



Structuri de Date si Algoritmi

- suport de curs -

Dobrovat Anca - Madalina

An universitar 2019 – 2020

Semestrul I

Seriile 21 + 25

Curs 2 & 3

08 - 15/10/2019



Curs 2 & 3 - Cuprins

1. Algoritmi (rest Curs 1)

Analiza performantei algoritmilor.

Cateva clase de complexitate pentru comportarea asimptotica a algoritmilor.

2. Structuri de date elementare

Clasificare. Operatii elementare.



Complexitatea algoritmilor

Analiza complexității unui algoritm => determinarea resurselor de care acesta are nevoie pentru a produce datele de ieșire.

Resurse - timpul de executare
- spatiu de memorie etc.

Obs: Modelul masinii pe care va fi executat algoritmul nu presupune existenta operatiilor paralele (operatiile se executa secvential).

Notatie: $T(n)$ – timp de rulare al unui algoritm (in general masurat in nr. de comparatii sau de mutari)

Cazuri:

- cel mai favorabil
- cel mai nefavorabil
- mediu



Complexitatea algoritmilor

De ce se alege, in general, cazul cel mai defavorabil?

- este cel mai raspandit
- timpul mediu de executare este de multe ori apropiat de timpul de executare in cazul cel mai defavorabil
- ofera o limita superioara a timpului de executare (avem certitudinea ca executarea algoritmului nu va dura mai mult)



Complexitatea algoritmilor

Numararea operatiilor elementare realizate de un
algoritm in cazul cel mai defavorabil

Aflarea maximului unui
vector de dimensiune n

operații elementare realizate de
algoritm în cazul cel mai defavorabil

<code>maxim = v[0];</code>	→	2	(o indexare + o atribuire)
<code>for (i=1; i<n;i++)</code>	→	$2n$	(o atribuire + n comparații + $(n-1)$ incrementări)
<code>if (maxim < v[i])</code>	→	$2(n-1)$	$(n-1)$ indexări + $n-1$ comparații)
<code>maxim = v[i];</code>	→	$2(n-1)$	$(n-1)$ indexări + $n-1$ atribuiri)
<code>return maxim</code>	→	1	(o întoarcere a rezultatului)

$T(n) = 6n-1$ operații elementare



Complexitatea algoritmilor

Exemple

1. Cautarea unei valori intr-un sir **ordonat** (Cautarea binara)

```
int main()
{
  int left = 0, right = n - 1;
  int mid = (left + right) / 2;
  while (left <= right && val != v[mid])
  {
    if (val < v[mid]) right = mid - 1;
    else left = mid + 1;
    mid = (left + right) / 2;
  }
  if (v[mid] == val) loc = mid;
  else loc = UNDEFINED;
}
```

- **$O(\log_2 n)$**

Exemplu: caut (fara succes) elementul 10 in
sirul $v = (20, 30, 40, 50, 60, 70, 80, 90, 100)$

- compar 10 cu 60 (elem din mijloc); $10 \neq 60$
- $10 < 60 \Rightarrow$ caut in $v = (20, 30, 40, 50)$

- compar 10 cu 30 (noul elem din mijloc)
- $10 < 30 \Rightarrow$ caut in $v = (20)$

- compar 10 cu 20 (unicul elem)
- cautare fara succes

$n = 9, \lceil \log_2 9 \rceil = 3$



Complexitatea algoritmilor

Exemple

2. Ordonarea unui sir folosind Interschimbarea directa

```
int main()
{
    int v[100], n, i, j, aux;
    // citire vector
    for (i = 1; i < n; i++)
        for (j = i+1; j <= n; j++)
            if (v[i] > v[j])
                { aux = v[i];
                  v[i] = v[j];
                  v[j] = aux;
                }
    // Afisare vector ordonat
}
```

Caz	Comparatii	Mutari
Cel mai favorabil	$n(n - 1) / 2$	0
Cel mai defavorabil	$n(n - 1) / 2$	$3n(n - 1) / 2$
mediu	$n(n - 1) / 2$	$3n(n - 1) / 4$



Complexitatea algoritmilor

Exemple

3. Ordonarea unui sir folosind **Insertia directa**

```
int main()
{
    int v[100], n, i, j, aux;
    // citire vector
    for (i = 2; i<=n; i++)
    {
        x = v[i];
        j = i - 1;
        while (j>0 && x < v[j])
        {
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = x;
    }
    // Afisare vector ordonat
}
```

Caz	Comparatii	Mutari
Cel mai favorabil	$n - 1$	$2(n - 1)$
Cel mai defavorabil	$\frac{1}{2} n^2 + \frac{1}{2} n - 1$	$\frac{1}{2} n^2 + \frac{3}{2} n - 2$
mediu	$(\frac{1}{2} n^2 + \frac{1}{2} n - 1)/2$	$C_{\text{mediu}} + 2(n-1)$

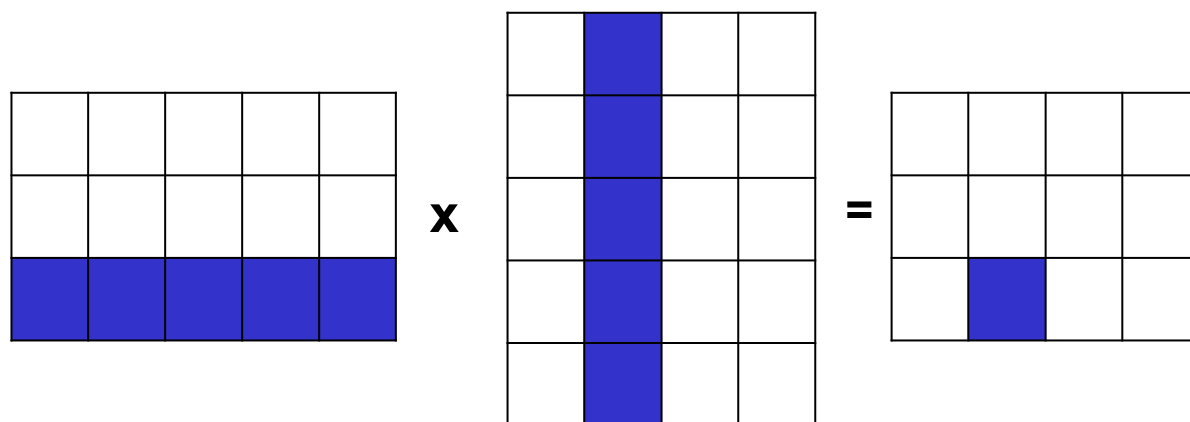


Complexitatea algoritmilor

Exemple

4. Inmultirea a doua matrice

```
int main()
{
  int a[10][20], b[20][30], c[10][30];
  int n, m, p, i, j, k;
  // citire matrice a si b
  for(i=1; i<=n; i++)
    for(k=1; k<=p; k++)
      { c[i][k] = 0;
        for(j=1; j<=m; j++)
          c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
  // Afisare matrice produs
}
```



$$A_{n,m} \times B_{m,p} = C_{n,p}$$

$$O(m \cdot n \cdot p)$$



Complexitatea algoritmilor

Notatii:

$T(n)$ – timp de rulare al unui algoritm (in general masurat in nr. de comparatii sau de mutari)

C_{\max} – numarul maxim de comparatii (obtinut in cazul cel mai defavorabil)

C_{\min} – numarul minim de comparatii (cazul cel mai favorabil)

M_{\max} – numarul maxim de mutari (operatii elementare) – caz defavorabil

M_{\min} – numarul minim de mutari – caz favorabil



Complexitatea algoritmilor

Notatia asimptotica

$T(n)$ – timp de rulare al unui algoritm (comparatii / mutari)

Obs: Exista un timp minim si un timp maxim de rulare

$$C_{\min} \leq T(n) \leq C_{\max}$$



Complexitatea algoritmilor

Notatia asimptotica

$T(n)$ – timp de rulare al unui algoritm (comparatii / mutari)

Obs: Exista un timp minim si un timp maxim de rulare

$$C_{\min} \leq T(n) \leq C_{\max}$$

Margini superioare (si inferioare) ([4])

Timp de rulare $T(n)$ - margine superioara

$$T(n) \leq \frac{1}{2}n^2 + (3/2)n - 2 \text{ (expl.)} \Rightarrow T(n) = \mathbf{O(n^2)}$$

Timp de rulare $T(n)$ - margine inferioara

$$3(n - 1) \leq T(n) \text{ (expl.)} \Rightarrow T(n) = \mathbf{\Omega(n)}$$

Timp de rulare $T(n)$ in cazul cel mai nefavorabil

$$T(n) = a \cdot C_{\max} + b \text{ } (\exists \text{ const } a, b > 0) \Rightarrow T(n) = \mathbf{\Theta(n^2)}$$

[4] Ceterchi R. – Algoritmi si Structuri de Date (note de curs 2013)



Complexitatea algoritmilor

Ordinul de crestere a functiilor

Notatia asimptotica -

comportarea lui $T(n)$ cind $n \rightarrow \infty$ ([4])

Formal

$f : \mathbb{N} \rightarrow \mathbb{R}_+$ (f asimptotic pozitiva)

$O(g) := \{f \mid \exists c > 0, \exists n_0 \text{ a.i. } 0 \leq f(n) \leq cg(n) \text{ oricare } n \geq n_0\}$

$\Omega(g) := \{f \mid \exists c > 0, \exists n_0 \text{ a.i. } 0 \leq cg(n) \leq f(n) \text{ oricare } n \geq n_0\}$

$\Theta(g) := \{f \mid \exists c_1, c_2 > 0, \exists n_0 \text{ a.i. } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ oricare } n \geq n_0\}$

[4] Ceterchi R. – Algoritmi si Structuri de Date (note de curs 2013)



Complexitatea algoritmilor

Notatia asimptotica – Proprietati [4]

$$\Theta(g) = O(g) \cap \Omega(g)$$

Tranzitivitate

$$f = \Theta(g), g = \Theta(h) \implies f = \Theta(h)$$

$$f = O(g), g = O(h) \implies f = O(h)$$

$$f = \Omega(g), g = \Omega(h) \implies f = \Omega(h)$$

Reflexivitate

$$f = \Theta(f)$$

$$f = O(f)$$

$$f = \Omega(f)$$

Simetrie

$$f = \Theta(g) \text{ daca si numai daca } g = \Theta(f)$$

[4] Ceterchi R. – Algoritmi si Structuri de Date (note de curs 2013)



Complexitatea algoritmilor

Ordonarea dupa ratele de crestere asimptotica

Ordonati funcțiile pe baza creșterii lor asimptotice

$$f_1(n) = n + \sin n$$

$$f_2(n) = \ln n$$

$$f_3(n) = n + \sqrt{n}$$

$$f_4(n) = \frac{1}{n}$$

$$f_5(n) = 13 + \frac{1}{n}$$

$$f_6(n) = 13 + n$$

$$f_7(n) = (n + \sin n) \cdot (n^{20} - 5)$$

$$f_8(n) = n^e$$

$$f_9(n) = n^n$$

$$f_{10}(n) = n \cdot \ln n$$

$$f_{11}(n) = n \cdot (\ln n)^2$$

$$f_{12}(n) = \log_2 n$$

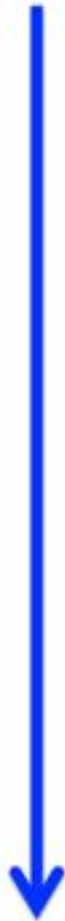
$$f_{13}(n) = e^n$$



Complexitatea algoritmilor

Ordonarea dupa ratele de crestere asimptotica

Ordonarea funcțiilor pe baza creșterii lor asimptotice


$$f_4(n) = \frac{1}{n}$$

$$f_5(n) = 13 + \frac{1}{n}$$

$$f_2(n) = \ln n \quad f_{12}(n) = \log_2 n$$

$$f_1(n) = n + \sin n \quad f_3(n) = n + \sqrt{n} \quad f_6(n) = 13 + n$$

$$f_{10}(n) = n \cdot \ln n$$

$$f_{11}(n) = n \cdot (\ln n)^2$$

$$f_8(n) = n^e$$

$$f_7(n) = (n + \sin n) \cdot (n^{20} - 5)$$

$$f_{13}(n) = e^n$$

$$f_9(n) = n^n$$



Structuri liniare (Liste. Stive. Cozi)

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 (fiecare element are un singur element **precedent** si un singur element **succesor**).

Exemple de structuri **liniare** – liste, stive, cozi

Exemple de structuri **neliniare**

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



Structuri liniare (Liste. Stive. Cozi)

Clasificare

Dupa succesiunea elementelor:

- succesorul unui element e in zona imediat alaturata (liste secventiale)
- orice element retine si adresa succesorului (liste înlantuite).

Dupa modul de efectuare al operatiilor de intrare (inserarile) si de iesire (stergerile):

- 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];}
```

Cautare (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  
    write('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 lineare cu restrictii la i/o: Stiva (LIFO)

- **LIFO (Last In First Out)**: ultimul introdus este primul extras
- locul unic pt. ins./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 lineare cu restrictii la i/o: **Stiva (LIFO)**

Exemplificarea mecanismului RECURSIVITATII și ordinea efectuării operațiilor

$n! =$ $\left[\begin{array}{l} 1, \text{ dacă } n=0 \\ n*(n-1)!, \text{ dacă } n \geq 1 \end{array} \right.$

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

$4! = 4*3! = 4 * 6 = 24$
 $3! = 3*2! = 3 * 2 = 6$
 $2! = 2*1! = 2 * 1 = 2$
 $1! = 1*0! = 1 * 1 = 1$
 $0! = 1$

**Adâncimea
recursivității**

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



Liste liniare alocate secvential

Structuri lineare 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

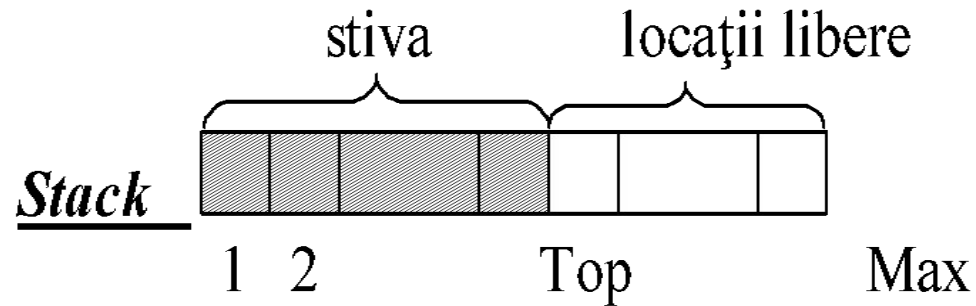
A_5	0	-	-	1
A_4	1	-	-	1
A_3	2	-	-	2
A_2	3	-	-	6
A_1	4	-	-	24

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 = 0;
```

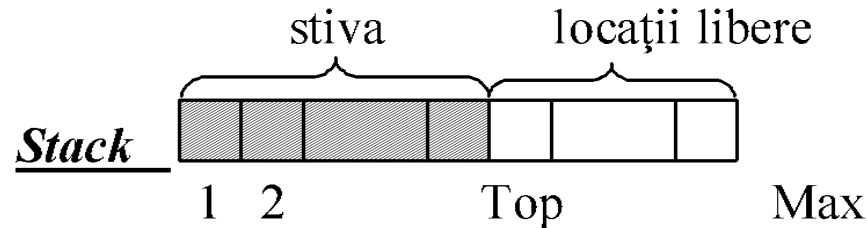
Pascal

```
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)
        // Overflow
    else
    {
        Top++;
        Stack[Top] = Val;
    }
}
```

Inserare

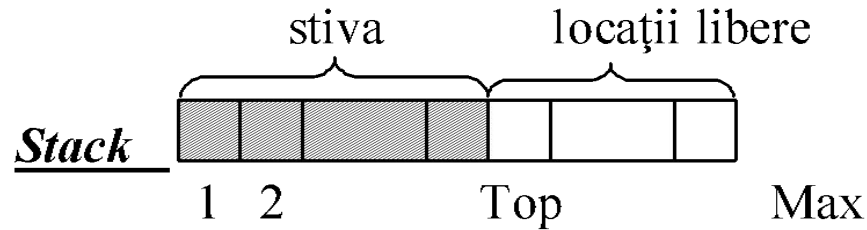
Pascal

```
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 == 0)
        // 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

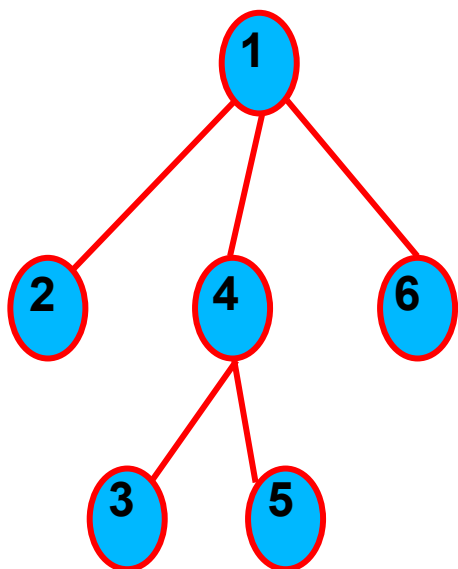
Structuri lineare cu restrictii la i/o: **Cooda** (FIFO)

- **FIFO (First In First Out)**: primul introdus este primul extras
- capat pt. Inserari: sfirsit (*Rear*)
- capat pt. stergeri: inceput (*Front*)
- ***Push (Val)*** - inserarea
 - **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 lineare 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.

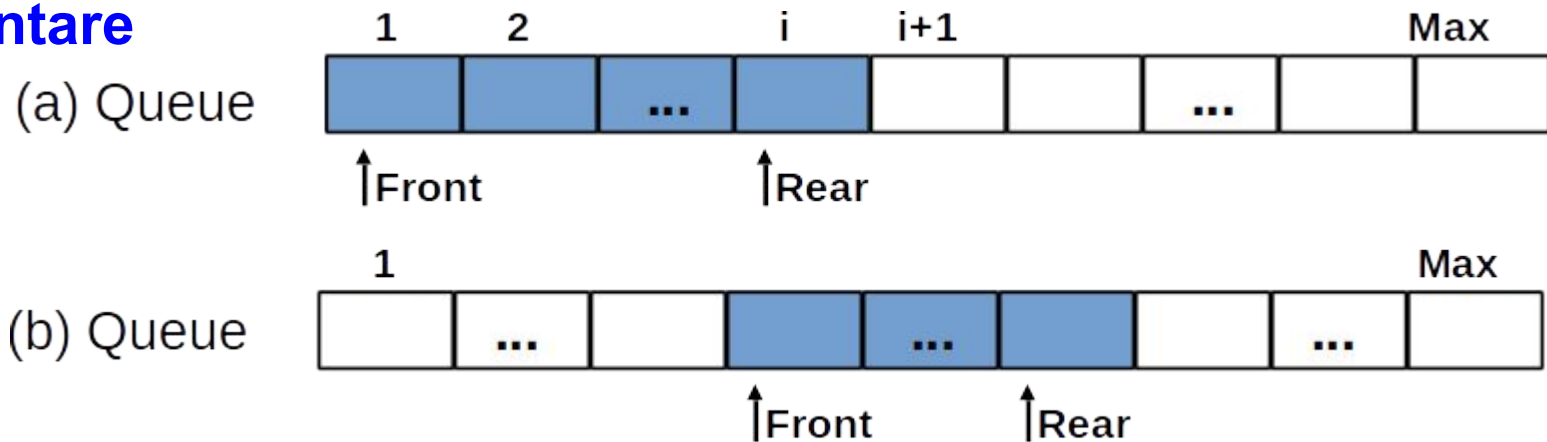
PUSH 1	<table><tr><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	1									
1											
POP	<table><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>										Afis: 1
PUSH 2	<table><tr><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>		2								
	2										
PUSH 4	<table><tr><td></td><td>2</td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>		2	4							
	2	4									
PUSH 6	<table><tr><td></td><td>2</td><td>4</td><td>6</td><td></td><td></td><td></td><td></td><td></td></tr></table>		2	4	6						
	2	4	6								
POP	<table><tr><td></td><td></td><td>4</td><td>6</td><td></td><td></td><td></td><td></td><td></td></tr></table>			4	6						Afis: 2
		4	6								
POP	<table><tr><td></td><td></td><td></td><td>6</td><td></td><td></td><td></td><td></td><td></td></tr></table>				6						Afis: 4
			6								
PUSH 3	<table><tr><td></td><td></td><td></td><td>6</td><td>3</td><td></td><td></td><td></td><td></td></tr></table>				6	3					
			6	3							
PUSH 5	<table><tr><td></td><td></td><td></td><td>6</td><td>3</td><td>5</td><td></td><td></td><td></td></tr></table>				6	3	5				
			6	3	5						
POP	<table><tr><td></td><td></td><td></td><td></td><td>3</td><td>5</td><td></td><td></td><td></td></tr></table>					3	5				Afis: 6
				3	5						
POP	<table><tr><td></td><td></td><td></td><td></td><td></td><td>5</td><td></td><td></td><td></td></tr></table>						5				Afis: 3
					5						
POP	<table><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>										Afis: 5

Coada vida



Coada in alocare statica

Implementare



C / C++

```
#define MAX 100  
  
int Queue[MAX];  
int Front, Rear;  
Front = Rear = 0;
```

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)
        // Overflow
    else
        {if (Rear == 0)
            //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 == 0 || Front >
Rear)
            // Coadă vida
        else
        {
            X = Queue[Front];
            Front++;
        }
    }
}
```

Stergere

Pascal

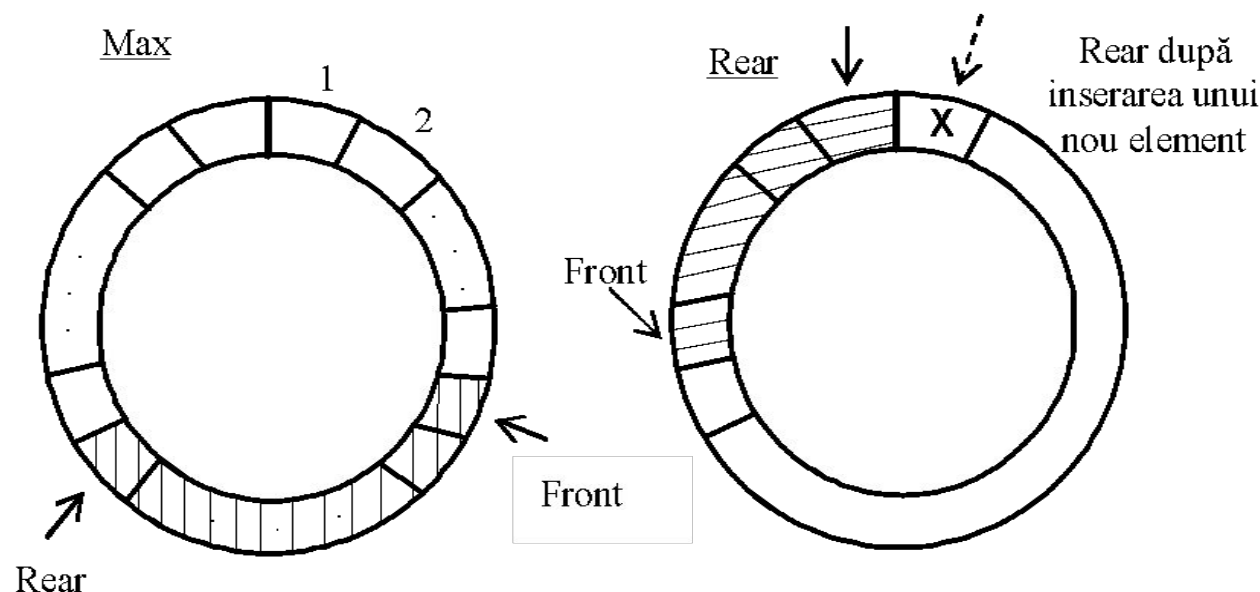
```
procedure Pop (var X:integer);
begin
    if (Front = MAX) then
        // Underflow
    else
        begin
            if (Front = 0 OR Front > Rear)
                // Coadă vida
            else begin
                X :=
Queue[Front];
                Front := Front + 1;
            end;
     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



Liste liniare alocate secvential

Structuri lineare cu restrictii la i/o: Alte tipuri de cozi

Coada circulara (in alocare statica)



Pe coada circulara: $\text{aritmetica (mod Max)}$ la incrementarea indicilor

Coada vidă: $\text{Front} = \text{Rear} = 0$.

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

Coada cu un singur element: $\text{Rear} = \text{Front} \neq 0$.



Liste liniare alocate secvential

Structuri lineare 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,

1		3	4
12			
			6
10	9		7

Afis: 3, 7, 12

1			4
			6
10	9		

Afis: 6, 1

Afis: 10, 4

1			4
			6
10	9		

1			4
			6
10	9		

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



Structuri lineare cu restrictii la i/o: Alte tipuri de cozi

Cozi circulare

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--;
```



Structuri lineare cu restrictii la i/o: Alte tipuri de cozi

Cozi circulare

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--;
}
```



Liste liniare alocate secvential

Structuri lineare cu restrictii la i/o: Alte tipuri de cozi

Coada 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 lineare 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 apare 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

Dezavantaj:

- 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 = 0;
```

Declaraire

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

Existenta spatiu de memorare

```
if (n<100)  
    //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)
{   alocare(prim);
    a[prim].inf = val;
    a[prim].urm = 0;
    ultim = prim; }
else if (n<100)
{   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');
```

[illegible]



Liste liniare inlantuite alocate static

C / C++

Inserare

Pascal

Inserarea valorii “val_ins” dupa valoarea “val”

```
int p, nou;
if (n<100)
{
    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”

```
int p, nou;
if (n<100)
    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++

Inserare

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++

Inserare

Pascal

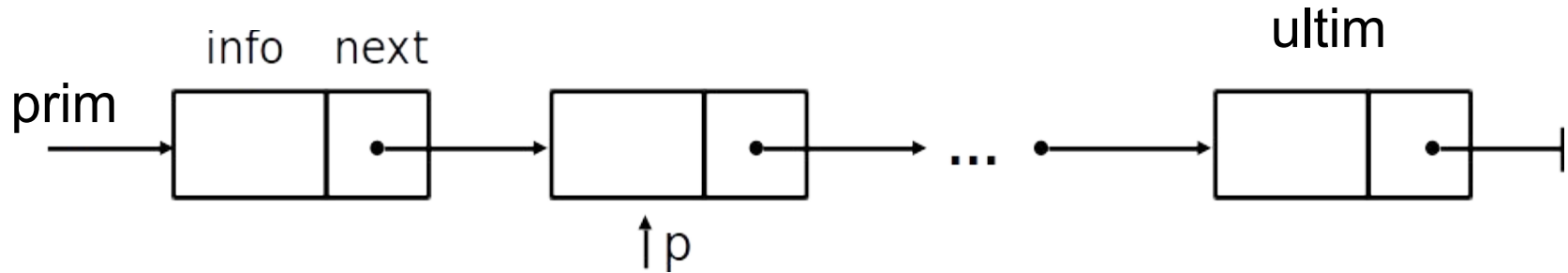
Stergerea valorii “val” din lista

```
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++

Pascal

Cautare

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

```
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}
```



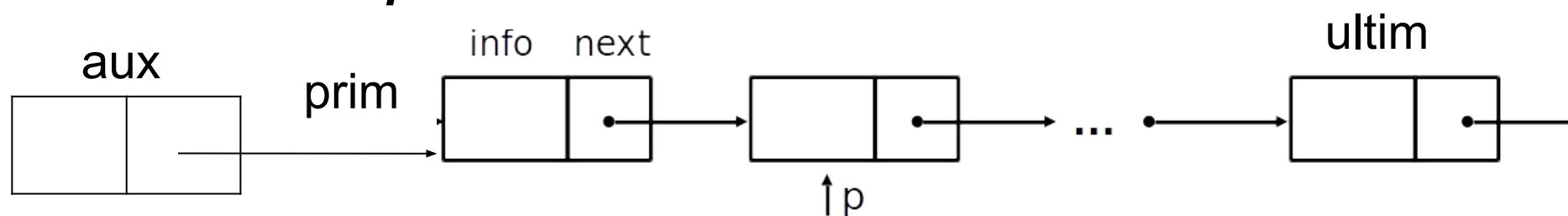
Liste simplu inlantuite

C / C++

Inserare

Pascal

Inserarea la inceputul listei



// aux = nodul de inserat

{aux = nodul de inserat}

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

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



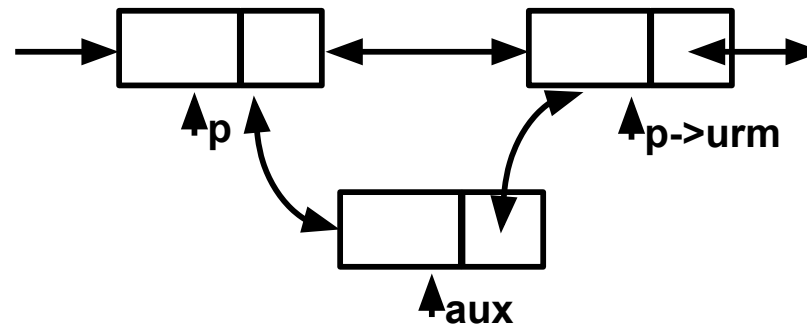
Liste simplu inlantuite

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;
```



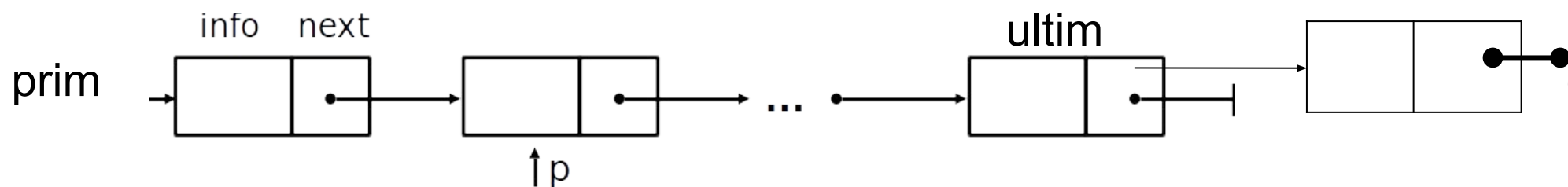
Liste simplu inlantuite

C / C++

Inserare

Pascal

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; }
```

```
{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;
```

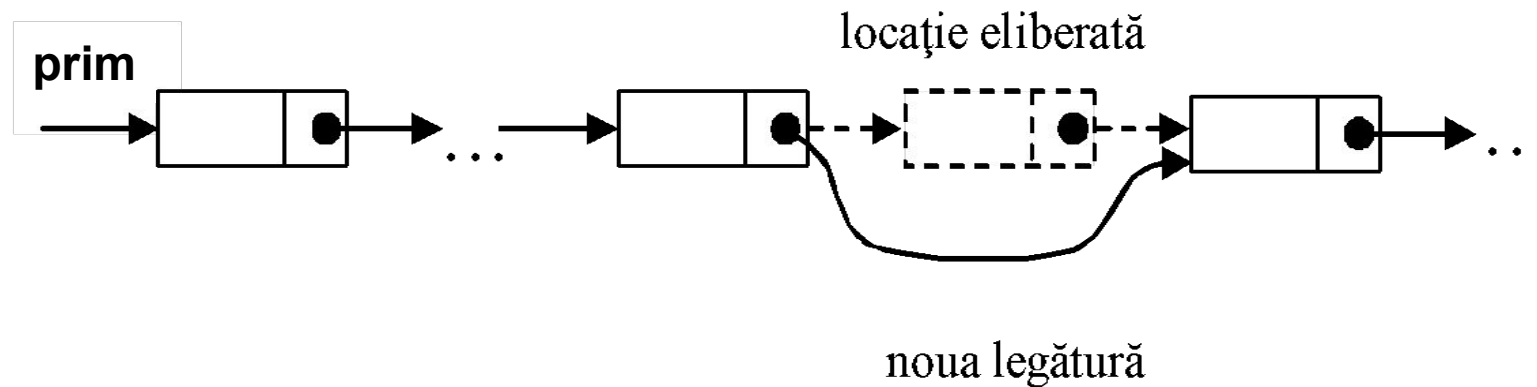


Liste simplu inlantuite

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)



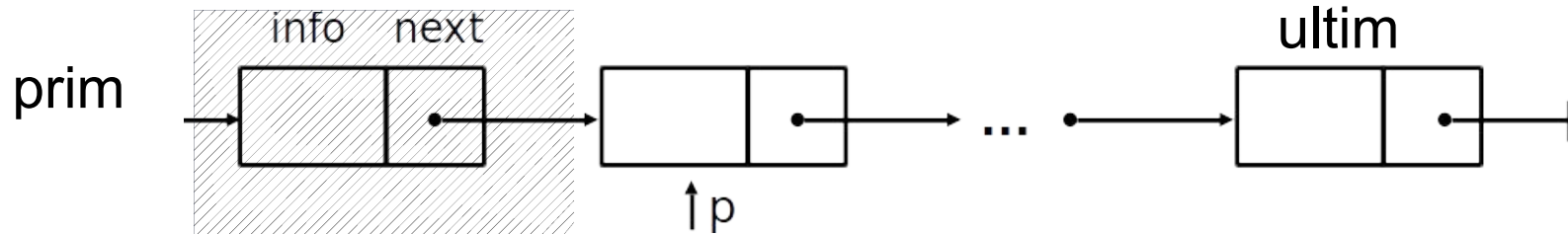
Liste simplu inlantuite

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
```



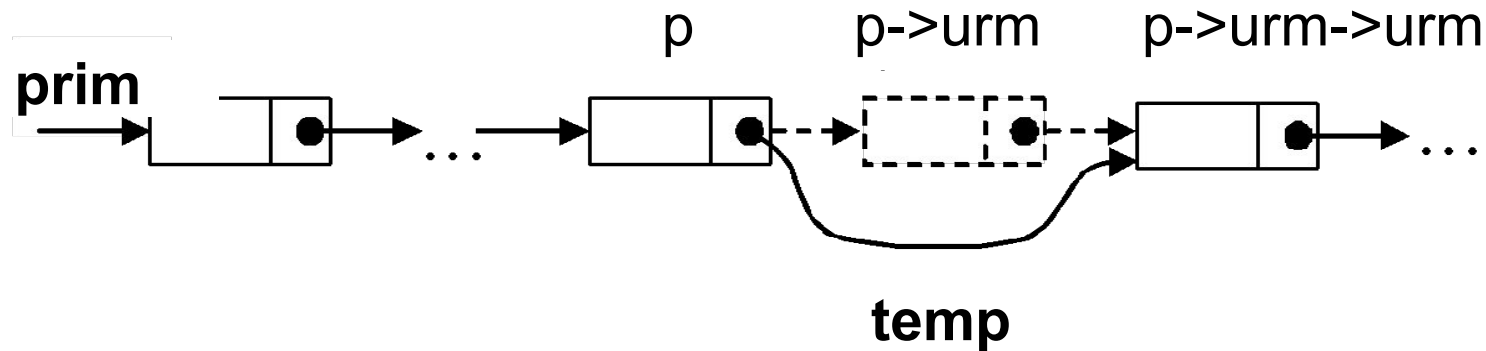
Liste simplu inlantuite

C / C++

Pascal

Stergere

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);
```




Aplicatii

Liste simplu inlantuite

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

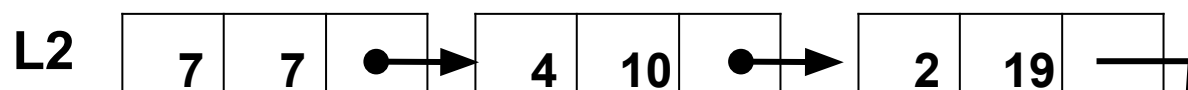
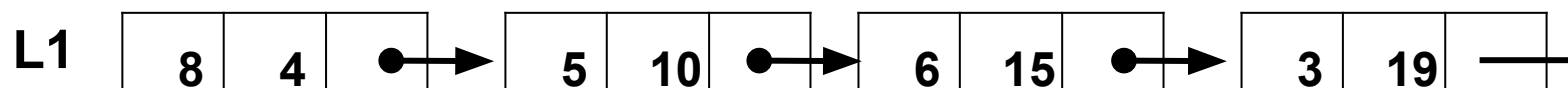
Reprezentarea vectorilor rari

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
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

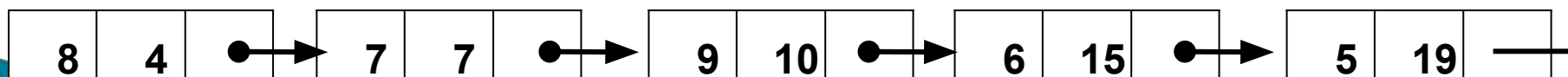
V2	0	0	0	0	0	0	7	0	0	4	0	0	0	0	0	0	0	0	2	0
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Transformarea in liste simplu inlantuite



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

L1+L2



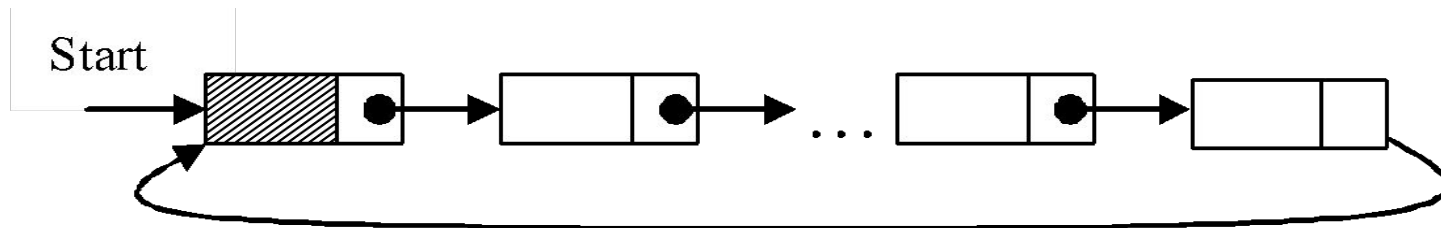


Alte tipuri de liste

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



Nod într-o listă dublu înlănțuită.



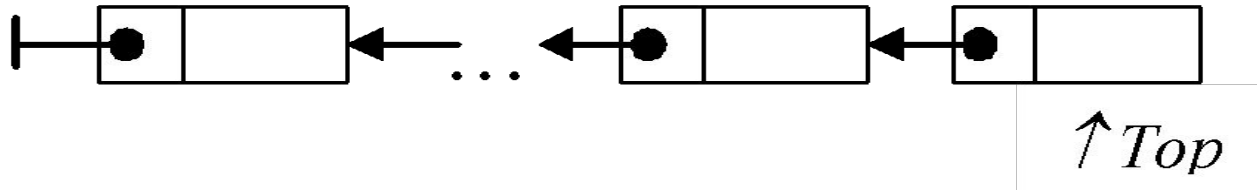
Listă circulară cu nod marcaj.



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";  
}
```



Aplicatii

Stive si cozi

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**.

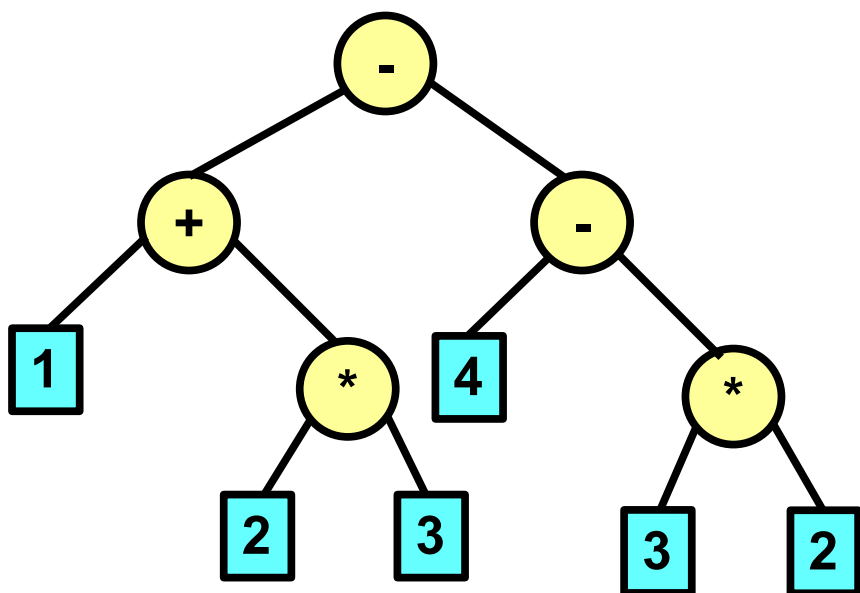


Aplicatii

Stive si cozi

Evaluarea unei expresii în notatie postfixata

Arborele binar asociat expresiei



Parcuregere în preordine:

- + 1 * 2 3 - 4 * 3 2

Parcuregere în inordine:

1 + 2 * 3 - 4 - 3 * 2

Parcuregere în postordine:

1 2 3 * + 4 3 2 * - -



Aplicatii

Stive si cozi

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')

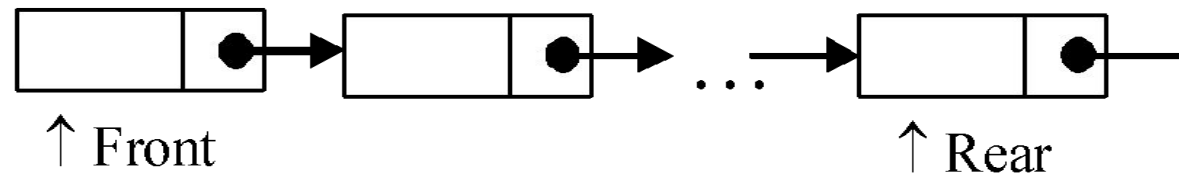
Se repeta **Pas 2** pana la terminarea sirului de caractere introdus.

Pas ultim. Rezultatul este singura valoare aflata in stiva.

[illegible]



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 înțeluite, respectând restricțiile!



Coada in alocare dinamica

Exemplificare operatiilor 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

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

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



Aplicatii

Stive si cozi

Implementare

Sirul lui Hamming

Semnificatia variabilelor utilizate

h - vectorul care stocheaza sirul lui Hamming;

p - indexul asociat acestui vector;

c2 - coada ce contine elementul 2^*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^*x ;

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

m3 - valoarea primului element din coada c3;

c5 - coada ce contine elementul 5^*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

Algoritm

Sirul lui Hamming

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 dintre capetele celor 3 cozi;
- se pune acest minim in vectorul care retine stocheaza sirul lui Hamming;
- se avanseaza in coada (cozile) din care a provenit minimul.



Aplicatii

Stive si cozi

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

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++];  
}
```