibuc.ro

RULES=*val*		Control detaliat al chenarelor din tabel
BORDERCOLOR=*culoare*		Culoarea chenarului tabelului
BORDERCOLORLIGHT=*culoare*		Culoarea deschisă dintre cele două culori specificate
BORDERCOLORDARK=*culoare*		Culoarea închisă dintre cele două culori specificate
<TR	</TR>	Linie a tabelului
BGCOLOR=*culoare*		Culoare de fond a întregii linii
ALIGN=*aliniere*		Alinierea celulelor în această linie (left, center, right)
<TD	</TD>	Celulă de date a tabelului
BGCOLOR=*culoare*		Culoare de fond a celulei
COLSPAN=*x*		Numărul de coloane pe care se va întinde această celule de date
ROWSPAN=*x*		Numărul de linii pe care se va întinde această celule de date
ALIGN=*aliniere*		Alinierea orizontală a materialului din celulă. Valori posibile: left, center, right
VALIGN=*aliniere*		Alinierea verticală a materialului din celulă. Valori posibile top, middle, bottom
BACKGROUND=*url*		Specifică imaginea de fond a celulei
<FRAMESET	</FRAMESET>	Defineşte împărţirea paginii în cadre

https://biblioteca-digitala.ro / https://unibuc.ro

COLS=*x*		Numărul şi dimensiunea relativă a cadrelor pe orizontală
ROWS=*x*		Numărul şi dimensiunea relativă a cadrelor pe verticală
<FRAME	</FRAME>	Defineşte un cadru specific
SRC=*url*		URL-ul sursă al cadrului
NAME=*nume*		Numele cadrului(utilizat cu TARGET=*nume* ca parte a unui marcaj ancoră <A>)
SCROLLING=*derulare*		Defineşte opţiuni de derulare. Valori posibile: on, off, auto
FRAMEBORDER=*x*		Dimensiunea marginii din jurul cadrului
<NOFRAMES>	</NOFRAMES>	Secţiune a paginii afişată pentru utilizatorii care nu pot vizualiza o pagină cu cadre

Atributele marcajului TABLE	
Atribut	**Semnificaţie**
BORDER=*n*	Lăţimea zonei umbrite din jurul tabelului. Dacă n=0, se elimină şi caroiajul din interiorul tabelului
CELLSPACING=*n*	Spaţierea dintre celule
CELLPADDING=*n*	Spaţierea dintre chenar şi conţinutul celulei
WIDTH=*n*	Lăţimea dorită, prioritară fată de lăţimea calculată

Valori pentru atributul <TABLE FRAME=	
Valoare	**Semnificaţie**
VOID	Elimină toate muchiile exterioare ale tabelului
ABOVE	Afişează o muchie în partea superioară a cadrului tabelului
BELOW	Afişează o muchie în partea inferioară a cadrului tabelului

https://biblioteca-digitala.ro / https://unibuc.ro

HSIDES	Afişează câte o muchie în partea superioară şi inferioară a cadrului tabelului
LHS	Afişează o muchie în partea stângă a cadrului tabelului
RHS	Afişează o muchie în partea dreaptă a cadrului tabelului
VSIDES	Afişează o muchie în partea din stânga şi din dreapta a cadrului tabelului
BOX	Afişează câte o muchie pe toate laturile cadrului tabelului
BORDER	Afişează câte o muchei pe toate laturile cadrului tabelului

Marcaje ce permit îmbunătăţiri ale tabelelor şi cadrelor		
Marcaj HTML	**Marcaj de închidere**	**Semnificaţie**
<TABLE		
ALIGN=*left*		Aliniază tabelul la marginea din stânga a paginii, iar textul curge prin partea dreaptă
ALIGN=*right*		Aliniază tabelul la marginea din dreapta a paginii, iar textul curge prin partea stângă
HSPACE=*x*		Spaţiu suplimentar pe orizontală în jurul tabelului
VSPACE=*x*		Spaţiu suplimentar pe verticală în jurul tabelului
COLS=*x*		Specifică numărul de coloane ale unui tabel
<TD		
NOWRAP		Nu permite despărţirea textului pe linii în cadrul unri celule
ALIGN=*baseline*		Aliniază celulele de date cu linia de bază a textului adiacent
ALIGN=*caracter*		Aliniază o coloană faţă de un anumit caracter

https://biblioteca-digitala.ro / https://unibuc.ro

		(prestabilit este ".")
ALIGN=*justify*		Aliniază atât marginea din stânga, cât şi din dreapta a unui tabel
<COLGROUP>	</COLGROUP>	Specifică un număr de definiţii de coloane raelizate cu ajutorul marcajelo COL
<COL		Defineşte o anumită lăţime de coloană
WIDTH=*x*		Lăţimea unei coloane
<THEAD>	</THEAD>	Organizarea tabelelor: acest marcaj defineşte titlul tabelului
<TBODY>	</TBODY>	Defineşte corpul tabelului
<FRAMESET		
BORDER=*x*		Specifică starea "on" sau "off" pentru chenarul cadrului (1 sau 0)
FRAMEBORDER=*x*		Specifică mărimea chenarului
FARMESPACING=*x*		Mărimea spaţiului dintre două cadre adiacente
<FRAME		
MARGINHEIGHT=*x*		Spaţiul suplimentar de deasupra şi de dedesubtul unui anumit cadru
MARGINWIDTH=*x*		Spaţiul suplimentar de la stânga şi la dreapta unui anumit cadru
<IFRAME		Cadru intern
SRC=*url*		Sursa cadrului
NAME=*s*		Numele ferestrei
HEIGHT=*x*		Înălţimea cadrului înglobat

https://biblioteca-digitala.ro / https://unibuc.ro

WIDTH=*x*		Lățimea cadrului înglobat

Marcaje pentru formulare		
Marcaj HTML	**Marcaj de închidere**	**Semnificația**
<FORM	</FORM>	Formular HTML interactiv
ACTION=*url*		Programul CGI de pe serverul care recepționează datele
METHOD=*metoda*		Modul în care datele sunt transmise serverului (GET sau POST)
<INPUT		Câmp de text sau alte date de intrare
TYPE=*opțiune*		Tipul câmpului de intrare <INPUT>. Valori posibile:text, password, checkbox, hidden, file, radio, submit, reset, image
NAME=*nume*		Numele simbolic al valorii câmpului
VALUE=*valoare*		Conținutul prestabilit al câmpului de text
CHECKED=*opțiune*		Buton/casetă marcată în mod prestabilit
SIZE=*x*		Numărul de caractere al unui câmp de text
MAXLENGHT=*x*		Numărul maxim de caractere acceptate
<SELECT	</SELECT>	Grup de casete de validare
NAME=*nume*		Numele simbolic al valorii câmpului
SIZE=*x*		Numărul de articole de meniu care se afișează odată (prestabilit=1)
MULTIPLE		Permite selecția unor articole de meniu multiple
<OPTION		Alegere particulară într-un meniu <SELECT>
VALUE=*valoare*		Valoarea rezultantă a acestei selecții din meniu

https://biblioteca-digitala.ro / https://unibuc.ro

<TEXTAREA	</TEXTAREA>	Câmp de intrare de tip text pe linii multiple
NAME=*nume*		Numele simbolic al valorii câmpului
ROWS=*x*		Numărul de linii al casetei de text
COLS=*x*		Numărul de coloane (caractere) pe linie al casetei de text

Marcaje HTML care dispun de atributul TABINDEX	
Nume marcaj	**Semnificație**
A	Marcaj de tip ancoră
AREA	Hartă de imagine la client
OBJECT	Includerea uniu obiect
INPUT	Câmpuri de text, buton de interblocare, casetă de validare
SELECT	Meniu derulant sau cu opțiuni multiple
TEXTAREA	Casetă de intrare de tip text pe linii multiple
BUTTON	Similar cu INPUT TYPE=button

Alte marcaje pentru formulare		
Marcaj HTML	**Marcaj de închidere**	**Semmnificația**
<INPUT		
TYPE=*button*		Buton de uz general
onCLICK=*s*		Scriptul care se execută atunci când se execută clic pe buton (de regulă, un script JavaScript)
<LABEL	</LABEL>	Eticheta asociată unui anumit element de formular
FOR=*s*		Specifică elementul asociat etichetei (se utilizează ID=s în cadrul elementului)
<FIELDSET	</FIELDSET>	Împarte formularul în părți logice
ALIGN=*s*		Specifică alinierea legendei afișate (top bottom, left, right)
TABINDEX=*x*		Specifică ordinea elementelor

https://biblioteca-digitala.ro / https://unibuc.ro

		atunci când utilizatorul apasă tasta Tab
ACCESSKEY=*c*		Specifică tasta care asigură comanda rapidă de la tastatură asociată unui anumit element
DISABLED		Elementul este inactiv, dar este afişat pe ecran
READONLY		Elementul este afişat pe ecran, dar nu poate fi editat

Marcaje pentru hărți de imagini		
Marcaj HTML	**Marcaj de terminare**	**Semnificație**
<IMG		Marcajul de includere a imaginilor
ISMAP		Specifică faptul că este o hartă de imagine pe server
USEMAP=*nume_harta*		Utilizează harta la client specificată
<MAP>	</MAP>	Specifică regiunile unei hărți de imagine la client
NAME=*#nume_harta*		Numele unei hărți de imagine
<AREA		Definește o regiune particulară într-o hartă de imagine
SHAPE=*forma*		Forma unei regiuni. Valori posibile: rect, circ, point, poly
COORDS=*coordonate*		Coordonatele unei imagini: Valorile variază funcție de forma regiunii
HREF=*url*		URL-ul asociat regiunii specificate

https://biblioteca-digitala.ro / https://unibuc.ro

BIBLIOGRAFIE

1. **Acostăchioaie D.**, Programare C şi C++ pentru Linux, Polirom, Iaşi, 2002
2. **Acu C. I.**, Optimizarea Paginilor Web Editura Polirom, 1989.
3. **Aho A.V., Hopcroft J.E., Ullman J.D.**, Data Structures and Algorithms, Addison-Wesley,1985.
4. **Aho A.V., Hopcroft J.E., Ullman J.D.**, The Design and Analysis of Computer Algorithms, Addison-Wesley,1975.
5. **Albeanu Gr.**, Algoritmi şi limbaje de programare, Universitatea "Spiru Haret", Ed. Romania de Mâine, Bucureşti, 2000
6. **Aspru O.**, Grafica in Turbo C, Ed. Adidas, Rm. Valcea, 1994
7. **Barthelemey J.P., Cohen G., Lobstein A.**, Complexite.Algorithmique et problemes de communications, Masson Paris Milan Barcelone Bonn,1992.
8. **Bârză S.**, Algoritmică şi programare - Note de curs, Vol.I Programare Statică, Editura Universităţii Bucureşti, 2001;
9. **Bârză S.**, Culegere de probleme de algoritmică şi programare Vol.I Programare Statică, Editura Universităţii Bucureşti, 2001;
10. **Bereanu C**, Algoritmi. Complexitatea calculului (vol.I), Editura SITECH, Craiova,2000.
11. **Bereanu C.**, Algoritmi si complexitate, Tipografia Univ. Craiova,1998.
12. **Blajina O. A.**, Bazele programarii in limbajul C++, Ed. Printech, Bucuresti, 2000
13. **Borger E.**, Computability.Complexity.Logic,North-Holland, Amsterdam,1989.
14. **Brassard, G., Bratley, P.** "Algorithmics - Theory and Practice", Prentice-Hall, Englewood Cliffs, 1988.
15. **Catrina O., Cojocaru I.**, Turbo C++, Ed.Teora, Bucuresti, 1994
16. **Cazacu C.** , Teoria calculabilitatii efective, Ed. Univ. Iasi,1996.
17. **Cazacu C., Cazacu M.**, Teoria algoritmilor, Tipografia Univ.Iasi,1977.

https://biblioteca-digitala.ro / https://unibuc.ro

18. **Cormen T., Leiserson G., Rivest R.,** Introduction to Algorithms, Editia in lb. romana, Computer Libris Agora, Cluj, 2000.
19. **Cormen, T.H., Leiserson, C.E., Rivest, R.L.** "Introduction to Algorithms", The MIT Press, Cambridge, Massachusetts, 1992
20. **Cristea V., Giumale C., Kalisz E, Paunoiu Al.,** Limbajul C standard, Ed. Teora, Bucureşti, 1992
21. **Ellis, M., Stroustrup, B.** "The Annotated C++ Reference Manual", Addison-Wesley, Reading, 1991.
22. **Foley J.D., A. Van Dam**, Fundamentals of interactive Computer Graphics, Addison-Wesley, 1981(ed. I), 1990(ed. II) ·
23. **Frentiu M., B.Parv**, Elaborarea programelor. Metode si tehnici moderne, Ed.Promedia, Cluj-Napoca, 1994
24. **Frentiu M., V.Prejmereanu**, Algoritmica si programare, Lito. Univ. "Babes-Bolyai", Cluj-Napoca, 1995
25. **Frentiu, M., I.Lazar, S.Motogna Si V.Prejmereanu**, vol.I - Elaborarea algoritmilor, 1997, 188 pagini; vol.II - Ed.Universitatii "Babes-Bolyai", Cluj-Napoca
26. **Ghilic-Micu B., Rosca I. Gh., Apostol C.,** Algoritmi in programare, Bucuresti , Editura A.S.E., 2003
27. **Graham, R.L., Knuth, D.E., Patashnik**, O. "Concrete Mathematics", Addison-Wesley, Reading, 1989
28. **Gugoiu T.,** HTML prin exemple, Editura Teora, 2002.
29. **Hodkinson I.,** Computability, Algorithms and Complexity, department of Computing Imperial College 180 Queen's gate London SW7 2BZ UK,1996
30. **Homorodean M. A., Iosupescu I.**, Internet şi pagini Web, Editura Niculescu, 2001
31. **Horowitz, E., Sahni, S.** "Fundamentals of Computer Algorithms", Computer Science Press, Rockville, 1978
32. **Jamsa K., Lalani S., Weakley S.,** Programarea paginilor Web, Editura All Educational S.A. 1997.
33. **Knuth, D.E.** "Tratat de programarea calculatoarelor. Algoritmi fundamentali", Editura Tehnica, Bucuresti, 1974
34. **Knuth, D.E.** "Tratat de programarea calculatoarelor. Sortare si cautare", Editura Tehnica, Bucuresti, 1976
35. **Kozen D.C.,** The Design and Analysis of Algorithms, Springer-Verlag,1991.
36. **Lippman, S. B.** "C++ Primer", Addison-Wesley, Reading, 1989.

https://biblioteca-digitala.ro / https://unibuc.ro

37. **Livovschi, L., Georgescu, H.** "Sinteza si analiza algoritmilor", Editura Stiintifica si Enciclopedica, Bucuresti, 1986.
38. **Mitrana V.**, Provocarea algoritmilor. Probleme pentru concursurile de informatică, Ed. Agni,
39. **Mocanu M., Marian Gh, Badica C., BAdica Carmen**, 333 probleme de programare, Teora, 1993
40. **Moldoveanu Fl.**, Grafica pe calculator, TEORA, 1996
41. **Moldoveanu Fl., Racovita Z., Petrescu S., Hera G., Zaharia M.**, Grafica pe calculator, Ed. Teora.
42. **Negrescu L.**, Introducere in limbajul C, Ed.MicroInformatica, 1990
43. **Negrescu L.**, Limbajele C şi C++ pentru începători, Ed. microInformatica, Cluj-Napoca, 2001
44. **Odagescu I.**, Optimizarea algoritmilor, Ed. Militară, Bucureşti, 1991
45. **Patrut B.**, Aplicatii in C si C++, Teora, 1999
46. **Pătruț B.**, Grafică în OOP ... şi nu numai .., Ed. ADIAS, 1995
47. **Păun V.**, Bazele programarii in limbajul C++, Probleme alese, Uz didactic, Univ. din Pitesti, 1999
48. **Pavlidis T.**, Algorithms for Graphics and Image Processing, Springer-Verlag,1982
49. **Perjeriu E., Vaduva I.**, Indrumar pentru lucrări de laborator la cursul de Bazele Informaticii,Universitatea din Bucureşti, 1986
50. **Petrovici V., Goicea Fl.**, Programarea in limbajul C, Ed. Tehnica, 1993
51. **Popa M.**, Algoritmi in teoria grafurilor, Ed. Credis , 2003
52. **Popa M., Basca O.**, Culegere de probleme de structuri de date si sisteme de operare, Tipografia Univ. Buc., 1976
53. **Popa M.**, Metode si tehnici de programare, Suport de curs, 2003
54. **Popa M., Popa Mariana**, Grafuri si retele, Ed. Fundatiei Romania de Maine, 2003
55. **Popa M.**, Bazele modelarii retelelor de calculatoare, Ed. Univ. din Buc., 2004
56. **Popa M., Vaduva I.**, Culegere de probleme de sisteme informatice, Ed. Univ. Bucuresti, 1988
57. **Popovici C., Georgescu H., State L.**, Bazele Informaticii, vol. I, Universitatea din Bucureşti, 1990
58. **Popovici M. D., Popovici M. I.**, C++. Tehnologia orientată spre obiecte. Aplicații,Ed. Teora, Bucureşti, 2000
59. **Rinehart M.**, Sa invatam C++ pas cu pas, Editura Tehnica, 1997

https://biblioteca-digitala.ro / https://unibuc.ro

60. **Rogers D.F.,** Procedural elements for Computer Graphics, Mc Graw-Hill, 1985
61. **Rosca I. Gh., Ghilic-Micu B., Stoica M.,** Algoritmi in programare : aplicatii, Bucuresti , Editura A.S.E., 2003
62. **Saru D.,** Algoritmi si limbaje de programare, Editura didactica si pedagogica, Bucuresti, 1998
63. **Schildt Herbert**, C++ - Manual complet, ed. Teora, Bucureşti, 2000 Cluj-Napoca, 2000
64. **Sedgewick, R.** "Algorithms in C", Addison-Wesley, Reading, 1990
65. **Sedgewick, R.** "Algorithms", Addison-Wesley, Reading, 1988
66. **Sethi, R**. "Programming Languages. Concepts and Constructs", Addison-Wesley, Reading, 1989
67. **Skiena S.**, The Algorithm Design Manual, Springer Verlag, New York, Inc. 1998 (www.cs.sunysb.edu/~algorith/)
68. **Smeureanu I., Luca D., Odagescu I., Dardala M, Furtuna F.,** Grafica interactive pe calculatoare personale, Ed. Militara, Bucuresti, 1995
69. **Smith, J.H.** "Design and Analysis of Algorithms", PWS-KENT Publishing Company, Boston, 1989
70. **Somnea D.,** Iniţiere în JavaScript şi tehnologii Netscape, Ed. Tehnica, 1998
71. **Spircu T.,** Introducere în informatică, Ed. Teora, 1993
72. **Stanciu, V., Gheorghe, M., Pană, A.** - Informatică generală, Ed. Dual Tech, Bucureşti, 2001
73. **Standish, T.A**. "Data Structure Techniques", Addison-Wesley, Reading, 1979.
74. **Stroustrup, B.** "The C++ Programming Language", Addison-Wesley, Reading, 1991
75. **Stroustrup, B.** "The Design and Evolution of C++", Addison-Wesley, Reading, 1994
76. **Surcel Tr., Sofronie Gh., Baron C., Toma L.,** Informatică generală, Ed. Calipso, Bucureşti, 1998
77. **Tanenbaum A.** S., Retele de calculatoare, editia a IV-a, Ed. Biblos, Bucuresti, 2003
78. **Taylor D**., Crearea paginilor WEB cu HTML 4, Editura TEORA, 1999
79. **Thomas C.H., Charles E.Leiserson, Ronald R.Rivest**, Introducere în algoritmi, Ed.Libris, 2000

VERIFICAT 2017
VERIFICAT 2007
https://biblioteca-digitala.ro / https://unibuc.ro

80. **Tudor S.**, Bazele programarii in C++, Ed. L&S Infomat
81. **Tudor S., Hutanu V.**, Crearea şi programarea paginilor Web Editura L&S Infomat, 2000.
82. **Văduva I., Barbu Gh.,** Bazele Informaticii, Editura Tehnică, 1997
83. **Vlada M., Posea A., Nistor I., Constantinescu C.,** Grafică pe calculator în limbajele Pascal şi C, Implementare şi Aplicaţii, vol. I,II, Ed. Tehnică, 1992
84. **Wagner R., Wyke A.**, JavaScript, Editura Teora, 2001.
85. **Weihrauch K.,** Computability,Springer,Berlin,1987
86. **Wilf H.S.,** Algorithmes et complexite, Prentice-hall,London,1989
87. **Williams M., Visual C++4, Teora, 1997**
88. **Williams R., Tollet J.**, Design pentru Web, Editura Corint, 2003.
89. **Zaharia M. D.,** Structuri de date şi algoritmi - exemple în limbajele C şi C++, Ed. microInformatica, Cluj-Napoca, 2002
90. **Zoican R., Munteanu N., Popovici E.**, Tehnici de programare. Structuri de date, algoritmi, aplicatii, Editura Media Publishing, Bucuresti, 1995

https://biblioteca-digitala.ro / https://unibuc.ro

Tiparul s-a executat sub c-da nr. 1330/2004
la Tipografia Editurii Universității din București

https://biblioteca-digitala.ro / https://unibuc.ro

DATA RESTITUIRII		
21 NOV 2005	12.10.2013	
22 NOV 2005	05.02.2014	
03. DEC. 2005	04 APR. 2014	
30 JAN 2006		
30 JAN. 2007		
25. FEB. 2009		
0. APR. 2009		
1 IUNI 200		
01. NOV 2009		
23. APR. 2010		
3. NOV. 201		
—		

BIBLIOTECA CENTRALA
UNIVERSITARA CAROL I

DE SPIRITU ET ANIMA

316.500 b.

https://biblioteca-digitala.ro / https://unibuc.ro

Prin această carte, autorii își exprimă concepția asupra modului de abordare a cursului numit *Informatică* de la Facultățile de Informatică și de la cele de Matematică-Informatică pentru specializările Informatică și Matematică-Informatică. Utilizarea eficientă a calculatorului pentru rezolvarea de probleme din orice domeniu nu este posibilă fără o inițiere în domeniul algoritmilor, al metodelor și tehnicilor de elaborare a acestora, dar și al complexității lor. Acesta este motivul pentru care prima parte a lucrării dezvoltă metodele de reprezentare a algoritmilor, prezintă câteva metode și tehnici de elaborare a algoritmilor și studiază complexitatea algoritmilor elaborați prin fiecare din aceste tehnici.

Structurată pe trei părți, primele două având câte patru capitole și ultima două capitole, lucrarea tratează aspectele fundamentale privind reprezentarea, elaborarea și complexitatea algoritmilor (în prima parte), privind limbajul C++, programarea orientată obiect, implementarea structurilor de date de tip stivă, coadă, arbore în C++ și privind grafica pe calculator în C++ (partea a doua) și în fine privind modalități eficiente de realizare a paginilor Web interactive și dinamice (partea a treia).

ISBN 973575979-9
9789735759797

https://biblioteca-digitala.ro / https://unibuc.ro