Metode de programare

Asist.univ.dr. Simona Elena Vârlan

Structura curs

- ✓**1 ore de curs** – ambele specializări, titular curs Simona Elena Vârlan (cabinet D213, vineri)

- ✓ 2 ore de laborator + 1 oră de seminar

  titular Simona Elena Vârlan

Evaluarea

- Examen final (E) – 50%
- Evaluare pe parcursul semestrului a activității de seminar/laborator prin rezolvarea de probleme/teme / lucrare de control (L) – 50%

- Nota finală: (L + E) /2 >= 5

Curs 1

Recapitulare cunoștințe fundamentale de la cursurile de Programare și Structuri de date

Sumar

1. Noţiunea de algoritm

- În procesul de rezolvare a unei probleme folosind calculatorul există două etape:
- *1. Definirea şi analiza problemei*
- *2. Proiectarea şi implementarea unui algoritm* care rezolvă problema
- *1. Definirea şi analiza problemei* poate fi la rândul ei descompusă în:
- ✓ specificarea datelor de intrare
- ✓ specificarea datelor de ieşire

1. Noţiunea de algoritm

Specificarea **datelor de intrare** constă în:

1. Ce date vor fi primite la intrare

2. Care este formatul (forma lor de reprezentare) datelor de intrare

3. Care sunt valorile permise sau nepermise pentru datele de intrare

4. Există unele restricţii (altele decât la 3) privind valorile de intrare

5. Câte valori vor fi la intrare, sau dacă nu se poate specifica un număr fix de valori, cum se va şti când s-au terminat de introdus datele de intrare.

1. Noţiunea de algoritm

Specificarea **datelor de ieşire** trebuie să ţină cont de următoarele aspecte:

1. Care din valorile rezultate în cursul aplicării algoritmului de calcul, asupra datelor de intrare, vor fi afişate (necesare utilizatorului), în acest pas se face diferenţierea clară între *date intermediare* şi *date de ieşire.*

2. Care va fi formatul datelor de ieşire (de exemplu un număr real poate fi afişat cu trei sau cu cinci zecimale, sau un text poate fi afişat integral sau parţial).

1. Noţiunea de algoritm

- O definiţie a noţiunii de algoritm poate fi:

*O **succesiune finită** de operaţii cunoscute care se execută într-o **succesiune logică bine stabilită** astfel încât plecand de la un set de date de intrare, să obtinem într-un interval de **timp finit** un set de date de ieşire.*

Un exemplu simplu de algoritm ar fi suita de operaţii matematice făcută în rezolvarea unei ecuaţii matematice de gradul II:

$$a \cdot x^2 + b \cdot x + c = 0$$

coeficienții a, b și c se schimbă dar modul de procesare a valorilor lor, nu

1. Noţiunea de algoritm

- Proprietăţile unui algoritm sunt:

1. ***Generalitate***. Un algoritm trebuie să poată fi utilizat pentru o *clasă* întreagă *de probleme*, nu numai pentru o problemă particulară. Din această cauză, o metodă de rezolvare a unei ecuații particulare nu poate fi considerată *algoritm*.
2. ***Finitudine***. Orice algoritm trebuie să permită obținerea rezultatului după un *număr finit* de prelucrări (pași).
3. ***Determinism***. Un algoritm trebuie să prevadă, fără ambiguități și fără neclarități, modul de soluționare a tuturor situațiilor care pot să apară în rezolvarea problemei. Dacă în cadrul algoritmului nu intervin elemente aleatoare, atunci ori de câte ori se aplică algoritmul aceluiași set de date de intrare trebuie să se obțină același rezultat.

1. Noţiunea de algoritm

Tipuri de prelucrări:

Prelucrările care intervin într-un algoritm pot fi *simple* sau *structurate*.

- ***Prelucrările simple*** sunt *atribuiri* de valori variabilelor, eventual prin evaluarea unor expresii;
- ***Prelucrările structurate*** pot fi de unul dintre tipurile:
  - ***Liniare***. Sunt secvențe de prelucrări simple sau structurate care sunt efectuate în ordinea în care sunt specificate;
  - ***Alternative***. Sunt prelucrări caracterizate prin faptul că în funcție de realizarea sau nerealizarea unei condiții se alege una din două sau mai multe variante de prelucrare;
  - ***Repetitive***. Sunt prelucrări caracterizate prin faptul că aceeași prelucrare (simplă sau structurată) este repetată cât timp este îndeplinită o anumită condiție.

1. Noţiunea de algoritm

- Concluzia:

Un algoritm este independent de tipul de limbaj în care este transpus sau de tipul de calculator pe care este executat.

2. Reprezentarea unui algoritm

Modul de descriere a unui algoritm, este independent de un limbaj de programare, existând două metode clasice:

1. metoda schemei logice

2. metoda pseudocod-ului

3. Elementele programării structurate

- Structurile de control de bază – cu ajutorul lor se pot descrie algoritmii: secvența, decizia alternativă binară (if) și structura repetitivă cu test inițial (while)
- Care sunt cele derivate?
- Cum sunt reprezentate cu scheme logice sau pseudocod?

3. Elementele programării structurate

- Programarea pe baza celor trei structuri (şi a celor auxiliare) se numeşte **programare structurată**.

- Informaticienii Böhm şi Jacopini au formulat un **principiu al programării structurate**, sub forma unei teoreme: cele trei structuri (secvenţa, decizia şi repetiţia condiţionată anterior) sunt suficiente pentru a descrie orice algoritm.

Teorema programării structurate a lui Böhm şi Jacopini care se poate enunţa astfel:

Orice schemă logică nestructurată poate fi înlocuită cu una echivalentă, dar structurată, prin adăugarea de noi acţiuni şi condiţii.

Aşadar, orice program poate fi pus sub o formă structurată, adică să conţină doar structurile de bază şi/sau structurile auxiliare, prin utilizarea unor variabile boolene asociate unor funcţii suplimentare

3. Elementele programării structurate

Exemplu:

Algoritmul următor determină cel mai mic element (notat min) din şirul de n elemente X și este **într-o formă nestructurată**

```
        Citeste n, X
        i = 1
        min = X(0)
A:      dacă i>n atunci
                treci la pasul B  -> instrucțiune de salt necondiționată, nu e secvențială
        altfel
                dacă X(i)<min atunci
                        min = X(i)
                i = i+1
                treci la pasul A
B:      Scrie min
```

3. Elementele programării structurate

Exemplu:

Algoritmul rescris **într-o formă structurată**, utilizând structurile de bază

```
Citeste n, X
min = X(0)
i = 1
cât timp i <= n execută
        dacă X(i)<min atunci
                min = X(i);
        i = i+1
Scrie min
```

Cum se rescrie algoritmul folosind o structură repetitivă derivată?

4. Concepția unui algoritm

Pași necesari:

1. Problema care va fi rezolvată, trebuie citită cu atenţie.

2. Apoi se stabilesc prelucrările care sunt necesare obţinerii rezultatelor dorite.

Pentru a crea un algoritm eficient trebuie evidenţiate datele de intrare şi datele de ieşire.

5. Proiectarea algoritmilor

Există două metode generale de proiectare a algoritmilor, a căror denumire provine din modul de abordare a rezolvării problemelor:

metoda top-down (descendentă) şi

metoda ascendentă (bottom-up).

5. Proiectarea algoritmilor

Motoda top-down

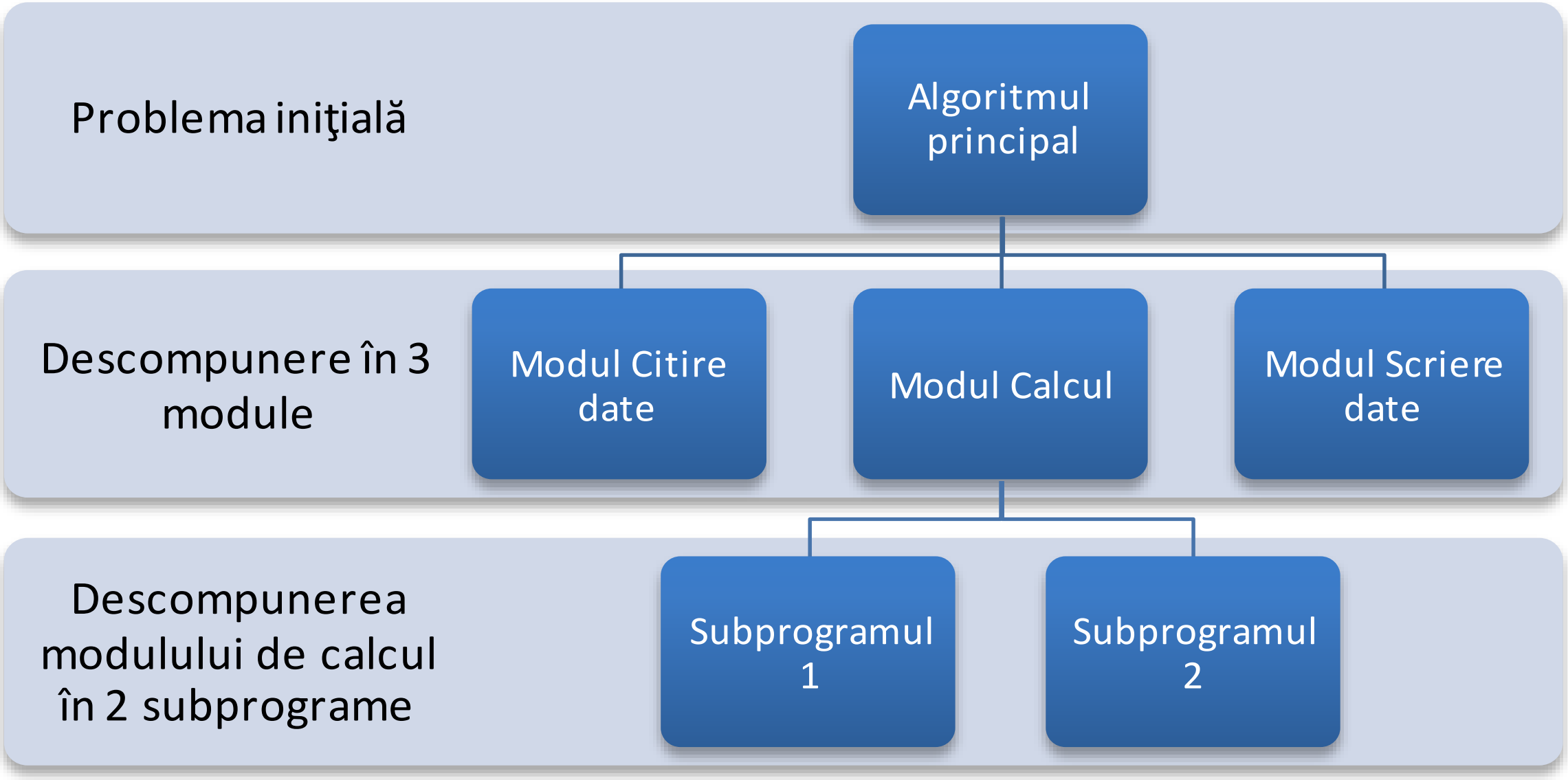
Proiectarea descendentă (top-down) porneşte de la problema de rezolvat, pe care o descompune în mai multe probleme ce se pot rezolva separat (care la rândul lor pot fi descompuse în subprobleme).

Primul pas al metodei descompune algoritmul principal în subprobleme.

Pasul doi – pentru fiecare subproblemă în parte elaborăm un subalgoritm de rezolvare, urmând aceeași metodă.

În final se combină soluțiile să formeze soluția problemei inițiale.

5. Proiectarea algoritmilor



5. Proiectarea algoritmilor

Avantajele proiectării top-down(cunoscută şi sub denumirea "Divide et impera")

Permite programatorului să reducă complexitatea problemei: subproblemele în care a fost descompusă fiind mai simple, şi să amâne detaliile pentru mai târziu. Când descompunem problema în subprobleme nu ne gandim cum se vor rezolva subproblemele ci doar care sunt ele şi conexiunile dintre ele.

Permite lucrul în echipe mari: Prin descompunerea problemei în mai multe subprobleme, fiecare subproblemă poate fi dată spre rezolvare unei subechipe.

5. Proiectarea algoritmilor

Metoda bottom-up

În cazul metodei ascendente (bottom-up) va fi scris mai întâi subalgoritmul apelat şi apoi cel care apelează. Se ajunge astfel la o mulţime de subalgoritmi care se apeleaza între ei. Este important să se cunoască care subalgoritm apelează pe care, lucru redat printr-o structură arborescentă, ca şi în cazul programarii descendente.

Principalul dezavantaj - faptul că erorile vor fi detectate târziu, abia în faza de asamblare.

De cele mai multe ori proiectarea algoritmilor combină cele două metode, ajungând astfel la o proiectare mixtă.

5. Proiectarea algoritmilor

Proiectarea modulară

Prin proiectare / programare modulară un algoritm se va rezolva prin folosirea **modulelor.**

Modulul este considerat o unitate structurală de sine stătătoare, care poate fi un program, un subprogram, sau o unitate de program. Poate fi format din mai multe submodule.

Fiecare modul realizează o funcţie bine precizată în cadrul întregului program.

Modulele sunt independente, dar pot “comunica” între ele prin transmitere de parametri sau prin apeluri.

5. Proiectarea algoritmilor

Proiectarea modulară

Programarea modulară este strâns legată de programarea ascendentă şi de programarea descendentă, ambele presupunând folosirea subalgoritmilor pentru toate subproblemele întâlnite.

Avantajele programarii modulare:

- Descompunerea unei probleme complexe în subprobleme;
- Probabilitatea apariţiei erorilor în conceperea unui subprogram este mult mai mică decât dacă s-ar lucra cu tot programul iniţial.
- Rezolvând o problemă mai simplă (un modul), testarea acestui modul se poate face mult mai uşor decât testarea întregului algoritm;
- Permite munca în echipă, modalitate prin care se ajunge la scurtarea termenului de realizare a programului;
- Modulele se pot refolosi.

5. Proiectarea algoritmilor

Proiectarea structurată – concluzii finale

Înglobează toate metodele de proiectare a algoritmilor descrise anterior.

De asemenea, programarea structurată este definită ca fiind programarea în care abordarea este top-down, organizarea muncii este facută pe principiul echipei, iar în proiectarea algoritmilor se folosesc cele trei structuri de calcul definite de Bohm-Jacopini (secvențială, decizională, repetitivă).

6. Corectitudinea algoritmilor

Corectitudinea *– spunem că un algoritm este corect dacă el furnizează în mod corect datele de ieşire pentru toate situaţiile regăsite în datele de intrare.*

Există totuşi algoritmi care sunt corecţi, clari, generali şi furnizează soluţia într-un timp finit însă mai lung sau folosesc mai multă memorie decât alţi algoritmi.

Vom încerca să găsim algoritmi care să dea soluţia într-un timp cât mai scurt, cu cât mai puţină memorie folosită. Cu alte cuvinte vom încerca să elaboram algoritmi **eficienţi.**

Numim deci ***eficienţă*** *- capacitatea algoritmului de a da o soluţie la o problemă într-un timp de executie cât mai scurt, folosind cât mai puţină memorie.*

În elaborarea algoritmilor apar frecvent erori. Prezentăm câteva dintre cele mai întâlnite tipuri de erori şi câteva metode de tratare a lor.

6. Corectitudinea algoritmilor

Erori în datele iniţiale

- Provin din introducerea, ca valori de intrare, a unor date de tip diferit de cel prevazut la elaborarea algoritmului. În aceste situații algoritmul nu se comportă în modul aşteptat, generând erori sau eventuale date de ieşire eronate.
- O altă eroare frecventă este confuzia între variabilele globale şi locale

Erori de trunchiere, erori de rotunjire şi erori de calcul

- Eroare de trunchiere este diferenţa dintre rezultatul exact (pentru datele de intrare curente) şi rezultatul furnizat de un algoritm dat utilizând aritmetica exactă.
- Eroare de rotunjire este diferenţa dintre rezultatul produs de un algoritm dat utilizând aritmetica exactă şi rezultatul produs de acelaşi algoritm utilizând o aritmetică cu precizie limitată (de exemplu aritmetica virgulei mobile).
- Eroarea de calcul este suma dintre eroarea de trunchiere şi eroarea de rotunjire, dar de obicei una dintre acestea predomină.

6. Corectitudinea algoritmilor

Erori reziduale

- Sunt acele erori care rezultă din diferenţa între valorile obţinute efectiv şi cele asteptate a fi obţinute.

Erori de sintaxă

- Sunt erorile cele mai frecvente şi cel mai uşor de corectat (greșeli de tastare, de punctuație, de plasare a instrucțiunilor, confuzia între șirurile de caractere și numere).

Erori de logică (semantice)

- Se generează atunci când nu se obţin rezultatele aşteptate deși nu sunt generate erori în timpul execuției programului sau erori sintactice.

6. Corectitudinea algoritmilor

Tehnici de depistare a erorilor

Urmarirea secventială a algoritmului (logica programului) folosind reprezentarea în pseudocod a algoritmului.

Utilizarea comentariilor - este bine să inserăm cât mai multe comentarii în algoritmi cu explicații sau chiar transformarea liniilor de cod in comentarii și rularea programului pentru a urmări mai ușor modul de execuție și rezultatele obținute.

Adaugarea de instrucţiuni de afişarea a valorilor variabilelor pas cu pas, a rezultatelor unor expresii sau a unor mesaje pentru a verifica execuția corectă a programului.

Descompunerea programelor complexe în mai multe module pentru a mări gradul de localizare a erorilor dar și pentru reutilizarea codului.

7. Verificarea corectitudinii algoritmilor

Un algoritm realizat trebuie să fie:

- corect,
- clar,
- sigur în funcţionare,
- uşor de modificat,
- portabil,
- eficient,
- însoţit de o documentaţie corespunzătoare.

Există numeroase tehnici de verificăre şi validare a algoritmilor:

- **tehnica testarii programelor** şi **depanarea programelor**

7. Verificarea corectitudinii algoritmilor

Tehnica testării programelor

Prin testare se verifică funcţionarea programului, de a găsi posibilele defecte ale acestuia astfel încât programul să funcționeze la parametri prevăzuți.

Procesul de detectare şi apoi de eliminare a erorilor unui algoritm are două componente, numite:

- verificare
- Validare

Testarea programului pe diverse seturi de date de test

Se bazează pe construirea unor eşantioane de date de intrare care să conducă la depistarea unor erori în funcționarea programului.

Seturile de date de test trebuie să acopere inclusiv **situații de excepție** şi să verifice dacă fiecare subproblemă a problemei date este rezolvată corect .

7. Verificarea corectitudinii algoritmilor

Metode de testare a programelor (teste software)

1. **Testarea funcţională sau metoda cutiei negre (black box)** - presupune construirea datelor de test astfel încât să permită testarea **fiecărei funcţiuni a programului (funcționalitatea programului)**;

- Tester-ul știe doar ce trebuie să facă programul nu și cum face;
- Cazurile de testare se construiesc cu ajutorul cerințelor și specificațiilor.

2. Testarea structurală sau metoda cutiei transparente (white box) - presupune construirea datelor de test astfel încât **toate părţile programului** să poată fi testate.

- Se testează structura internă și modul de prelucrare a datelor;
- Tester-ul trebuie să știe cum a fost făcut programul și să aibă experiență de programare

3. **Testarea prin metoda cutiei gri (gray box)** – presupune ca tester-ul să cunoască algoritmii și structurile de date din program, dar să testeze ca un utilizator final.

7. Verificarea corectitudinii algoritmilor

Depanarea unui program (debugging)

Constă în localizarea erorii, determinarea naturii sale. Ea se poate face în mod:

- static, după executarea programului
- dinamic, în timpul execuţiei acestuia

Mai toate mediile de programare ofera posibilitatea de debugg care permite execuția programului pas cu pas sau stabilirea unor puncte de întrerupere în care să inspectăm starea programului.

8.Algoritmi elementari ce folosesc structuri fundamentale

1. Să se scrie un program care verifică dacă un număr n este perfect sau nu.

Un număr perfect este egal cu suma divizorilor lui, inclusiv 1 (exemplu: 6 = 1 + 2 + 3).

Exemplu:

- Pentru n = 28, se va afişa mesajul

  Numar perfect (divizorii lui 28 sunt 1, 2, 4, 7, 14)

8.Algoritmi elementari ce folosesc structuri fundamentale

Pas 1: Stabilim care sunt datele de intrare, adică cele care vor fi prelucrate cu ajutorul algoritmului, împreună cu datele de ieşire.

În cazul problemei date, avem:

- Date de intrare: n număr natural
- Date de ieşire: mesaj corespunzător

8.Algoritmi elementari ce folosesc structuri fundamentale

Pas 2: Analiza problemei

1) La începutul problemei, vom inițializa o variabilă de tip suma cu valoarea 0.

2) Pentru fiecare valoare i de la 1 la n-1 vom verifica dacă i este divizor al numarului n. Daca este divizor atunci il insumam la valoarea s.

3) Verificam daca suma obtinuta este egala cu numarul n. Daca da atunci scriem mesajul

‘Numarul este perfect’.

8.Algoritmi elementari ce folosesc structuri fundamentale

```
natural n, i, s
citeşte n
i = 1
s = 0
repetă
        dacă n % i = 0 atunci
                s = s + i
        sfârşit dacă
        i = i + 1
până când i > n-1
```

->

```
dacă s = n atunci
        scrie ‘Este perfect’
altfel
        scrie ‘NU este perfect’
sfârşit dacă


stop
```

8.Algoritmi elementari ce folosesc structuri fundamentale

Se citesc două numere întregi a şi b. Să se realizeze in pseudocod un algoritm care să verifice dacă cele doua numere sunt prietene.

Spunem ca două numere sunt prietene dacă suma divizorilor proprii ai unui număr este egală cu celălalt număr şi invers.

Exemplu:

Pentru n = 220, si m = 284 se vor afişa valorile:

Divizorii lui 220, sunt 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 și 110. Suma este 284

Divizorii lui 284, sunt 1, 2, 4, 71 și 142. Suma lor este 220

8.Algoritmi elementari ce folosesc structuri fundamentale

Pas 1: Stabilim care sunt datele de intrare, adică cele care vor fi prelucrate cu ajutorul algoritmului, împreună cu datele de ieşire.

În cazul problemei date, avem:

- Date de intrare: **n** si **m** numere naturale
- Date de ieşire: numerele sunt sau nu prietene

8.Algoritmi elementari ce folosesc structuri fundamentale

Pas 2: Analiza problemei

- La începutul problemei, vom iniţializa valoarea unei variabile pentru suma divizorilor lui **n** cu 0, iar apoi valoarea unei variabile pentru suma divizorilor lui **m** cu 0.
- Apoi, într-un ciclu repetitiv avand n/2 pasi vom calcula suma divizorilor lui n.
- Apoi, într-un ciclu repetitiv avand m/2 pasi vom calcula suma divizorilor lui m.
- La sfarsit vom verifica daca suma divizorilor primului numar este egală cu cel de-al doilea număr, iar suma divizorilor celui de-al doilea numar este egală cu primul număr

8.Algoritmi elementari ce folosesc structuri fundamentale

```
natural n, m, i, j, suma_n, suma_m
citeşte n
suma_n = 0

pentru i=1,n/2 execută

        daca n%i=0 atunci
                suma_n = suma_n + 1
        sfârșit daca

sfârșit pentru
suma_m = 0
pentru j=1,m/2 execută
        dacă m%j=0 atunci
                suma_m = suma_m + j
        sfârșit dacă
sfârșit pentru
dacă suma_n = m AND suma_m=n  atunci
        scrie “Numere prietene”
altfel
        scrie “Numere neprietene”
sfârșit dacă
stop
```

8.Algoritmi elementari ce folosesc structuri fundamentale

Se citeşte un număr întreg a. Să se realizeze un algoritm care să verifice dacă numărul citit este sau nu palindrom. Numim palindrom un număr care este egal cu oglinditul său.

De exemplu dacă se citeşte pentru **a** valoarea 323 atunci algoritmul va afişa „PALINDROM”, iar dacă va citi 123 va afişa „NU”.

Pas 1: Stabilim care sunt datele de intrare, adică cele care vor fi prelucrate cu ajutorul algoritmului, împreună cu datele de ieşire.

- Date de intrare: a număr natural
- Date de ieşire: mesaj corespunzător

8.Algoritmi elementari ce folosesc structuri fundamentale

Pas 2: Analiza problemei

Algoritmul se bazează pe problema de calcul a numărului invers.

De aceea algoritmul descompune în cifre numărul **a** şi formează numărul **invers**, dupa care compară acest număr invers cu numărul iniţial citit.

Pentru asta avem nevoie de o copie a numărului „**a**” deoarece în structura repetitivă se modifică valoarea numărului introdus, iar noi avem nevoie de valoarea iniţiala pentru verificare.

8.Algoritmi elementari ce folosesc structuri fundamentale

```
citeste a
aux = a  // salvez valoarea inițială a lui a în aux
inv = 0 //initial inv este nul

cât timp aux ≠ 0 execută
        inv = inv * 10 + aux % 10
        aux = aux / 10

dacă inv = a atunci
        scrie ”Palindrom”
altfel
        scrie ”Nu este palindrom”
```

9. Structuri de date - recapitulare

O ***structură de date*** este o colecţie de elemente asupra căreia se pot efectua anumite operaţii.

Clasificarea structurilor de date:

1.În funcţie de ***tipul datelor*** memorate în cadrul structurii, structurile de date se pot clasifica în:

- •structuri de date ***omogene*** – toate datele componente sunt de acelaşi tip (de exemplu tabloul);
- •structuri de date ***neomogene*** – pot conţine date de tipuri diferite (de exemplu înregistrarea).

2. În funcţie de ***modul de alocare a memoriei interne***, structurile de date sunt de două tipuri:

- •structuri de date ***statice (folosind vectori)*** – ocupă o zonă de memorie de dimensiune fixă, alocată pe întreaga durată a execuţiei blocului în care este declarată (de exemplu tabloul, fişierul, lista, stiva, coada);
- •structuri de date ***dinamice (folosind pointeri)*** – ocupă o zonă de memorie de dimensiune variabilă, alocată pe parcursul execuţiei programului, la cererea explicită a programatorului (de exemplu lista, stiva, coada).

10. Structura de date de tip listă

LISTA – structura de date logica, liniara, cu date omogene, in care fiecare element are un succesor si un predecesor, exceptand primul element, care nu are decat succesor si ultimul element care nu are decat predecesor.

Implementarea listelor:

a) in functie de modul de alocare a memoriei interne:

- implementare statica (folosind vectori)
- implementare dinamica (folosind pointeri)

b) in functie de modul de aranjare a elementelor listei:

- secventiala – numai statica
- inlantuita – statica si dinamica

Operaţii elementare care se pot efectua asupra unei liste:

- crearea unei liste vide;
- inserarea unui element în listă;
- eliminarea unui element din listă;
- accesarea unui element din listă;
- afişarea elementelor unei liste.

Exemplu

`14, 2, 6, 77`

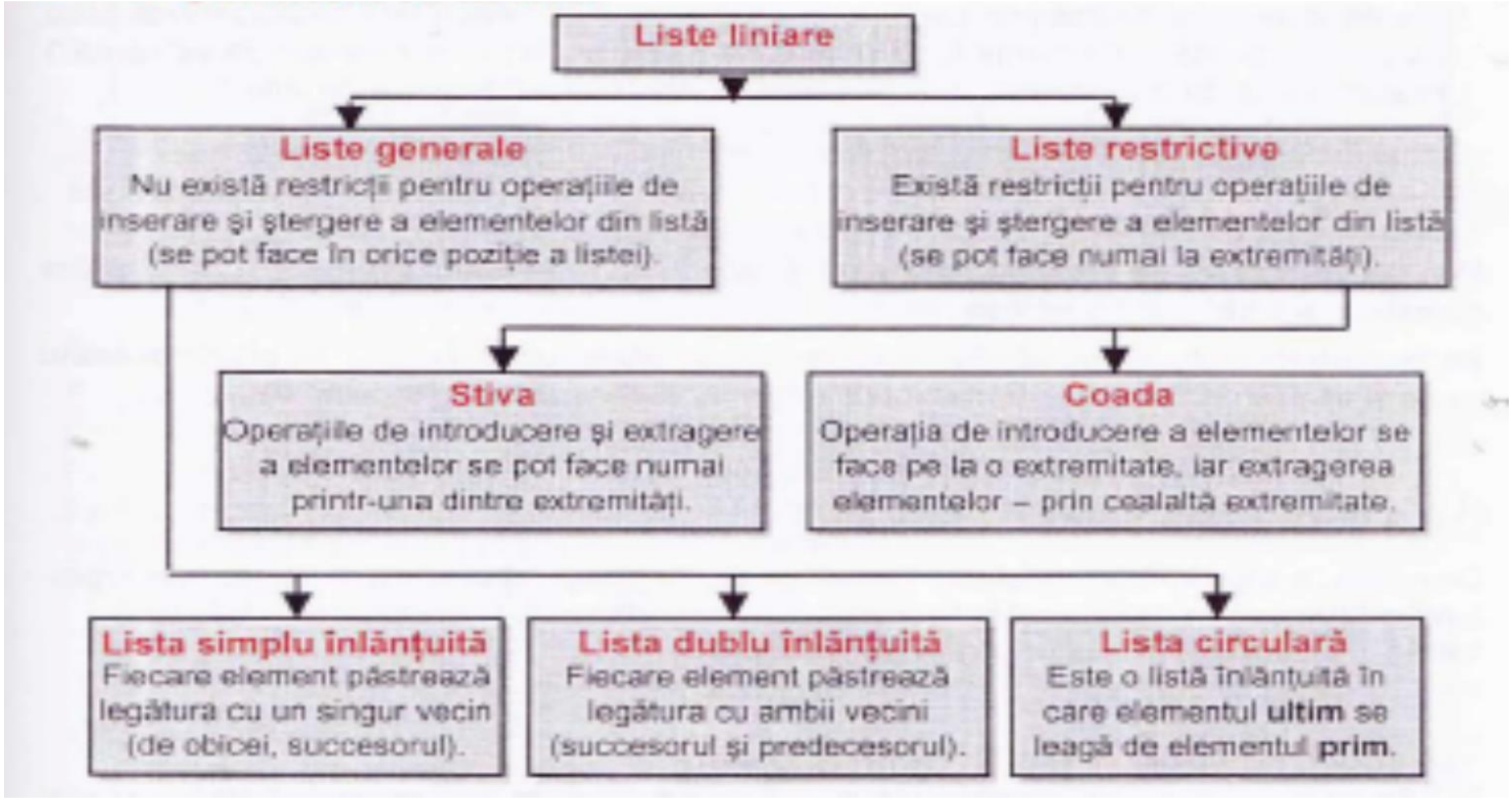
LISTA

14	2	6	77

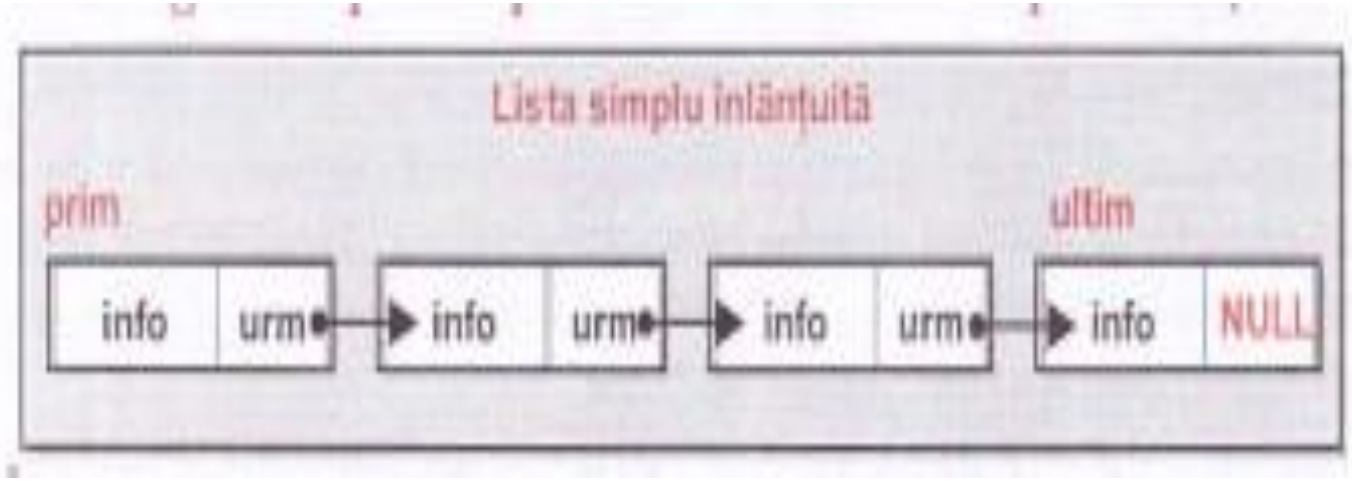
Implementarea prin alocare înlănțuită

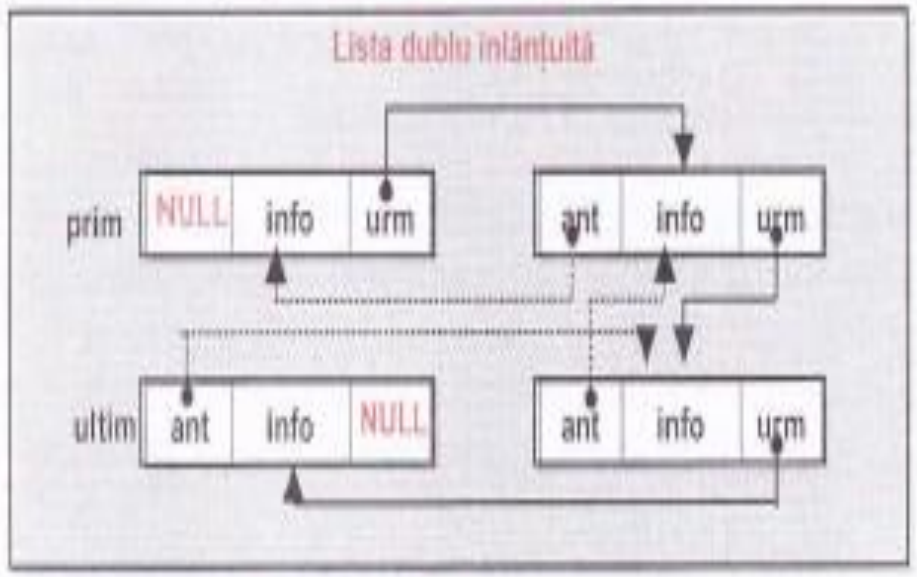
- Nodurile sunt aranjate aleatoriu in memorie.
- Necesita un mecanism prin care sa se precizeze ordinea reala a nodurilor adica:
  - pozitia primului nod (prim)
  - pozitia ultimului nod (ultim)
  - succesorul fiecarui nod (urm)
- Metoda folosita este ca un nod al listei sa contina doua tipuri de informatii:
  - informatia propriu-zisa
  - informatia de legatura – adresa urmatorului nod.

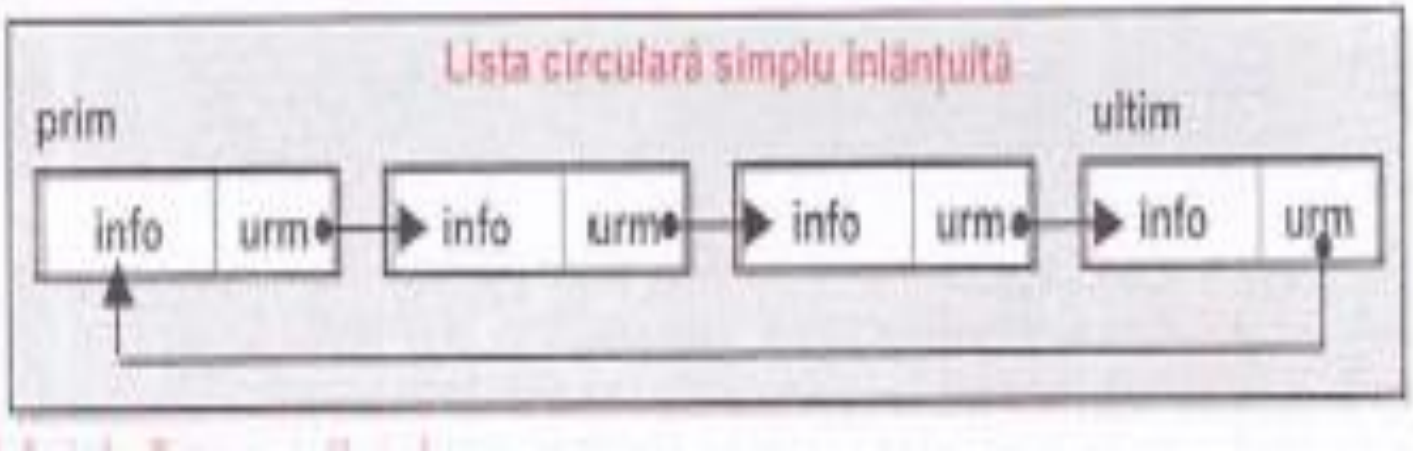
Clasificarea Listelor



Clasificarea listelor







Declararea listei

```
const unsigned NMAX=100;
typedef unsigned  adresa;
struct nod
{
    <tip_1>  <info_11>;
  ……..……
    adresa urm;
};
 nod lista [NMAX+1];
```

Algoritmi pentru prelucrarea listelor

Se considera ca un nod al listei contine numai un camp cu informatie si campul pentru realizarea legaturii:

```
const unsigned NMAX=100;
typedef unsigned  adresa;

struct nod
{int info;
adresa urm;};
 nod lista [NMAX+1];

adresa prim, ultim, p;
unsigned nr_el, liber[NMAX];
int n;
```

unde:

prim – adresa primului nod

ultim – adresa ultimului nod

p – adresa nodului curent

n – valoarea atribuita campului info

nr_el – lungimea listei

liber – vector harta a nodurilor

- liber [p] are valoarea 1 daca nodul p este liber si 0 daca este ocupat

Algoritmi pentru prelucrarea listelor

Testarea listei:

a)Lista vida

```
int este_vida (adresa prim )
{ return prim = NULL; }
```

b)Lista plina

```
int este_plina ( )
{ return nr_el = NMAX; }
```

Algoritmi pentru prelucrarea listelor

Initializarea listei: se creeaza lista vida

```
void init ( adresa &prim, adresa &ultim )
{ ultim = prim = NULL;
   nr_el = 0;
   for (  p = 1; p < = NMAX; p ++ )
         liber [ p ] = 1;
}
```

Algoritmi pentru prelucrarea listelor

Alocarea memoriei: se gaseste prima pozitie libera si se aloca memorie pentru noul nod.

```
adresa aloc_mem (  )
{
   for (  p = 1; ! liber [ p ]; p ++ ) ;
   liber [ p ] = 0 ; nr_el ++;
}
```

Algoritmi pentru prelucrarea listelor

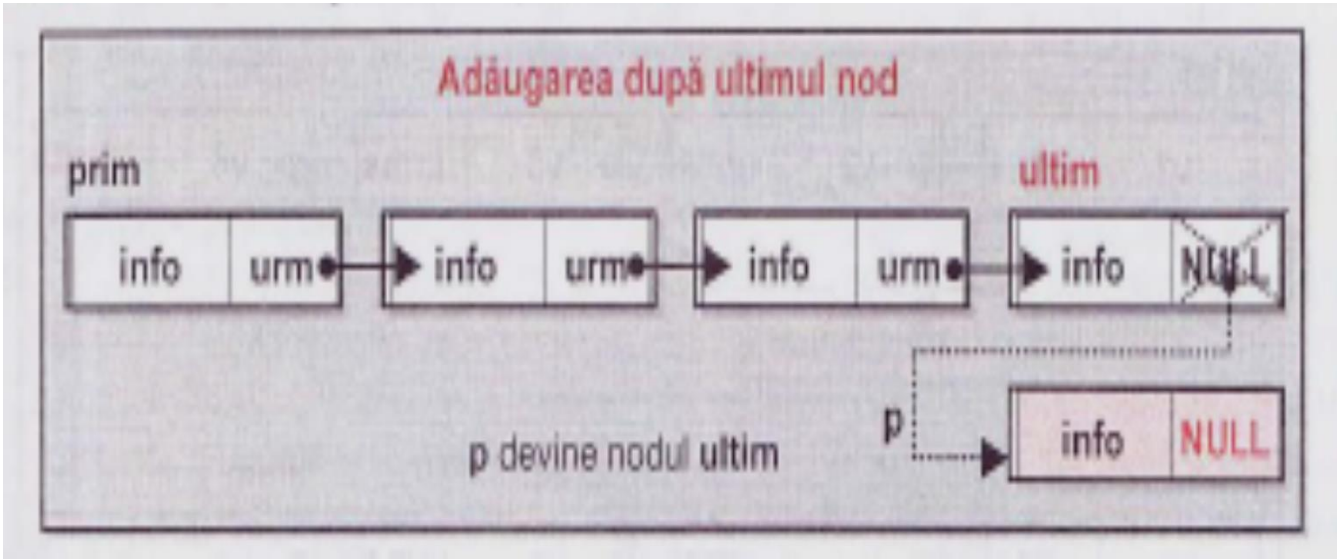
Adaugarea primului nod:

```
void adaug_prim ( adresa &prim, adresa &ultim, int n )
{ prim = aloc_mem ( );
  lista [ prim ] . info =  n;
  lista [ prim ] .urm = NULL;
  ultim = prim;
}
```

Algoritmi pentru prelucrarea listelor

Adaugarea dupa ultimul nod:

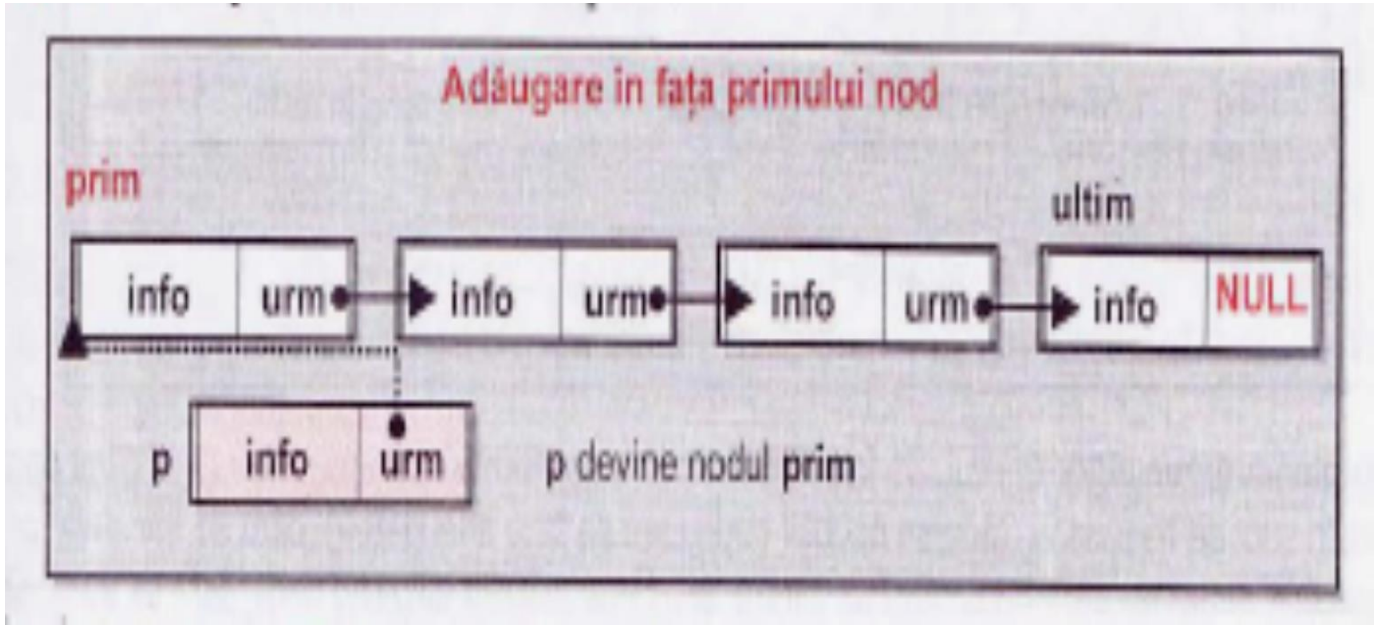
```
void adaug_dupa_ultim ( adresa &prim, adresa &ultim, int
  n )
{ p = aloc_mem ( );
  lista [ p ] . info =  n;
  lista [ p ] .urm = NULL;
  lista [ultim ].urm = p;
  if (este_vida ( prim ))   prim = p;
  ultim = p;
}
```



Algoritmi pentru prelucrarea listelor

Adaugarea in fata primului nod:

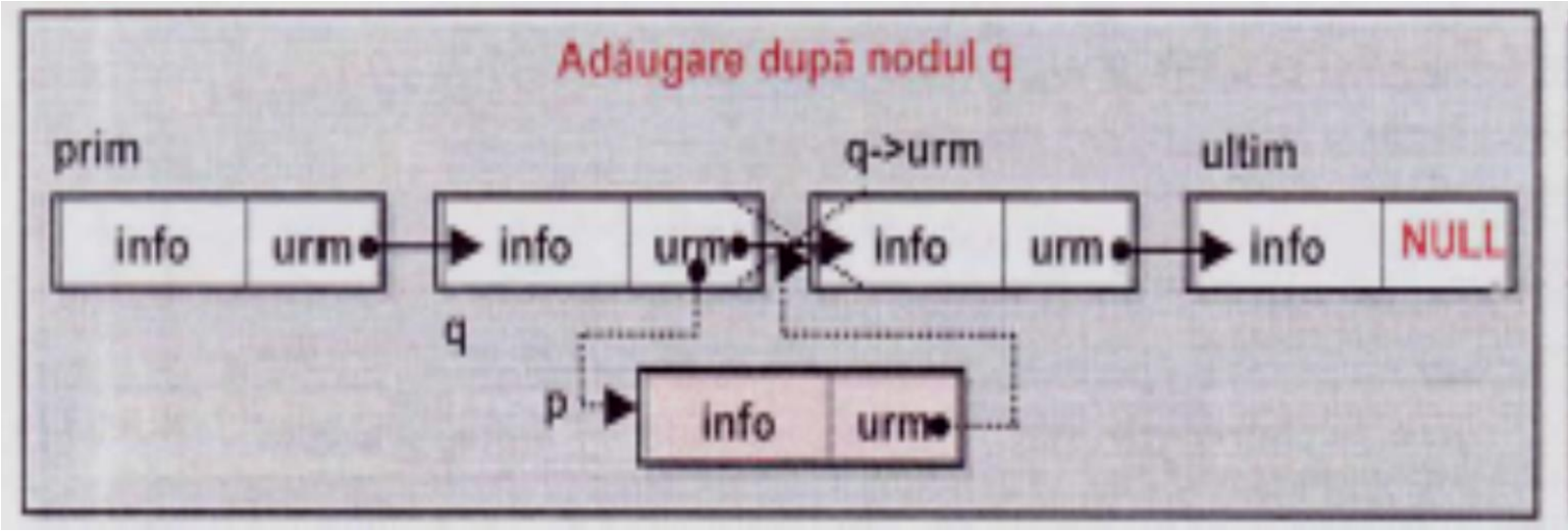
```
void adaug_inainte_prim ( adresa &prim, adresa
  &ultim, int n )
{ p = aloc_mem ( );
  lista [ p ] . info =  n;
  lista [ p ] .urm = prim;

if (este_vida ( prim ))   ultim = p;
  prim = p;  }
```



Algoritmi pentru prelucrarea listelor

Adaugarea in interiorul liste: a) Dupa nodul q

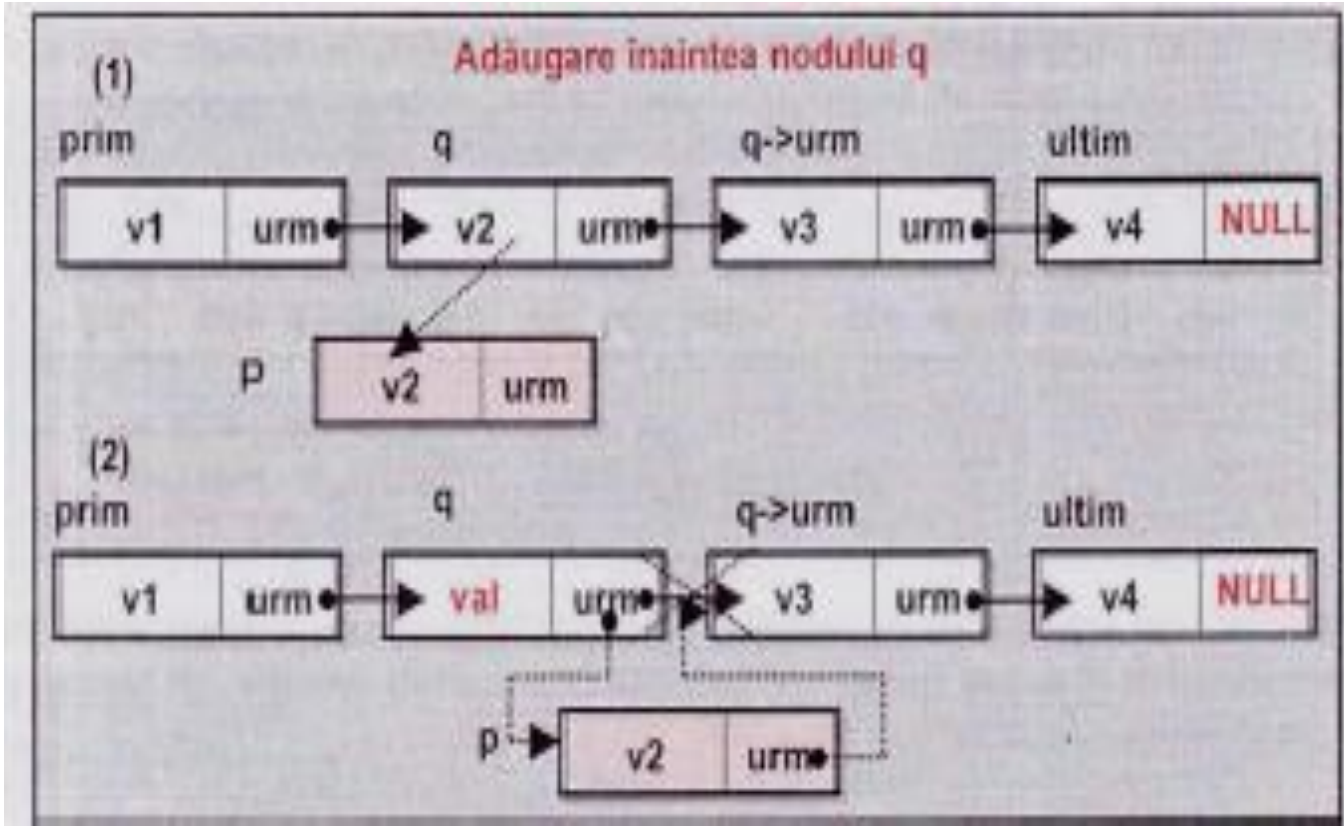
```
void adaug_dupa (int info_nod_q,   int info_nod_nou )
{
//aflu nod q
p = aloc_mem ( );
  lista [ p ] . info =  n;
  lista [ p ] .urm = lista [ q ].urm;
  lista [ q ] .urm = p;
  if ( lista [ p ] .urm = = NULL )   ultim = p;     }
```



Algoritmi pentru prelucrarea listelor

Adaugarea in interiorul liste: b) Inainte de nodul q

```
void adaug_in_fata (int info_nod_q,   int info_nod_nou )
{ //aflam nod q
  p = aloc_mem ( );
  lista [ p ] . info =  lista [ q ] . info;
  lista [ q ] . info = n;
  lista [ p ] .urm = lista [ q ].urm;
  lista [ q ] .urm = p;
  if ( lista [ p ] .urm = = NULL )
  ultim = p;   }
```



Algoritmi pentru prelucrarea listelor

Parcurgerea listei:

```
void parcurge ( adresa prim )
{ for ( p = prim; p ! = NULL; p = lista [ p ] . urm )
    // se prelucreaza   lista [ p ] . info;
```

Exercitiu: Eliminarea unui element -> laborator

Alte probleme cu liste – EXERCIȚII laborator

1. Să se creeze o listă cu numere întregi folosind crearea prin adăugarea elementelor la inceputul listei (stiva).

Se cere:

a) Să se afișeze conținutul listei

b) Să se determine suma și maximul elementelor listei

- Exemplu:

Pentru n=4 și elementele {-3, 10, -7, 5}, se obtine suma = 5 si maximul = 10

Pas 1: Stabilim care sunt datele de intrare, împreună cu datele de ieşire.

- Date de intrare: lista cu n elemente
- Date de ieşire: suma și maximul

Pas 2: Analiza problemei

1. Construirea listei(stiva) cu n elemente
2. Afișarea listei
3. Parcurgerea listei si adunarea elementelor
4. Parcurgerea listei și aflarea maximului
5. Afișarea sumei și a maximului

Gândim problema pe subprograme ce pot fi apelate.

1. Realizăm o metodă pentru crearea listei.
2. Realizăm o metodă care returnează suma și o metodă care returnează maximul.
3. Realizăm o metodă de afișare a listei.

1. Crearea listei - dinamic utilizând pointeri

```
struct lista
{  int info;
   lista *urm;
};

lista *p, *prim, *ultim;  //nod curent, nod prim , nod ultim

void creare(lista *&prim, lista *&ultim)   {
```

//avem nevoie de referință la pointer pentru a pointa către următoarea adresă din memorie, altfel rămâne null

```
citeste n    //se citește n – numărul de elemente dorit
citeste inf   //informația

//se adaugă primul element și se atribuie prima informație acestuia
prim=new lista  //se alocă memorie
prim ->info = inf
prim->urm = NULL   //primul nod va fi și ultimul
ultim=prim
// se citesc celelalte informații pentru celelalte noduri
pentru i = 2 .. n
            citeste inf
            p = new lista  //nodul curent considerat noul nod de introdus în listă
            p->info=inf
            p->urm = prim
            prim =p
}
```

2. Afișarea listei

void afisare (lista *prim) {

p, nodul curent este nodul prim

cat timp nodul_current (p) nu este NULL

scrie p->info

p=p->urm }

3. Calcul suma și maximul

int suma (lista *prim) {

suma=0

p, nodul curent este nodul prim

cat timp p nu este NULL

suma+=p->info

trec la urmatorul nod

return suma }

```
int maxim (lista *prim)   {
        p, nodul curent este nodul prim
        maxim = informatia primului nod
        cat timp nodul curent diferit de null
                daca p->info > maxim
                        maxim = p->info
                p=p->urm  //se trece la următorul nod
        return maxim        }
```

2. Să se creeze o listă cu numere întregi folosind crearea prin adăugare la sfârșitul listei. Se cere:

a) Să se afișeze conținutul listei

b) Să se determine numărul de numere prime

- Exemplu:

Pentru n=4 și elementele {11, 10, 13, 5}, se obțin 3 numere prime

Pas 1: Stabilim care sunt datele de intrare, împreună cu datele de ieşire.

- Date de intrare: lista cu n elemente
- Date de ieşire: numărul de numere prime

1. Crearea listei - dinamic utilizând pointeri

```
struct lista
{  int info;
   lista *urm;
};

lista *p, *prim, *ultim;  //nod curent, nodul prim , nod ultim

void creare(lista *&prim, lista *&ultim)   {
```

//avem nevoie de referință la pointer pentru a pointa către următoarea adresă din memorie, altfel rămâne null

```
citeste n    //se citește n – numărul de elemente dorit
citeste inf   //informația

//se adaugă primul element și se atribuie prima informație acestuia
prim=new lista  //se alocă memorie
prim ->info = inf
prim->urm = NULL   //primul nod va fi și ultimul
ultim=prim
// se citesc celelalte informații pentru celelalte noduri
pentru i = 2 .. n
            citeste inf
            p = new lista  //nodul curent considerat noul nod de introdus in listă
            p->info=inf
            p->urm = NULL  //se adaugă la sfârșitul listei
            ultim->urm = p
            ultim =p
}
```

2. Afișarea listei

void afisare (lista *prim) {

p, nodul curent este nodul prim

cat timp nodul_current (p) nu este NULL

scrie p->info

p=p->urm }

3. Verific dacă un număr este număr prim

int verific_prim (int nr) {

int flag = 1

pentru i = 2 .. nr/2

daca nr%i =0 flag=0

return flag } //dacă flag rămâne 1 atunci numărul nu este prim, altfel return 0

4. Verific dacă elementele din listă sunt numere prime

```
int prime (lista *prim)     {
                p, nodul curent este nodul prim
                contor = 0   //pentru a număr numerele prime
                cat timp nodul curent diferit de null
                        daca verific_prim (p->info ) = 1
                                contor ++
                        p=p->urm  //se trece la următorul nod
                return contor
        }
```

3.Să se creeze o listă cu primii n termeni din sirul lui Fibonacci. Se cere:

- **Exemplu: Pentru n=7 se obtine lista cu elementele 1, 1, 2, 3, 5, 8, 13**.

Pas 1: Stabilim care sunt datele de intrare, împreună cu datele de ieşire.

- Date de intrare: lista cu **primele două elemente** având informația 1
- Date de ieşire: celelalte elemente din șirul lui Fibonacci

1. Crearea listei - dinamic utilizând pointeri

```
struct lista
{  int info;
   lista *urm;
};

lista *p, *prim, *ultim;  //nod curent, nod prim , nod ultim

void creare(lista *&prim, lista *&ultim, int numar_elem_fibonacci)   {
```

//avem nevoie de referință la pointer pentru a pointa către următoarea adresă din memorie, altfel rămâne null

prim=new lista // se alocă memorie
prim ->info = 1 // valoarea 1 la primul element din lista
prim->urm = NULL   //primul nod va fi și ultimul
p=new lista // avem nevoie de un al doilea nod
p->info = 1
p->urm=NULL
prim->urm=p // primul nod pointează către p
ultim=p //acum avem două noduri
// se citesc celelalte informații pentru celelalte noduri
i=2, j=1, k=1 // pentru a calcula valoarea pentru nodul trei avem nevoie de doua valori 1, iar în i se va calcula elem Fibonacci
cât timp numar <= numar_elem_fibonacci
        i = j + k
        p = new lista //nodul curent considerat noul nod de introdus in listă
        p->info= i
        p->urm = NULL //se adaugă la sfârșitul listei
        ultim->urm = p
        ultim =p
        j = k și k = i

11. Structura de date de tip stivă

Stiva este un tip particular de listă, pentru care atât operaţia de inserare a unui element în structură, cât şi operaţia de extragere a unui element se realizează la un singur capăt, denumit ***vârful*** stivei.

-singurul element din stivă la care avem acces direct este cel de la vârful stivei;
- trebuie cunoscut în permanenţă vârful stivei;

- stiva este utilizată atunci când programul trebuie să amâne execuţia unor operaţii, pentru a le executa ulterior, în ordinea inversă apariţiei lor;

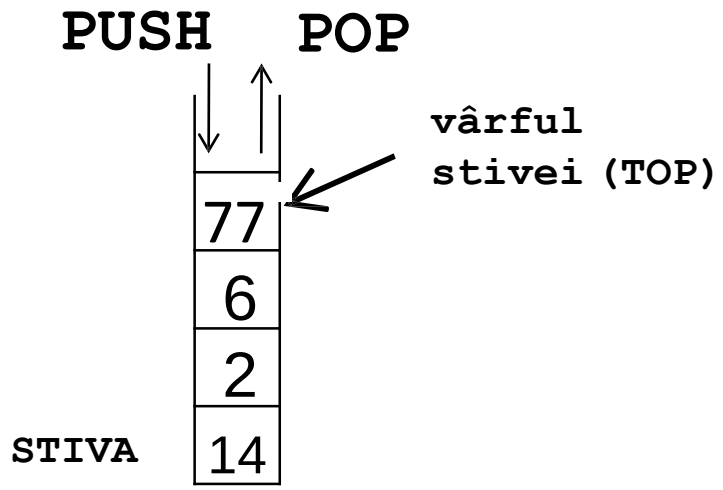
- stiva funcţionează după principiul **`LIFO`** (**`Last In First Out`**);

Operaţii elementare care se pot efectua asupra unei stive:

- crearea unei stive vide;
- înserarea unui element în stivă (**PUSH**);
- extragerea unui element din stivă (**POP**);
- accesarea elementului de la vârful stivei (**TOP**).

Exemplu

14, 2, 6, 77



Reprezentarea stivelor

Cel mai siplu mod de a **implementa o stivă** constă în memorarea elementelor sale într-un vector.

Declararea stivei:

```
<tip> <identificator>[<nr_elemente>];
```

Exemplu

```
int STIVA[11];
```

1. Crearea unei stive vide

- se iniţializează numărul de elemente din stivă cu 0;

Exemplu

```
int  vârf=0;
```

2. Inserarea unui element în stivă

- se verifică dacă stiva nu este plină;
- se măreşte vârful stivei;
- se plasează la vârf noul element;

Exemplu

```
if(varf==10)
    cout<<"Stiva este plină";
 else
 {
     varf++;
     STIVA[varf]=e;
 }
```

3. Extragerea unui element din stivă

- se verifică dacă stiva nu este vidă;
- se reţine elementul din vârful stivei într-o variabilă;
- se micşorează cu o unitate vârful stivei;

Exemplu

```
if(varf==0)
        cout<<"Stiva este vidă";
 else
 {
    e=STIVA[varf];
    varf--;
 }
```

4. Accesarea elementului din vârful stivei

- se memorează elementul din vârful stivei într-o variabilă;

Exemplu

```
e=STIVA[varf];
```

12. Structura de date de tip coadă

Coada este un tip particular de listă, pentru care operaţia de inserare a unui element se realizează la un capăt, iar operaţia de extragere a unui element se realizează la celălalt capăt.

- singurul element din coadă la care avem acces direct este cel de la început;
- trebuie cunoscut în permanenţă începutul cozii şi sfârşitul cozii;

-coada este utilizată atunci când informaţiile trebuie prelucrate exact în ordinea în care “au sosit” şi ele sunt reţinute în coadă până când pot fi prelucrate;

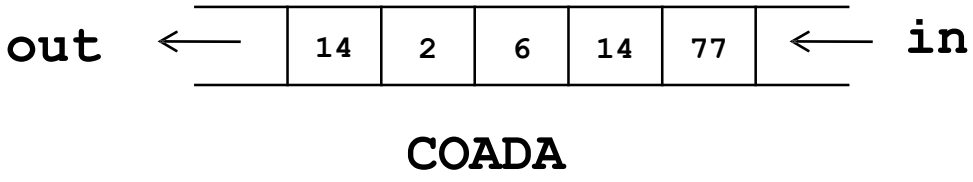
- coada funcţionează după principiul **`FIFO`** (**`First In First Out`**);

Operaţii elementare care se pot efectua asupra unei cozi:

- crearea unei cozi vide;
- înserarea unui element în coadă;
- eliminarea unui element din coadă;
- accesarea unui element din coadă.

Exemplu

`14,2,6,77`



Reprezentarea cozilor

Cel mai simplu mod de a **implementa o coadă** constă în memorarea elementelor sale într-un vector.

Declararea cozii:

```
<tip> <identificator>[<nr_elemente>];
```

- **Exemplu**

```
int COADA[11];
```

1. Crearea unei cozi vide

- se iniţializează numărul de elemente din coadă cu **0**, pentru aceasta se iniţializează variabilele **început** şi **sfârşit**;

Exemplu

```
int          început=1, sfarsit=0;
```

2. Inserarea unui element în coadă

- se verifică dacă coada nu este plină;
- se măreşte sfârşitul cozii cu o unitate;
- se plasează la sfârşit noul element;

Exemplu

```
if(sfarsit==10)
        cout<<"Coada este plină";
 else
 {
     sfarsit++;
     COADA[sfarsit]=e;
 }
```

3. Eliminarea unui element din coadă

- se verifică dacă coada nu este vidă;
- se reţine elementul de la începutul cozii într-o variabilă;
- se măreşte cu o unitate începutul cozii;

Exemplu

```
if(inceput>sfarsit)
                cout<<"Coada este vidă";
              else
              {
                   e=COADA[inceput];
                   inceput++;
              }
```

4. Accesarea unui element din coadă

- se memorează elementul de la începutul cozii într-o variabilă;

Exemplu

```
e=COADA[inceput];
```

13. Bibliografie şi webografie

1. Cerchez E., Şerban M., *Informatică. Manual pentru clasa a X-a*, Editura Polirom, Iaşi, 2000

2. http://ro.wikipedia.org/wiki/Lista_(structur%C4%83_de_date)

3. http://ro.wikipedia.org/wiki/Stiv%C4%83_(structur%C4%83_de_date)

4. http://en.wikipedia.org/wiki/Stack_(abstract_data_type)