

# **Structuri liniare**

## **Liste. Stive. Cozi**

**- Inserare, cautare, stergere -**

**Lectii de pregatire pentru Admitere**

**29 februarie 2020**



## Cuprins

- **Liste (simple, duble, circulare)**
- **Stive, Cozi (simple, speciale)**

**Subiectele vor fi abordate atat din perspectiva alocarii  
statice cat si a alocarii dinamice!**

## Structuri liniare (Liste. Stive. Cozi)

**Structura liniara:** relatie de ordine totala pe multimea elementelor (eventual cu exceptia **primului** si **ultimului** element, fiecare element are exact un element **precedent** si exact un element **succesor**).

Exemple de structuri **liniare**: liste (contine date omogene), stive, cozi

Exemple de structuri **neliniare**:

- arbori
- elemente aflate in relatie de adiacenta data de o matrice

## Structuri liniare (Liste. Stive. Cozi)

### Clasificare

#### Dupa modul de succesiune al elementelor:

- un bloc contiguu de locatii de memorie: succesul unui element este in zona imediat alaturata (liste alocate secvential)
- elementele nu sunt neaparat stocate in locatii de memorie succesive, un element retine si adresa succesului (liste înlantuite)

#### Dupa modul de efectuare al operatiilor de intrare (inserari) si de iesire (stergeri):

- structuri liniare fara restrictii de intrare/iesire
- structuri liniare cu restrictii de intrare/iesire (stive si cozi)

## Structuri liniare - Liste

### Operatii de baza

**Traversarea** - operatia care acceseaza fiecare element al structurii, o singura data, in vederea procesarii (*vizitarea* elementului).

**Cautarea** - se cauta un element cu cheie data in structura (*cu* sau *fara* succes): consta dintr-o traversare - eventual incompleta a structurii, in care vizitarea revine la comparatia cu elementul cautat.

**Inserarea** - adaugarea unui nou element, cu pastrarea tipului structurii.

**Stergerea** - extragerea unui element al structurii (eventual in vederea unei procesari), cu pastrarea tipului structurii pe elementele ramase.

## **Liste liniare alocate secvential**

**Informatii de acelasi tip stocate in locatii de memorie contigue in ordinea indicilor (Nodurile se afla in pozitii succesive de memorie)**

**Avantaj:** acces direct la orice nod

**Dezavantaj:** multe deplasari la operatiile de inserare si stergere

## Liste liniare alocate secvential

### Exemple

- lista de numere intregi

3	-12	10	7	1
0	1	2	3	4

- lista de numere reale

0.3	-1.2	10	5.7	8.7	0.2	-1.5	1
-----	------	----	-----	-----	-----	------	---

- lista de caractere

A	&	*	+	@	c	M	#
---	---	---	---	---	---	---	---

### C / C++

```
int a[20];  
double b[30];  
char c[23];
```

### Pascal

#### Declarare

```
var a : array [1..20] of integer;  
var b : array [1..30] of double;  
var c : array [1..23] of char;
```

## Liste liniare alocate secvential

### C / C++

### Pascal

#### Traversare ( complexitate **O(n)** )

```
for (i = 0; i < n; i++)  
    // viziteaza a[i];
```

```
for i:= 1 to n do  
    { viziteaza a[i];}
```

#### Cutare ( liniara – complexitate **O(n)** )

```
int t = 0;  
for (i = 0; i < n; i++)  
    if (a[i] == x) t = 1;  
if (t == 0) // cautare fara succes
```

```
var t : boolean;  
t := false;  
for i:= 1 to n do  
    if (a[i] = x) then t := true;  
if (t = false) then  
    {cautare fara succes}
```



## Liste liniare alocate secvential

### Cautare liniara (componenta marcaj)

#### C / C++

```
int poz = 0, val;  
  
a[n] = val;  
while (a[poz] != val)  
    {  
        poz++;  
    }  
if (poz == n)  
    // cautare fara succes
```

#### Pascal

```
var val, poz: integer;  
poz := 1;  
  
a[n+1] := val;  
while (a[poz] <> val) do  
    poz := poz + 1;  
  
if (poz = n + 1) then  
    { cautare fara succes}
```

Numarul de comparatii:  $n + 1 + 1$

## Liste liniare alocate secvential

### Cautare binara (! pe vector ordonat) - $O(\log_2 n)$

#### C / C++

```
int l = 0, r = n-1, m, poz = -1;

m = (l+r) / 2;
while ((l <= r) && (val != a[m]))
{
    if (val < a[m]) r = m-1;
    else l = m+1;
    m = (l+r) / 2;
}

if (a[m] == val) poz = m;
```

#### Pascal

```
var l, r, m, poz: integer;
l := 1; r := n; poz := 0;

m := (l+r) div 2;
while (l <= r) and (val <> a[m]) do
begin
    if (val < a[m]) then r := m-1
    else l := m+1;
    m := (l+r) div 2;
end;

if (a[m] = val) then poz := m;
```

## Liste liniare alocate secvential

### Cautare binara (! pe vector ordonat) - $O(\log_2 n)$

#### Complexitate

Consideram cazul cel mai defavorabil (cautare fara succes)

**Notatie:**  $C(n)$  = numar de comparatii

- dupa o comparatie – cautarea se face pe un vector de lungime injumatatita
- in final avem un segment de un element

$$2^{C(n)} > n > 2^{C(n)-1} \Rightarrow C(n) < \log_2 n + 1 \Rightarrow \mathbf{C(n) = O(\log_2 n)}$$

## Liste liniare alocate secvential

### C / C++

### Pascal

#### **Inserare** (valoare **val** pe pozitia **poz**)

```
for (i = n-1; i >= poz; i--)  
    a[i+1] = a[i];  
a[poz] = val;  
n++;
```

```
for i:= n downto poz do  
    a[i+1] := a[i];  
a[poz]:=val;  
n := n+1;
```

#### **Stergere** (valoare de pe pozitia **poz**)

```
for (i = poz; i < n-1; i++)  
    a[i] = a[i+1];  
n--;
```

```
for i := poz to n-1 do  
    a[i] := a[i+1];  
n:=n-1;
```

## Liste liniare alocate secvential

### Structuri liniare cu restrictii la i/o: Stiva (LIFO)

- **LIFO (Last In First Out)**: ultimul element introdus este primul extras
- locul unic pentru inserari/stergeri: varf (**Top**)
- **Push (Val)** - inserarea valorii *Val* in stiva (**Stack**)
  - **Overflow (supradepasire)** - inserare in stiva plina
- **Pop(X)** - stergerea/extragerea din stiva (**Stack**) a unei valori care se depune in **X**
  - **Underflow (subdepasire)** - extragere din stiva goala

## Liste liniare alocate secvential

### Structuri liniare cu restrictii la i/o: **Stiva (LIFO)**

### Exemplificarea mecanismului RECURSIVITATII și ordinea efectuării operațiilor

$$n! = \begin{cases} 1, & \text{dacă } n=0 \\ n*(n-1)!, & \text{dacă } n \geq 1 \end{cases}$$

4! = 4*3!	= 4 * 6 = 24	↑ <b>Adâncimea recursivității</b>
3! = 3*2!	= 3 * 2 = 6	
2! = 2*1!	= 2 * 1 = 2	
1! = 1*0!	= 1 * 1 = 1	
0! = 1		

#### C / C++

```
long factorial(int n)
{
    if (n==0) return 1; //conditia de oprire
    return n*factorial(n-1); //recursivitate
}
```

#### Pascal

```
function factorial(n: integer): longint
begin
    if (n=0) then
        fact := 1 {conditia de oprire}
    else
        fact := n*factorial(n-1)
        {recursivitate}
end;
```

## Liste liniare alocate secvential

### Structuri liniare cu restrictii la i/o: **Stiva (LIFO)**

### Exemplificarea mecanismului RECURSIVITATII și ordinea efectuării operațiilor

Ce se întâmplă în stivă pentru apelul  $t = \text{factorial}(4)$  ?

STIVĂ

Se salvează un context de apel:

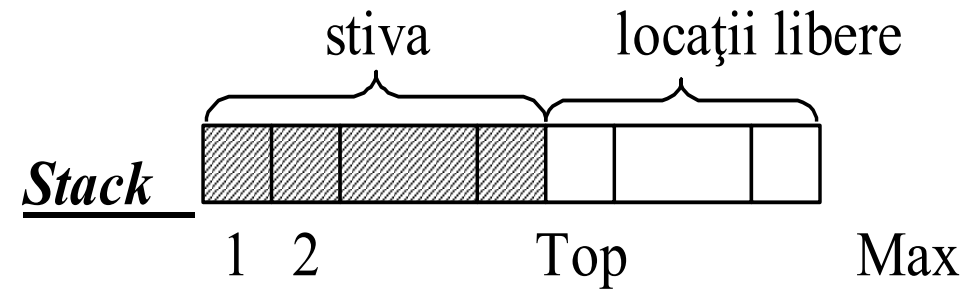
- 1.adresa de revenire
- 2.copii ale valorilor parametrilor efectivii
- 3.valorile variabilelor locale
- 4.copii ale regiștrilor
- 5.valoarea returnată

$A_5$	0	-	-	1
$A_4$	1	-	-	1
$A_3$	2	-	-	2
$A_2$	3	-	-	6
$A_1$	4	-	-	24

**$A_1$  = adresa de revenire pentru apelul  $\text{factorial}(4)$**

## Stiva in alocare statica

### Implementare



### Declarare

#### C / C++

```
#define MAX 100
```

```
int Stack[MAX];  
int Top = -1;
```

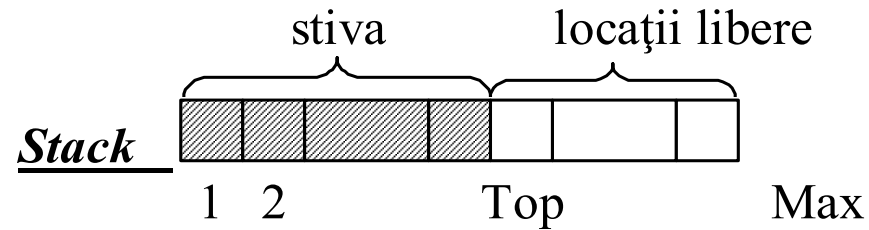
#### Pascal

```
var MAX: integer;  
Stack : array [1..100] of integer;  
Top:integer;  
Top := 0;  
MAX := 100;
```



## Stiva in alocare statica

### Implementare



### C / C++

```
void Push (int Val)
{
    if (Top == MAX-1)
        // Overflow
    else
    {
        Top++;
        Stack[Top] = Val;
    }
}
```

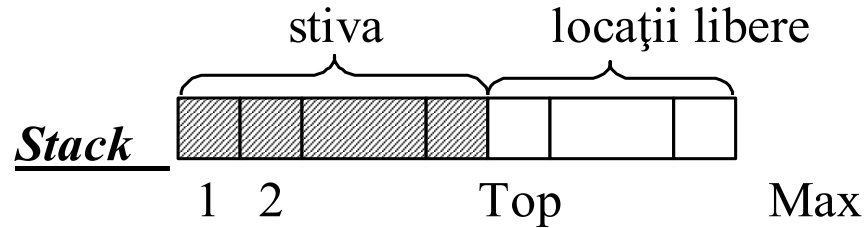
### Inserare

### Pascal

```
procedure Push (Val : integer);
begin
    if (Top = MAX) then
        // Overflow
    else
    begin
        Top := Top + 1;
        Stack[Top] := Val;
    end;
end;
```

## Stiva in alocare statica

### Implementare



### C / C++

### Stergere

### Pascal

```
void Pop (int &X)
{
    if (Top == -1)
        // Underflow
    else
    {
        X = Stack[Top];
        Top--;
    }
}
```

```
procedure Pop (var X:integer);
begin
    if (Top = 0) then
        // Underflow
    else
    begin
        X := Stack[Top];
        Top := Top - 1;
    end;
end;
```

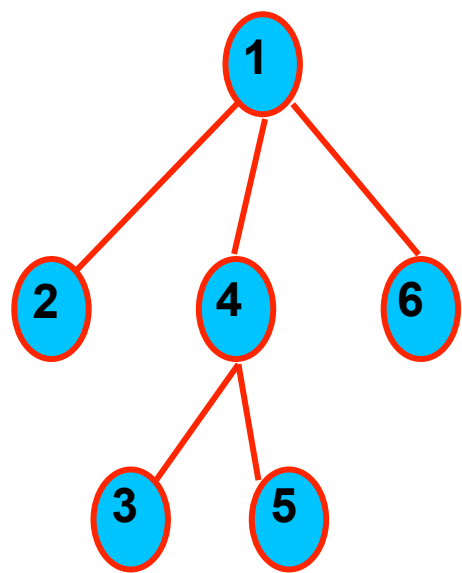
## Liste liniare alocate secvential

### Structuri liniare cu restrictii la i/o: Coada (FIFO)

- **FIFO ( First In First Out )**: primul introdus este primul extras
- capat pentru inserari: sfarsit (*Rear*)
- capat pentru stergeri: inceput (*Front*)
- **Push (Val)** - inserarea in coada (*Queue*)
  - **Overflow (supradepasire)** - inserare in coada plina
- **Pop(X)** - stergerea/extragerea din coada (*Queue*) a unei valori care se depune in *X*
  - **Underflow (subdepasire)** - extragere din coada goala

Structuri liniare cu restrictii la i/o: **Coada (FIFO)**

Exemplificarea operatiilor pe coada in parcurgerea unui arbore pe nivele (Breadth First)



BF: 1, 2, 4, 6, 3, 5.

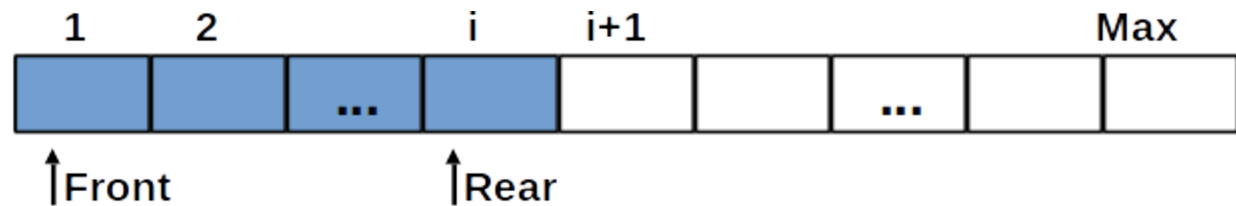
PUSH 1	1									
POP										Afis: 1
PUSH 2	2									
PUSH 4	2	4								
PUSH 6	2	4	6							
POP			4	6						Afis: 2
POP				6						Afis: 4
PUSH 3				6	3					
PUSH 5				6	3	5				
POP					3	5				Afis: 6
POP						5				Afis: 3
POP										Afis: 5

**Coada vida**

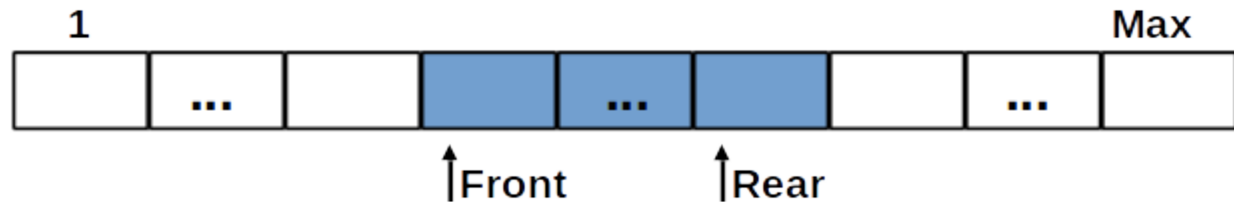
## Coada in alocare statica

### Implementare

(a) Queue



(b) Queue



### C / C++

```
#define MAX 100  
  
int Queue[MAX];  
int Front, Rear;  
Front = Rear = -1;
```

### Declarare

### Pascal

```
var MAX: integer;  
Queue : array [1..100] of integer;  
Front, Rear :integer;  
MAX := 100;  
Front := 0; Rear := 0;
```

## Coada in alocare statica

### Implementare

#### C / C++

```
void Push (int Val)
{
    if (Rear == MAX-1)
        // Overflow

    else
    {if (Rear == -1)
        //coada initial vida
        Front++;
        Rear++;
        Queue[Rear] = Val;}
}
```

#### Inserare

#### Pascal

```
procedure Push (Val : integer);
begin
    if (Rear = MAX) then
        // Overflow

    else
    begin
        if (Rear = 0) then
            // coada initial vida
            Front := Front + 1;
            Rear := Rear + 1;
            Queue[Rear] := Val;
        end;
    end;
```

## Coadă în alocare statică

### Implementare

#### C / C++

```
void Pop (int &X)
{
    if (Front == MAX)
        // Underflow
    else {
        if (Front == -1 || Front > Rear)
            // Coadă vidă
        else
        {
            X = Queue[Front];
            Front++;
        }
    }
}
```

#### Stergere

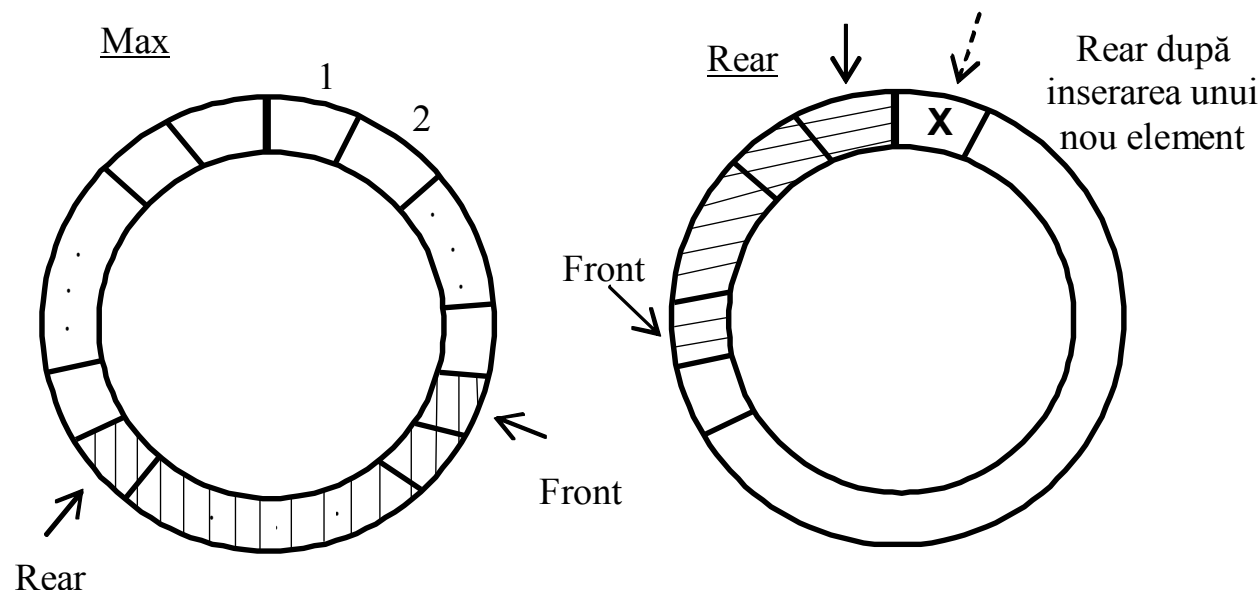
#### Pascal

```
procedure Pop (var X:integer);
begin
    if (Front = MAX) then
        // Underflow
    else
        begin
            if (Front = 0 OR Front > Rear)
                // Coadă vidă
            else begin
                X := Queue[Front];
                Front := Front + 1;
            end;
        end;
end;
```

## Liste liniare alocate secvential

### Structuri liniare cu restrictii la i/o: **Alte tipuri de cozi**

#### Coada circulara (in alocare statica)



Pe coada circulara: **aritmetica (mod Max)** la incrementarea indicilor

Coada vidă:  **$Front = Rear = 0$** .

Coada plină (pe versiunea circulară):  **$Rear + 1 = Front \pmod{Max}$** .

Coada cu un singur element:  **$Rear = Front \neq 0$** .



## Liste liniare alocate secvential

### Structuri liniare cu restrictii la i/o: **Alte tipuri de cozi**

#### Exemplificare utilizarii unei cozi circulare – Problema Josephus

- $n$  copii asezati in cerc sunt numarati din  $m$  in  $m$  plecand de la copilul  $k$ .
- fiecare al  $m$  – lea copil numarat iese din cerc.
- afisare ordine iesire copii din cerc

$n = 12$   
 $m = 3$   
 $k = 2;$

1	2	3	4
12			5
11			6
10	9	8	7

1	2	3	4
12			5
11			6
10	9	8	7

Afis: 2, 5, 8, 11

1		3	4
12			
			6
10	9		7

Afis: 3, 7, 12

1			4
			6
10	9		

Afis: 6, 1

1			4
			6
10	9		

Afis: 10, 4

1			4
			6
10	9		

Afis: 9

Ordine: 2,5,8,11,3,7,12,6,1,10,4,9

## **Liste liniare alocate secvential**

### **Structuri liniare cu restrictii la i/o: Alte tipuri de cozi**

#### **Cooda cu priorități - Priority Queues**

Elementele au, pe lângă cheie și o prioritate:

- cea mai înaltă prioritate este 1, urmată de 2, etc.

#### **Ordinea liniară este dată de regulile:**

- elementele cu aceeași prioritate sunt extrase (și procesate) în ordinea intrării;
- toate elementele cu prioritate  $i$  se află înaintea celor cu prioritate  $i+1$  (și deci vor fi extrase înaintea lor).

Extragerile se fac dintr-un singur capăt.

**Ca să se poată aplica regulile de mai sus la extragere, inserarea unui nou element cu prioritate  $i$  se va face la sfârșitul listei ce conține toate elementele cu prioritate  $i$ .**

## **Liste liniare alocate secvential**

### **Structuri liniare cu restrictii la i/o: Alte tipuri de cozi**

#### **DEQUE - Double Ended Queue**

- structură liniară în care inserările și ștergerile se pot face la oricare din cele două capete, dar în nici un alt loc din coadă.

În anumite tipuri de aplicații sau în modelarea anumitor probleme pot sa apara structuri de cozi cu restricții de tipul:

- inserările se pot face la un singur capăt și extragerile la amândouă.

## **Liste liniare inlantuite**

### **- alocate static si dinamic**

**Nodul** contine informatia si **indicele (adresa)** urmatorului nod

**Avantaj:** operatiile de adaugare sau stergere sunt rapide

**Dezavantaje:**

- Accesul la un nod se face prin parcurgerea nodurilor precedente**
- Indicele (adresa) nodului urmator ocupa memorie suplimentara**

## Liste liniare inlantuite alocate static

### C / C++

```
struct nod {
    int inf, urm;
};

nod a[100];
int n, prim, ultim;
int oc[100];
    // 0 - liber, 1 - ocupat
prim = ultim = -1;
```

### Declarare

### Pascal

```
nod = record
    inf: integer;
    urm: integer;
end;

var a: array[1..100] of
    nod;
    n, prim, ultim:
    integer;
    oc: array[1..100] of integer;
        {0 - liber, 1 - ocupat}
    prim := 0; ultim := 0;
```

**n = 7**  
**prim = 6**  
**ultim = 4**

a	10	11	22	40	65	38	77			
	3	7	5	0	2	1	4			
oc	1	1	1	1	1	1	1	0	0	0
	1	2	3	4	5	6	7	8	9	10

**Ordine: a[6], a[1], a[3], a[5], a[2], a[7], a[4]**

## Liste liniare inlantuite alocate static

### C / C++

### Pascal

#### Alocare

```
i = 0;  
while (oc[i] != 0) i++;  
oc[i] = 1;  
n++;
```

```
i := 1;  
while (oc[i]<>0) do i := i+1;  
oc[i] := 1;  
n := n+1;
```

#### Existenta spatiu de memorare

```
if (n<99)  
    //exista  
else  
    // nu exista
```

```
if (n<100) then  
    { exista }  
else  
    { nu exista }
```

#### Eliberare

```
// eliberare pozitie x  
oc[x] = 0;  
n--;
```

```
{eliberare pozitie x}  
oc[x]:=0;  
n:=n-1;
```

## Liste liniare inlantuite alocate static

C / C++

Inserare

Pascal

***Inserarea valorii “val” la sfarsitul listei***

ultim = 4

a	10	11	22	40	65	38	77			
	3	7	5	0	2	1	4			
oc	1	1	1	1	1	1	1	0	0	0
	1	2	3	4	5	6	7	8	9	10

***Exemplu val = 100***

nou = 8  
a[8].inf = 100  
a[8].urm = 0  
ultim = 8

a	10	11	22	40	65	38	77	100		
	3	7	5	8	2	1	4	0		
oc	1	1	1	1	1	1	1	1	0	0
	1	2	3	4	5	6	7	8	9	10

## Liste liniare inlantuite alocate static

### C / C++

### Inserare

### Pascal

#### *Inserarea valorii "val" la sfarsitul listei*

```
int nou;
if (prim== -1)
{  alocare(prim);
  a[prim].inf = val;
  a[prim].urm = 0;
  ultim = prim; }
else if (n<99)
{  alocare(nou);
  a[ultim].urm = nou;
  a[nou].inf = val;
  a[nou].urm = 0;
  ultim = nou;  }
else cout<<"lipsa spatiu";
```

```
var nou: integer;
if (prim=0) then begin
  alocare(prim);
  a[prim].inf := val;
  a[prim].urm := 0;
  ultim := prim
end
else if (n<100) then
begin
  alocare(nou);
  a[ultim].urm := nou;
  a[nou].inf := val;
  a[nou].urm := 0;
  ultim := nou
end
else write('lipsa spatiu');
```



## Liste liniare inlantuite alocate static

C / C++

Inserare

Pascal

***Inserarea valorii “val\_ins” dupa valoarea “val”***

a[3].inf = 22  
a[3].urm = 5

a	10	11	22	40	65	38	77			
	3	7	5	0	2	1	4			
oc	1	1	1	1	1	1	1	0	0	0

***Exemplu val = 100 dupa valoarea 22 (care se gaseste pe pozitia 3)***

p = 3  
nou = 8  
a[8].inf = 100  
a[8].urm = 5  
a[3].urm = 8

a	10	11	22	40	65	38	77	100		
	3	7	8	0	2	1	4	5		
oc	1	1	1	1	1	1	1	1	0	0

## Liste liniare inlantuite alocate static

### C / C++

### Inserare

### Pascal

***Inserarea valorii “val\_ins” dupa valoarea “val”***

```
int p, nou;
if (n<99)
{
    p = prim;
    while(a[p].inf != val)
        p = a[p].urm;
    alocare(nou);
    a[nou].inf = val_ins;
    a[nou].urm = a[p].urm;
    a[p].urm = nou;
    if (a[nou].urm == 0)
        ultim = nou;
}
else cout<<"lipsa spatiu";
```

```
var p, nou: integer;
if (n<100) then
begin
    p := prim;
    while(a[p].inf <> val)
        p = a[p].urm;
    alocare(nou);
    a[nou].inf := val_ins;
    a[nou].urm := a[p].urm;
    a[p].urm := nou;
    if (a[nou].urm = 0)
        ultim := nou;
end
else write('lipsa spatiu');
```

## Liste liniare inlantuite alocate static

C / C++

Inserare

Pascal

***Inserarea valorii “val\_ins” inaintea valorii “val”***

a[3].inf = 22  
a[3].urm = 5

<b>a</b>	10	11	22	40	65	38	77			
	3	7	5	0	2	1	4			
<b>oc</b>	1	1	1	1	1	1	1	0	0	0

***Exemplu val = 100 inaintea valorii 22***

Precedentul valorii

22 = pozitia p

p = 1

nou = 8

a[8].inf = 100

a[8].urm = 3

a[2].urm = 8

<b>a</b>	10	11	22	40	65	38	77	100		
	8	7	5	0	2	1	4	3		
<b>oc</b>	1	1	1	1	1	1	1	1	0	0

## Liste liniare inlantuite alocate static

### C / C++

### Inserare

### Pascal

#### *Inserarea valorii “val\_ins” inaintea valorii “val”*

```
int p, nou;
if (n<99)
    if (a[prim].inf == val) {
        alocare(nou);
        a[nou].inf = val_ins;
        a[nou].urm = prim;
        prim = nou; }
    else {
        p = prim;
        while(a[a[p].urm].inf != val)
            p = a[p].urm;
        alocare(nou);
        a[nou].inf = val_ins;
        a[nou].urm = a[p].urm;
        a[p].urm = nou; }
else cout<<"lipsa spatiu";
```

```
var p, nou: integer;
if (n<100) then
    if (a[prim].inf = val) then
        begin
            alocare(nou);
            a[nou].inf := val_ins;
            a[nou].urm := prim;
            prim := nou
        end
    else begin
        p := prim;
        while(a[a[p].urm].inf <> val)
            p = a[p].urm;
        alocare(nou);
        a[nou].inf := val_ins;
        a[nou].urm := a[p].urm;
        a[p].urm := nou;
        end
    else write('lipsa spatiu');
```

## Liste liniare inlantuite alocate static

C / C++

Stergere

Pascal

### Stergerea valorii "val" din lista

a[3].inf = 22  
a[3].urm = 5

a	10	11	22	40	65	38	77			
	3	7	5	0	2	1	4			
oc	1	1	1	1	1	1	1	0	0	0

### Exemplu val = 22

Precedentul valorii

22 = pozitia p

p = 1

aux = 3

a[8].inf = 100

a[8].urm = 2

a[2].urm = 8

a	10	11	22	40	65	38	77			
	5	7	5	0	2	1	4			
oc	1	1	0	1	1	1	1	0	0	0

aux

3

## Liste liniare inlantuite alocate static

### C / C++

### Stergere

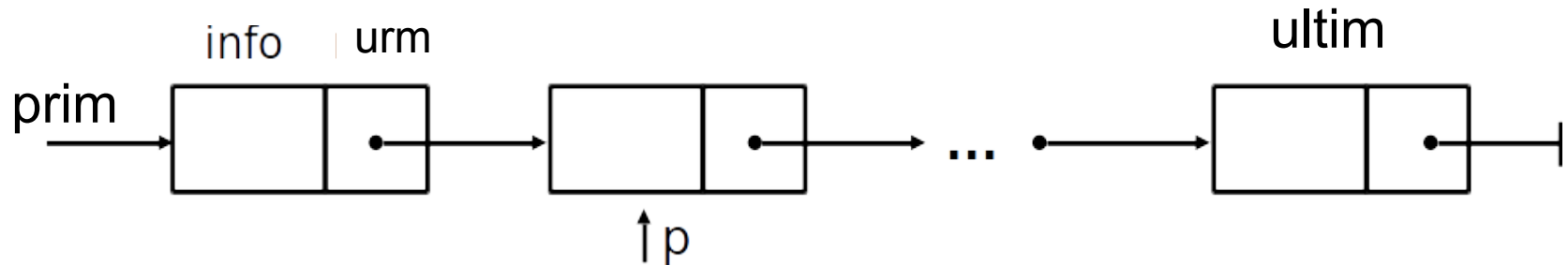
### Pascal

#### ***Stergerea valorii “val” din lista***

```
int p, aux;
if (a[prim].inf == val)
{
    aux = prim;
    prim = a[prim].urm;
}
else
{
    p = prim;
    while(a[a[p].urm].inf != val)
        p = a[p].urm;
    aux = a[p].urm;
    a[p].urm = a[aux].urm;
    if (aux == ultim)
        ultim = p;
}
eliberare(aux);
```

```
var p, aux: integer;
if (a[prim].inf = val) then begin
    aux := prim;
    prim := a[prim].urm;
end
else begin
    p := prim;
    while(a[a[p].urm].inf <> val)
        p := a[p].urm;
    aux := a[p].urm;
    a[p].urm := a[aux].urm;
    if (aux = ultim)
        ultim := p;
}
eliberare(aux);
```

## Liste liniare inlantuite alocate dinamic



- ***prim*** retine adresa primului nod din lista, iar ***ultim*** retine adresa sfarsitului listei;
- fiecare nod conține:

1) un câmp, pe care se reprezintă un element al mulțimii;  
**în algoritmi care urmează putem presupune că elementul ocupă un singur câmp, *info*;**

(2) un pointer către nodul următor, *urm*.

## Liste simplu inlantuite

### C / C++

### Pascal

#### Declarare

```
struct nod{  
    int info;  
    nod *urm;  
};  
nod *prim = NULL, *ultim;
```

```
type pnod = ^nod;  
    nod = record  
        inf :integer;  
        urm :pnod;  
    end;  
var prim, ultim : pnod;  
    prim := nil;
```

#### Traversare

```
nod *p;  
p = prim;  
while (p != NULL)  
{  
    // prelucrare p → info  
    p = p → urm;  
}
```

```
var p: pnod;  
p := prim;  
while (p <> nil) do  
    begin  
        {prelucrare p^.info}  
        p := p^.urm;  
    end
```



## Liste simplu inlantuite

### C / C++

```
nod *p;  
int x;  
  
p = prim;  
while (p != NULL && x != p->info)  
    p = p -> urm;  
  
if (p == NULL) // negasit;  
    else // gasit in p
```

### Pascal

### Cautare

```
var p: pnod;  
int x;  
  
p := prim;  
while (p <> nil) and (x <> p^.info) do  
    p := p^.urm;  
  
if (p = nil) then {negasit}  
    else {gasit in p}
```

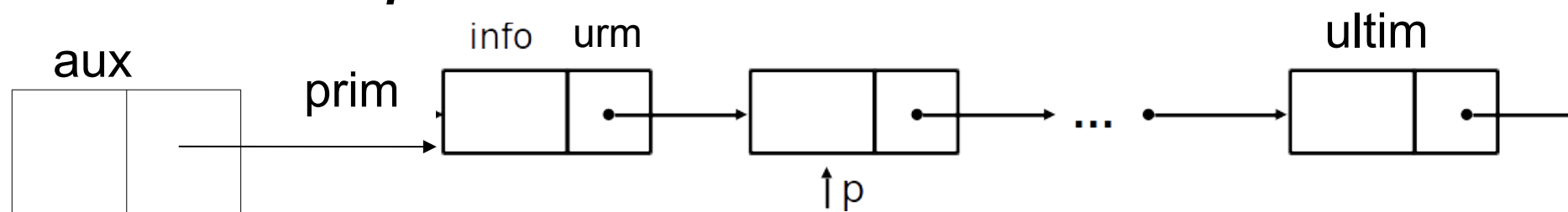
## Liste simplu inlantuite

C / C++

**Inserare**

Pascal

***Inserarea la inceputul listei***



// aux = nodul de inserat

```
nod* aux = new nod;  
// prelucrare aux->info  
if (prim != NULL)  
    aux->urm = prim;  
else  
    aux->urm = NULL;  
prim = aux;
```

{aux = nodul de inserat}

```
new (aux);  
// prelucrare aux^.info;  
if (prim <> nil) then  
    aux^.urm := prim  
else  
    aux^.urm := nil;  
prim := aux;
```

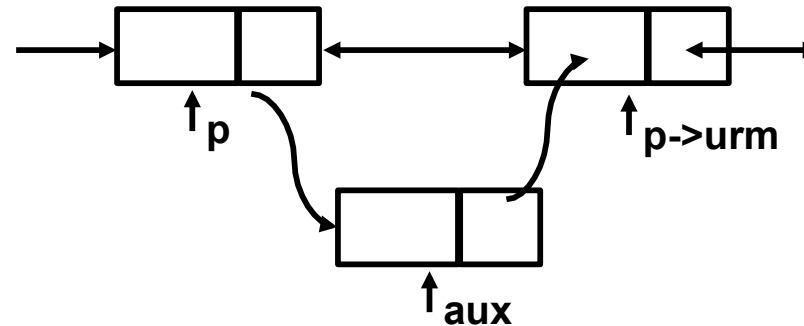
## Liste simplu inlantuite

C / C++

**Inserare**

Pascal

***Inserarea in interiorul listei***



```
nod *p, *aux;
```

```
aux = new nod;  
// prelucrare aux → info;  
aux → urm = p → urm;  
p → urm = aux;
```

```
var p, aux: pnod;
```

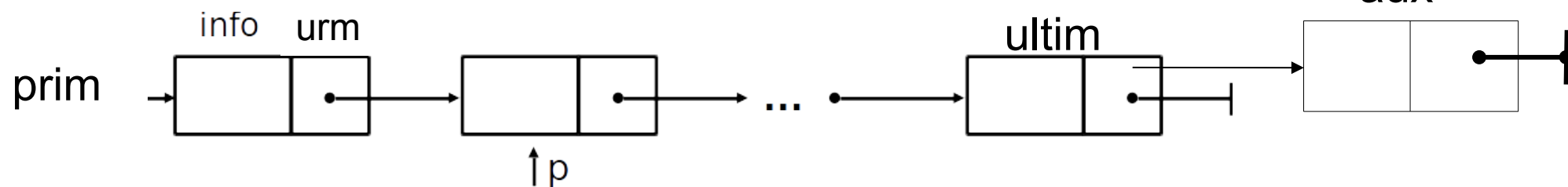
```
new (aux);  
{ prelucrare aux^.info;}
```

```
aux^.urm := p^.urm;  
p^.urm := aux;
```

## Liste simplu inlantuite

C / C++

### *Inserarea la sfarsitul listei*



```
// aux = nodul de inserat
nod* aux = new nod;
// prelucrare aux->info
aux->urm = NULL;
if (prim != NULL)
{ aux->urm = ultim;
  ultim = aux;}
else {prim = aux;
      ultim = aux; }
```

Inserare

Pascal

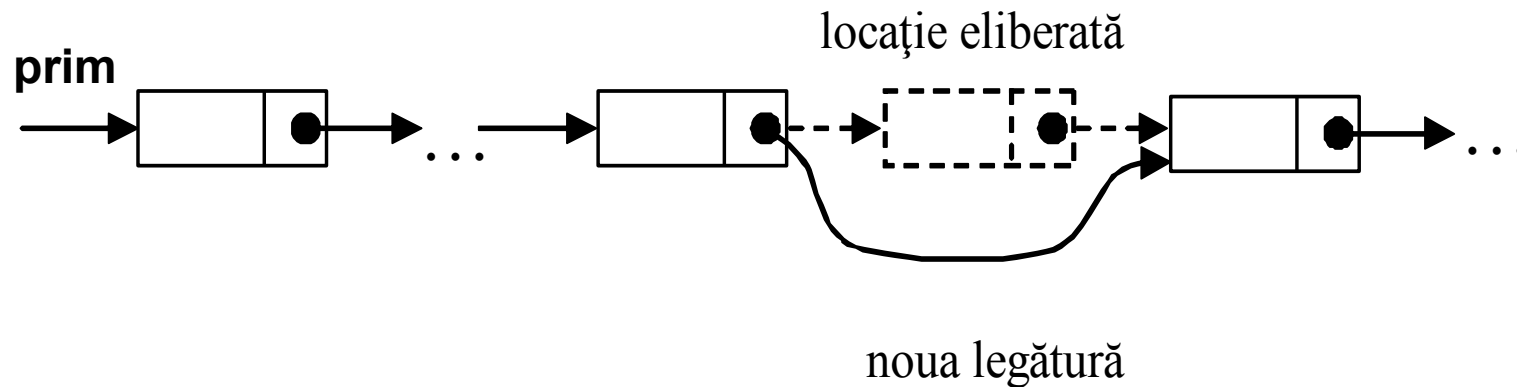
```
{aux = nodul de inserat}
new (aux);
{ prelucrare aux^.info;}
aux^.urm := nil;
if (prim <> nil) then begin
    aux^.urm := ultim;
    ultim := aux; end
else begin
    prim := aux;
    ultim := aux;
end;
```

## Liste simplu inlantuite

C / C++

**Stergere**

Pascal



Refacerea structurii de lista simplu inlantuita pe nodurile ramase

Eventual dezalocare de spatiu pentru nodul extras (sau alte operatii cu el)

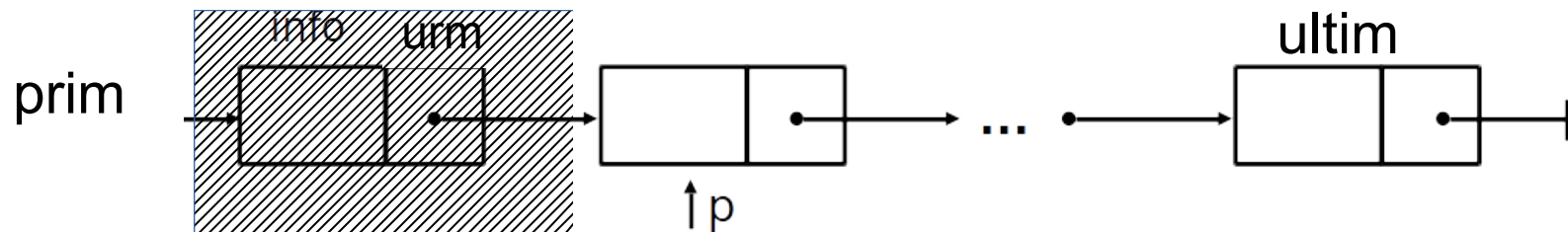
## Liste simplu inlantuite

C / C++

**Stergere**

Pascal

### *Stergerea la inceputul listei*



```
if (prim != NULL)
{
    nod *temp = prim;
    prim = prim → urm;
    delete temp;
}
```

```
temp : pnod;

if (prim <> nil) then
begin
    temp := prim;
    prim := prim ^ .urm;
    dispose (temp);
end
```

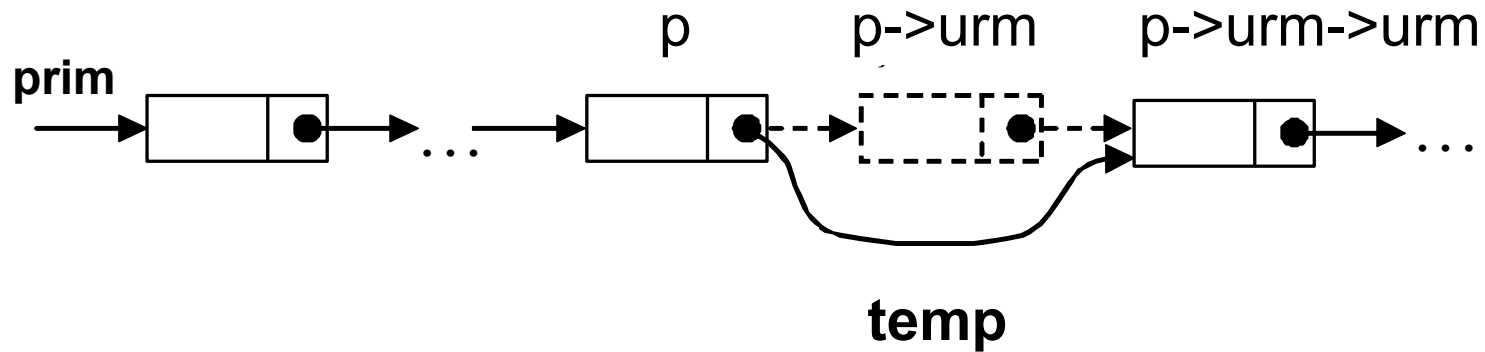
## Liste simplu inlantuite

C / C++

**Stergere**

Pascal

### *Stergerea in interiorul listei*

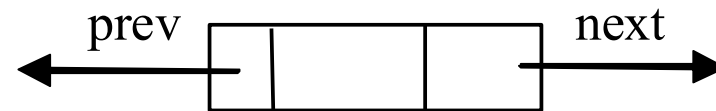


```
nod *temp = p → urm;  
p → urm = p → urm → urm;  
delete temp;
```

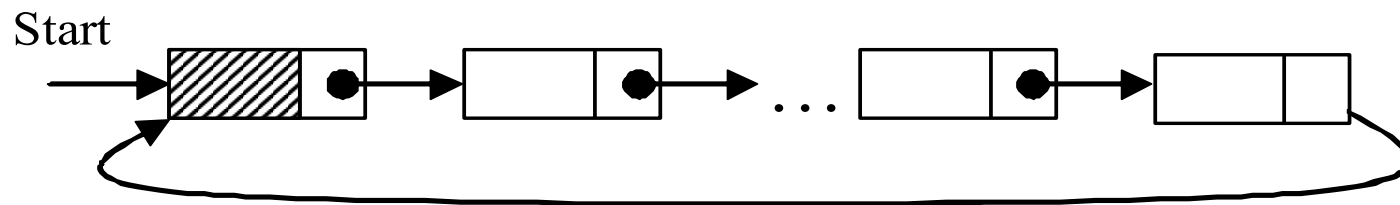
```
temp : pnod;  
temp := p^.urm;  
p^.urm := p^.urm^.urm;  
dispose (temp);
```

## Alte tipuri de liste

- cu nod marcaj
- circulare
- dublu inlantuite
- alte inlantuirii (liste de liste, masive, etc. )



Nod într-o listă dublu înlănțuită.



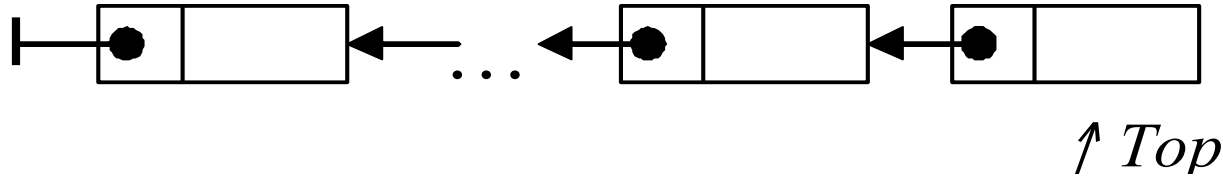
Listă circulară cu nod marcaj.



## Stiva in alocare dinamica

C / C++

Pascal



```
struct nod {  
    int info;  
    nod *urm;  
};  
  
nod * Top = NULL;
```

```
type pnod = ^nod;  
    nod = record  
        inf :integer;  
        urm :pnod;  
    end;
```

```
var Top : pnod;  
    Top := nil;
```

**Se refac operatiile de adaugare si stergere de la  
liste simplu inlantuite, respectand restrictiile!**

## Stiva in alocare dinamica

### *Exemplificare operatiilor C++*

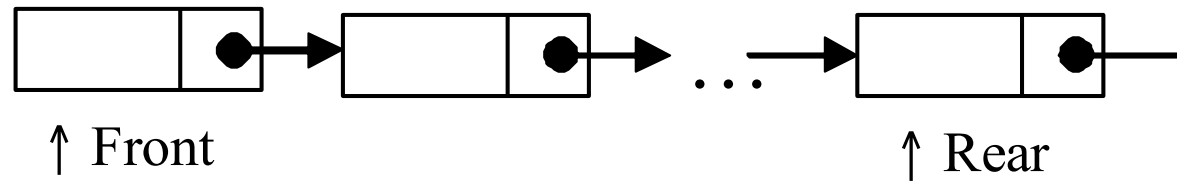
#### **void push(nod\*& Top, int val)**

```
{  
    nod* aux = new nod;  
    aux->info = val;  
    aux->urm = NULL;  
    if (Top == NULL)  
        Top = aux;  
    else  
    {  
        aux->urm = Top;  
        Top = aux;  
    }  
}
```

#### **void pop(nod\*& Top)**

```
{  
    if(Top!=NULL)  
    {  
        cout<<Top->info;  
        nod* aux = Top;  
        Top = Top ->urm;  
        delete aux;  
    }  
    else cout<<"Stiva vida\n";  
}
```

## Coadă în alocare dinamică



Inserari – **Rear**

Stergeri - **Front**

Coadă vidă: **Front = Rear = NULL**.

Coadă cu un singur element: **Rear = Front != NULL**.

**Se refac operațiile de adăugare și ștergere de la  
liste simplu înlantuite, respectând restricțiile!**

## Coadă în alocare dinamică

### Exemplificare operațiilor C++

```
void push(nod*& Front, nod*& Rear, int val)
```

```
{  
    nod* aux = new nod;  
    aux->info = val;  
    aux->urm = NULL;  
    if (Front == NULL)  
    {  
        Front = aux;  
        Rear = Front;  
    }  
    else  
    {  
        Rear->urm = aux;  
        Rear = aux;  
    }  
}
```

```
void pop(nod*& Front)
```

```
{  
    if (Front != NULL) {  
        nod * temp = Front;  
        if (Front == Rear)  
            Front=Rear=NULL;  
        else  
            Front=Front->next;  
        delete(temp); }  
}
```

## Aplicatii

### Stive si cozi

#### 1. Exemplificare mecanisme

Se dau structurile: o stiva  $S$  si doua cozi  $C1$  si  $C2$ , ce contin caractere. Cele trei structuri se considera de capacitate infinita, si initial vide. Se considera urmatoarele operatii:

‘ $x$ ’ : se introduce caracterul  $x$  in  $S$ ;

1 : daca  $S$  e nevida, se extrage un element si se introduce in  $C1$ , altfel nu se face nimic;

2 : daca  $C1$  e nevida, se extrage un element si se introduce in  $C2$ , altfel nu se face nimic;

3 : daca  $C2$  e nevida, se extrage un element si se introduce in  $S$ , altfel nu se face nimic.

(a) Sa se scrie continutul stivei  $S$  si al cozilor  $C1$  si  $C2$ , dupa executarea urmatoarelor secvente de operatii: **R 1 C 1 H 1 2 2 S E A R T 1 1 E E 2 2 2 1 1 2 2 3 3 3**

(b) Sa se scrie o secventa de operatii in urma careia stiva  $S$  sa contina cuvantul **BUBBLE**, coada  $C1$  sa fie vida, iar coada  $C2$  sa contina cuvantul **SORT**.

## Aplicatii

### Stive si cozi

#### 2. Parantezarea corecta

Dat un sir  $s = s_1 s_2 \dots s_n$  de caractere '(' si ')' sa se verifice daca acest sir este quasi - corect parantezat (i.e., pentru orice subsir  $s_1 s_2 \dots s_i$  avem ca numarul de caractere '(' este mai mare sau egal cu numarul de caractere ')').

In caz ca  $s$  nu este este quasi - corect parantezat, se va indica pozitia primei paranteze ')' care nu are corespondent.

**Expl:**  $()(())$  – corect

$()(())(())(())$  – corect

$()(())(())(())$  – incorect (prima paranteza ') care nu are corespondent este pe pozitia 7

## Aplicatii

### Stive si cozi

#### 2. Parantezarea corecta

#### Pascal

#### C / C++

```
bool ok=true;
for(int i=0; i<strlen(s); i++)
    {if(empty(Stack)) // Stiva e vida
        { if(s[i]=='')
            { ok=false; break;}
          push(s[i],Stack);}
      else
          if(s[i] == peek(Stack))
              push(s[i],Stack);
          else
              pop(Stack);
    }
if(ok) cout<<"Corect";
else cout<<"Incorect";
```

```
var ok:boolean; ok:=true;
for i:=0 to length(s) do
    begin
        if(empty(Stack)=true) then
            // Stiva vida
            begin
                if (s[i] = '' ) then
                    begin
                        ok=false; break;
                    end;
                push(s[i],Stack);
            end
        else
            if(s[i] = peek(Stack)) then
                push(s[i],Stack);
            else
                pop(Stack);
        end;
    end;
```

## Aplicatii

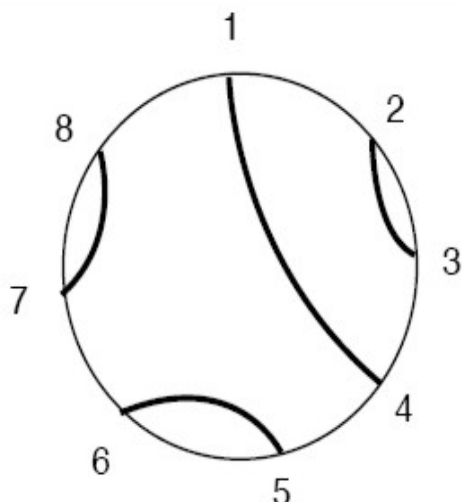
### Stive si cozi

### 3. Conectarea pinilor

Se da o suprafata circulara cu un numar  $n$  de pini pe margini (numerotati de la 1 la  $n$ ), impreuna cu o lista de perechi de pini ce trebuie conectati cu fire metalice.

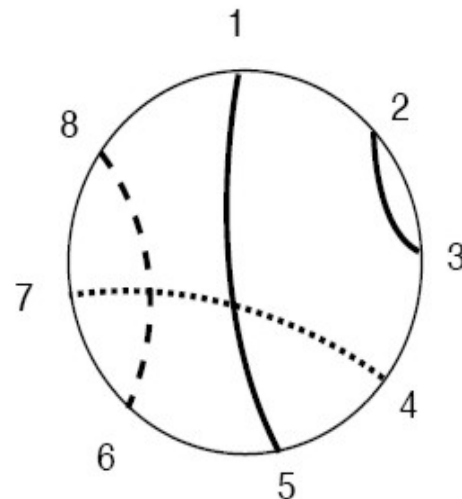
Problema cere sa determinati in timp  $O(n)$  daca pentru o configuratie ca mai sus, pinii pereche pot fi conectati, fara ca acestea sa se intersecteze.

*Pereche = {1,2,2,1,5,5,7,7}*



**Configuratie valida**

*Pereche = {1,2,2,4,1,6,4,6}*



**Configuratie invalida**



## Aplicatii

### Stive si cozi

### 3. Conectarea pinilor

#### C / C++

```
// citire vector pereche

for(int i=0; i<n; i++)
{ if(empty(Stack)) // Stiva e vida
    push(pereche[i],Stack);
  else
    if(pereche[i] == peek(Stack))
      pop(Stack);
    else
      push(pereche[i],Stack);
}

if (empty(Stack))
  cout<<"Configuratie valida";
else cout<<"Configuratie invalida";
```

#### Pascal

```
{ citire vector pereche}

for i:=1 to n do
  begin
    if(empty(Stack)=true) then
      // Stiva vida
      push(pereche[i],Stack);
    else
      if(pereche[i] = peek(Stack)) then
        pop(S);
      else
        push(pereche[i],Stack);
  end;

if (empty(Stack) = true)
  write('Configuratie valida')
else
  write('Configuratie invalida');
```

## Aplicatii

### Stive si cozi

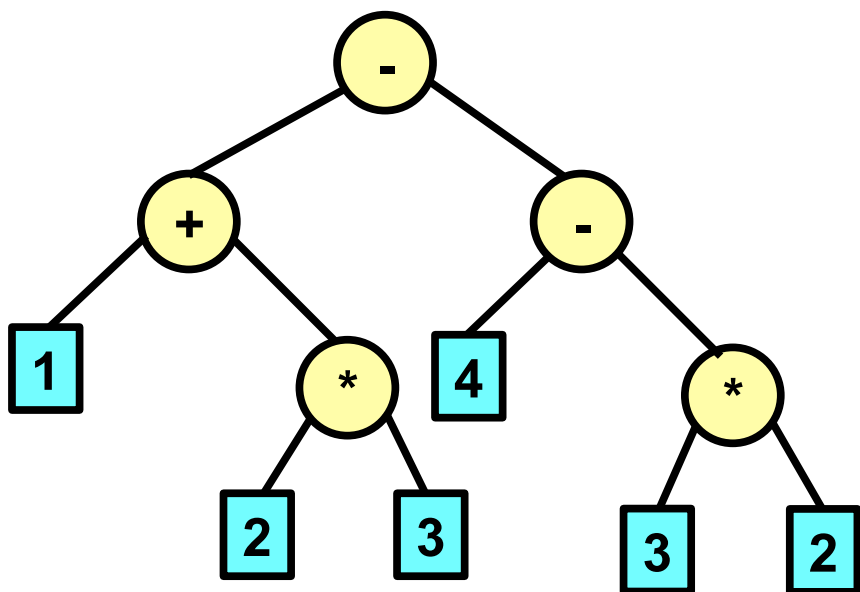
#### 4. Evaluarea unei expresii în notatie postfixata

Exemplu

$$(1 + 2 * 3) - (4 - 3 * 2)$$

in notatie postfixata: **1 2 3 \* + 4 3 2 \* - -**

#### Arborele binar asociat expresiei



Parcurgerea in preordine:

- + 1 \* 2 3 - 4 \* 3 2

Parcurgerea in inordine:

1 + 2 \* 3 - 4 - 3 \* 2

Parcurgerea in postordine:

1 2 3 \* + 4 3 2 \* - -

## Aplicatii

### Stive si cozi

#### 4. Evaluarea unei expresii în notatie postfixata

##### Algoritm

**Pas 1.** – se citeste un sir de caractere, reprezentand expresia in **postfix**; **se considera diferentierea între operanzi (/ operator) spatiul**; Stiva initial vida;

**Pas 2.** - se considera, pe rand, fiecare caracter.

Daca este “spatiu”, se trece la urmatorul;

Daca este operand → **Pas 3**;

Altfel → **Pas 4**;

**Pas 3.** - daca este operand, atunci:

- se extrag din stiva ultimele valori inserate, se aplica operandul si noua valoare se reintroduce in stiva

**Pas 4.** – se transforma caracterul in cifra si se adauga la numarul care va fi introdus in stiva

//  $\text{numar} = \text{numar} * 10 + (\text{caracter} - '0')$  \*

//  $\text{cifra} = \text{cod ASCII caracter} - 48$  (codul caracterului '0')

Se repeta **Pas 2** pana la terminarea sirului de caractere introdus.

**Pas ultim.** Rezultatul este singura valoare aflata in stiva.

## Aplicatii

### Stive si cozi

#### 4. Evaluarea unei expresii în notatie postfixata

Exemplu

$$(1 + 2 * 3) - (4 - 3 * 2)$$

in notatie postfixata:

1 2 3 \* + 4 3 2 \* - -

**Stiva, dupa fiecare pas**

		3	3						3	3	2	6	6			
	2	2	2	6	6		4	4	4	4	4	4	4	-2	-2	
1	1	1	1	1	1	7	7	7	7	7	7	7	7	7	7	9

## Aplicatii

### Stive si cozi

#### 4. Evaluarea unei expresii în notatie postfixata

##### *Detalii de implementare (C++)*

**Pas 2.** – parcurgerea sirului caracter cu caracter si verificarea acestuia (spatiu/ operator/ operand)

```
int evaluare(char *postfix) // postfix - sirul de caractere introdus
{
    int rez;

    for(int k = 0; k<strlen(postfix); k++)
    {
        if (postfix[k]==' ') continue;
        if (eOperator(postfix[k])==1) // daca acesta este operator
        {
            else
            {
                afis_fara_stergere();
            }
        }
        pop(rez);
        return rez;
    }
}
```

## Aplicatii

### Stive si cozi

#### 4. Evaluarea unei expresii în notatie postfixata

##### *Detalii de implementare (C++)*

**Pas 3.** – caracterul este operator (+,-,\*,/,%)

```
int eOperator(char x)
{
    if (x=='+' || x=='-' || x=='*' || x=='/' || x=='%') return 1;
    return 0;
}
```

```
if (eOperator(postfix[k])==1) // daca acesta este operator
{
    int op1, op2;
    pop(op1); // se scot ultimele 2 valori din stiva si se aplica operandul
    pop(op2);

    if(postfix[k]=='+') rez = op2+op1;
    else if (postfix[k]=='-') rez = op2-op1;
    else if (postfix[k]=='/') rez = op2/op1;
    else if (postfix[k]=='*') rez = op2*op1;
    else if (postfix[k]=='%') rez = op2%op1;

    push(rez); // se reintroduce in stiva rezultatul operatiei
}
```

## Aplicatii

### Stive si cozi

#### 4. Evaluarea unei expresii în notatie postfixata

##### *Detalii de implementare (C++)*

**Pas 4.** – caracterul este cifra

```
int eCifra(char x)
{
    if (x>='0' && x<='9') return 1;
    return 0;
}
```

```
else
{
    int numar=0; // daca acesta este cifra, se updateaza numarul care se introduce in stiva
    while(k<strlen(postfix) && eCifra(postfix[k])==1)
    {
        numar = numar * 10 + (postfix[k]-'0');
        k++;
    }
    push(numar); // numar intreg = cod ASCII caracter - 48 (codul caracterului '0')
}
```

## Aplicatii

### Stive si cozi

#### 5. Parcurgerea unui arbore pe nivele (BF)

##### C / C++

```
Front = 1; Rear = 1; // Q[ ] - coada
// a – matricea de adiacenta
cin >> nod; // de inceput
Q[Front] = nod;
viz[nod] = 1;
while (Front <= Rear)
{
    cout << Q[Front];
    for (i = 1; i <= n; i++)
        if (a[Q[Front]][i] == 1 && viz[i] != 1)
        {
            Rear++;
            Q[Rear] = i;
            viz[i] = 1;
        }
    Front++;
}
```

##### Pascal

```
Front := 1; Rear := 1;
read(nod); // de inceput
Q[Front] := nod;
viz[nod] := 1;
while (Front <= Rear) do
begin
    write(Q[Front], ' ');
    for i := 1 to n do
        if (a[Q[Front]][i] = 1) and (viz[i] != 1) then
            begin
                Rear := Rear + 1;
                Q[Rear] := i;
                viz[i] := 1;
            end;
    Front := Front + 1;
end;
```



## Aplicatii

### Stive si cozi

### 6. Sirul lui Hamming

Șirul lui Hamming se definește ca fiind mulțimea de numere  $H = \{2^i * 3^j * 5^k \mid i, j, k \text{ sunt numere naturale}\}$ . Deci primii 10 termeni ai acestui șir sunt 1, 2, 3, 4, 5, 6, 8, 9, 10, 12.

Se cere un algoritm care generează (eventual in ordine) termenii mai mici sau egali cu un  $M$  ai acestui șir.

Generarea termenilor șirului Hamming se bazează pe următoarea definiție a șirului:

1. 1 este termen al șirului (deoarece  $1 = 2^0 * 3^0 * 5^0$ )
2. dacă  $x$  este un termen al șirului, atunci  $2 * x$ ,  $3 * x$  și  $5 * x$  sunt termeni ai șirului
3. șirul conține numai numere care îndeplinesc punctele 1 și 2.

## Aplicatii

### Stive si cozi

#### Implementare

### 6. Sirul lui Hamming

#### Semnificatia variabilelor utilizate

h - vectorul care stocheaza sirul lui Hamming;

p - indexul asociat acestui vector;

c2 - coada ce contine elementul  $2 \cdot x$ , unde x este membru al sirului lui Hamming;

f2 si r2 - indecsii primului, respectiv ultimului element din c2;

m2 - valoarea primului element din coada c2;

c3 - coada ce contine elementul  $3 \cdot x$ ;

f3 si r3 - indecsii primului, respectiv ultimului element din c3;

m3 - valoarea primului element din coada c3;

c5 - coada ce contine elementul  $5 \cdot x$ ;

f5 si r5 - indecsii primului, respectiv ultimului element din c5;

m5 - valoarea primului element din coada c5;

$m = \min(m2, m3, m5)$ .

## Aplicatii

### Stive si cozi

#### 6. Sirul lui Hamming

##### Algoritm

Pas Initial:

- primul element al sirului h este 1;
- se initializeaza cele 3 cozi astfel: in c2 se insereaza valoarea 2, in c3 se insereaza valoarea 3 si in c5, valoarea 5.

Cat timp nu s-a ajuns la valoarea maxima M:

Pas repetitiv:

- se alege minimul  $m$  dintre capetele celor 3 cozi;
- se pune acest minim in vectorul care retine stocheaza sirul lui Hamming;
- se vor insera în cele 3 cozi numerele  $m \times 2$ ,  $m \times 3$ ,  $m \times 5$ ;
- se avanseaza in coada (cozile) din care a provenit minimul.

## Aplicatii

### Stive si cozi

### 6. Sirul lui Hamming

```
cin>>M;  
  m = 1 ;p=1; h[p]=m ;  
  
    r2=r3=r5=0;  
    c2[++r2]=m*2;  
    c3[++r3]=m*3;  
    c5[++r5]=m*5;  
    f2=f3=f5=1;  
  
    m2=c2[f2++];  
    m3=c3[f3++];  
    m5=c3[f5++];
```

## Aplicatii

### Stive si cozi

#### 6. Sirul lui Hamming

```
while (m<M)  {  
    m=m2;  
    if(m>m3) m=m3;  
    if(m>m5) m=m5;  
  
    if (m <= M) h[++p] = m ;  
    c2[++r2]=m*2;    c3[++r3]=m*3;    c5[++r5]=m*5;  
  
    if (m == m2) m2 = c2[f2++];  
    if (m == m3) m3 = c3[f3++];  
    if (m == m5) m5 = c5[f5++];  
}
```

## Aplicatii

### Cozi circulare

### 7. Josephus

- $n$  copii asezati in cerc sunt numarati din  $m$  in  $m$  plecand de la copilul  $k$ .
- fiecare al  $m$  – lea copil numarat iese din cerc.
- afisare ordine iesire copii din cerc

$n = 12$   
 $m = 3$   
 $k = 2;$

1	2	3	4
12			5
11			6
10	9	8	7

1	2	3	4
12			5
11			6
10	9	8	7

Afis: 2, 5, 8, 11

1		3	4
12			
			6
10	9		7

Afis: 3, 7, 12

1			4
			6
10	9		

Afis: 6, 1

Ordine: 2,5,8,11,3,7,12,6,1,10,4,9

1			4
			6
10	9		

Afis: 10, 4

1			4
			6
10	9		

Afis: 9

## Aplicatii

### Cozi circulare

### 7. Josephus

```
int n,i,k,m,x,p[100]; //poz - vectorul de pozitii ale copiilor

cout<<"nr copii = ";   cin>>n;
cout<<"initial = ";   cin>>k;
cout<<"salt = ";     cin>>m;

for (i = 1; i<=n; i++) poz[i] = i;

i = k;
cout<<poz[i]<<" ";
for (int j = i; j<n; j++) poz[j] = poz[j+1]; // eliminarea de pe pozitia k
    n--;
```

## Aplicatii

### Cozi circulare

### 7. Josephus

```
while (n>0)
{
    i = i+ m-1; //salt peste m pozitii
    if (i%n==0) i = n; // situatie speciala in cazul numerotarii 1..n
    else if (i > n) i = i % n; //simulare coada circulara
                                //prin pastrarea indicelui in intervalul [0,n-1];
    cout<<poz[i]<<" ";
    for (int j = i; j<n; j++) poz[j] = poz[j+1]; // eliminarea de pe pozitia i + m
    n--;
}
```



## Aplicatii

### Liste simplu inlantuite

### 8. Reprezentarea vectorilor rari

#### Un vector rar:

- are cel putin 80% dintre elemente = 0.
- reprezentare eficienta  $\rightarrow$  liste simplu inlantuite alocate dinamic
- fiecare nod din lista retine:
  - **valoarea**
  - **indicele din vector**

**Cerinte:** adunarea, respectiv, produsul scalar a doi vectori rari.

## Aplicatii

### Liste simplu inlantuite

### 8. Reprezentarea vectorilor rari

V1 si V2 – vectori rari

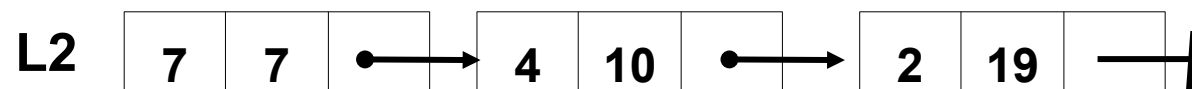
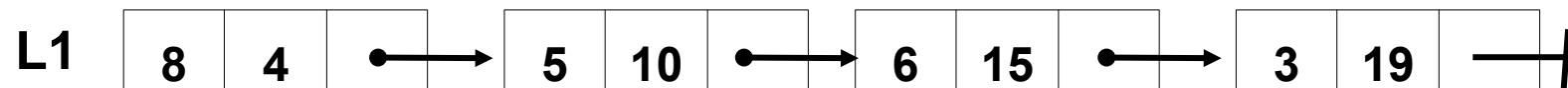
V1

0	0	0	8	0	0	0	0	0	5	0	0	0	0	6	0	0	0	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

V2

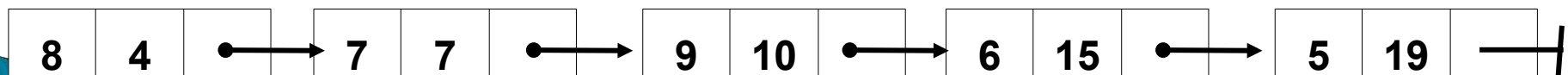
0	0	0	0	0	0	7	0	0	4	0	0	0	0	0	0	0	0	2	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

*Transformarea in liste simplu inlantuite*



*Produsul scalar* =  $5 \times 4 + 3 \times 2 = 26$

L1+L2



## Aplicatii

### Liste simplu inlantuite

### 8. Reprezentarea vectorilor rari

#### Crearea unui vector rar

```
void inserare(nod *&prim, nod *&ultim, int a, int b)
{  nod *q = new nod;
   q->val=a;   q->poz=b; q->urm=NULL;

   if(prim==NULL)
   {   prim = q;
       ultim = prim;}
   else
   {   ultim -> urm = q;
       ultim = q; }
}
```

```
struct nod
{
    int poz, val;
    nod*urm;
};
```

```
void creare_vector(int &n, nod *&p, nod *&u)
{ int i,a,b;
  cin>>n;
  for(i=1;i<=n;i++)
  {cin>>a>>b;
   inserare(p, u, a, b);
  }
}
```

## Aplicatii

### *Suma a doi vectori rari*

```
void suma (nod *prim1, nod *prim2, nod *&prim3, nod *&ultim3)  
{ nod *p1=prim1, *p2=prim2;  
  while ((p1!=NULL) || (p2!=NULL))  
  { if ((p2==NULL) || (p1 -> poz < p2 -> poz))  
    {   inserare(prim3, ultim3, p1 -> val, p1 -> poz);  
        p1=p1->urm; }  
    else  
      if ((p1==NULL) || (p2 -> poz < p1 -> poz))  
      {   inserare(prim3, ultim3, p2 -> val, p2 -> poz);  
          p2=p2->urm; }  
    else  
      {   inserare(prim3, ultim3, p1 -> val + p2 -> val, p1 -> poz);  
          p1=p1->urm;  
          p2=p2->urm;  
      }  
  }  
}
```

## Aplicatii

### Liste simplu inlantuite

### 8. Reprezentarea vectorilor rari

#### *Produsul scalar a 2 vectori rari*

```
int prod_scalar(nod *prim1, nod *prim2)
{
    int prod = 0;
    nod *p1, *p2;

    for (p2 = prim2; p2 != NULL; p2 = p2 -> urm)
        for (p1 = prim1; p1 != NULL; p1 = p1 -> urm)
            if (p2 -> poz == p1 -> poz) prod += p1 -> val * p2 -> val;
    return prod;
}
```

## Structuri liniare (Liste. Stive. Cozi)

### Concluzii

S-au recapitulat notiunile urmatoare:

- Structuri liniare în alocare statica si dinamica
- Structuri liniare fara restrictii de intrare/iesire
- Structuri liniare cu restrictii de intrare/iesire (stive si cozi)

# Succes!