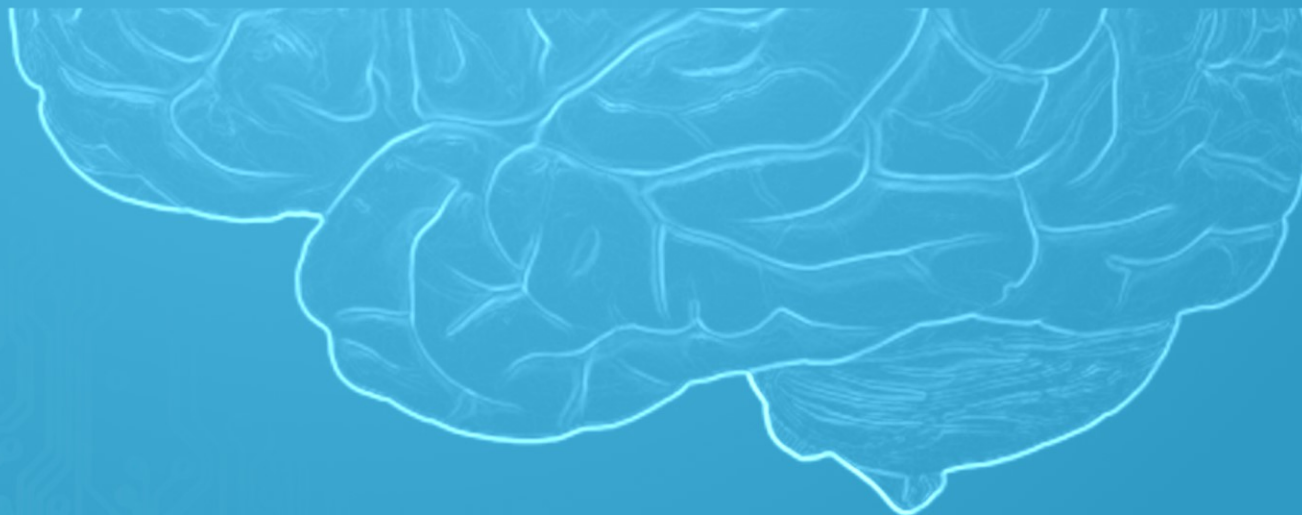




Structuri de date și algoritmi

Liste, Stive, Cozi - 1

Lect. Dr. Ing. Cristian Chilipirea – cristian.chilipirea@mta.ro





Moment de reculegere



Dumnezeu să-l odihnească în pace!

Locotenent-colonel conferențiar
universitar doctor inginer

Cezar Pleșca

1977 - 2021



Lt. col. conf. univ. dr. ing. Cezar Pleșca

- Olimpic Național și Internațional matematică
- Inginer în Știința Calculatoarelor
- Master
- Doctorat
 - Mențiunea “Foarte Onorabil”
 - Sisteme Multimedia Distribuite





Lt. col. conf. univ. dr. ing. Cezar Pleșca

- Prodecan pentru cercetare științifică
- Prodecan pentru învățământ



- 1 carte : “Tehnici de Watermarking pentru Multimedia”
- 2 manuale : “Structuri de Date și Algoritmi”, “Tehnici de Programare”
- 3 capitole în cărți internaționale (Hindawi, Lavoisier, Elsevier)
- 5 articole în jurnale (3 indexate ISI și 2 indexate BDI)
- 18 articole în conferințe internaționale (12 ISI și 6 BDI)
- 10 proiecte de cercetare naționale (2 ca director și 8 ca membru).





Recapitulare - pointeri

- Variabilă ce reține adresa în memorie a unui obiect.
- Declarație
 - ❑ `type *nume;`
 - ❑ `int *myPointer;`
 - ❑ `char *myPointer;`
- Extragere valoare
 - ❑ `int a = *myPointer;`
- Extragere pointer
 - ❑ `int *muPointer = &a;`



Pointeri – operații

- $\text{pointerNou} = \text{pointer} + \text{întreg}$
 - Pointerul mai dreapta cu un întreg număr de elemente
 - Ține cont de dimensiunea elementelor
- $\text{întreg} = \text{pointerA} - \text{pointerB}$
 - numărul de elemente între cei doi pointeri
- $\text{pointer}++$
- $\text{pointer}--$
- $\text{pointerNou} = \text{pointer} + \text{pointer} ?$
- $\text{pointerNou} = \text{pointer} * \text{pointer} ?$



Pointeri – operații

- $\text{pointerNou} = \text{pointer} + \text{întreg}$
 - ▣ Pointerul mai dreapta cu un întreg număr de elemente
 - ▣ Ține cont de dimensiunea elementelor
- $\text{întreg} = \text{pointerA} - \text{pointerB}$
 - ▣ numărul de elemente între cei doi pointeri
- $\text{pointer}++$
- $\text{pointer}--$
- ~~$\text{pointerNou} = \text{pointer} + \text{pointer} ?$~~
- ~~$\text{pointerNou} = \text{pointer} * \text{pointer} ?$~~



Vectori – alocare statică

- `type myVector[N];`
- `char myVector[10];`
 - ▣ 10 elemente alocate
 - ▣ Numerotate de la 0

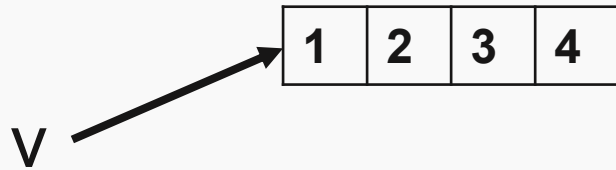
[0]	[1]	[2]	[3]	[4]	[5]	[6] ...
1	2	3	4	5	6	7

`myVector[3] == 4`



Vectori - alocare dinamică

```
int* v;  
v = (int*)malloc(N * sizeof(int));
```



`v[3] == 4`

<code>[0]</code>	<code>[1]</code>	<code>[2]</code>	<code>[3]</code>
1	2	3	4



Matrici

O matrice reprezintă un set de variabile grupate

Poate fi:

- 1D – vector (N elemente)
- 2D – matrice ($N*M$ elemente)
- 3+D – matrice multidimensională ($N*M*...*Z$ elemente)

Elementele sunt adresabile direct (caz 2D):

- `matrice[i][j]`; Elementul de pe poziția $i*N+j$
- `matrice[3][4]`; Elementul de pe poziția $3*N+4$



Matrice

- `type myMatrix[N][M];`
- `int myMatrix[10][20];`
 - 10*20 elemente alocate
 - numerotare de la 0

	<code>[] [0]</code>	<code>[] [1]</code>	<code>[] [2]</code>	<code>[] [3]</code>	<code>[] [4]</code>	<code>[] [5]</code>	<code>[] [6] ...</code>
<code>[0] []</code>	1	2	3	4	5	6	7
<code>[1] []</code>	8	9	10	11	12	13	14
<code>[2] []</code>	15	16	17	18	19	20	21
<code>[3] []</code>	22	23	24	25	26	27	28
<code>[4] []</code>	29	30	31	32	33	34	35

...

`myMatrix[3][4] == 26`



Matrice – alocare statică

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	8	9	10	11
[2]	15	16	17	18
[3]	22	23	24	25

`int myMatrix[4][4]`

- Rezultă în zonă continuă de memorie, rândurile sunt așezate unul după altul.

1	2	3	4	8	9	10	11	15	16	17	18	22	23	24	25
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----



Matrice – alocare statică

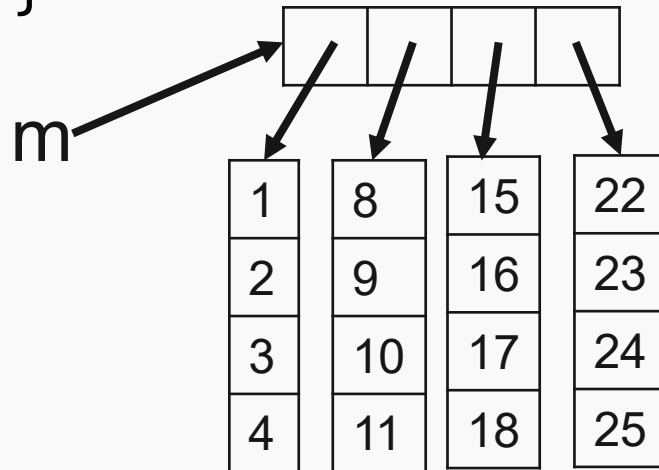
- Cache friendly
- Limitată ca spațiu (suntem pe stack)

1	2	3	4	8	9	10	11	15	16	17	18	22	23	24	25
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----



Matrice - alocare dinamică

```
int** m;  
m = (int**)malloc(N * sizeof(int *));  
for (int i = 0; i < N; i++) {  
    m[i] = (int*)malloc(N * sizeof(int));  
}
```

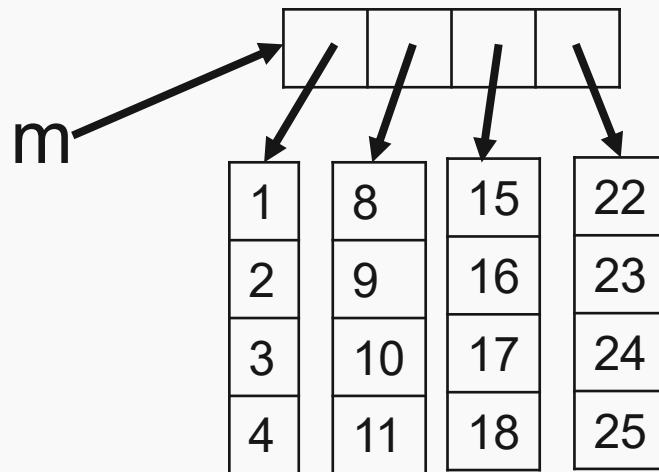


	[][0]	[][1]	[][2]	[][3]
[0][]	1	2	3	4
[1][]	8	9	10	11
[2][]	15	16	17	18
[3][]	22	23	24	25



Matrice - alocare dinamică

- Spațiu foarte mare (suntem pe heap)
- Posibilitate ca fiecare rând să aibă altă mărime (periculos)
- Nu este cache friendly



	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	8	9	10	11
[2]	15	16	17	18
[3]	22	23	24	25



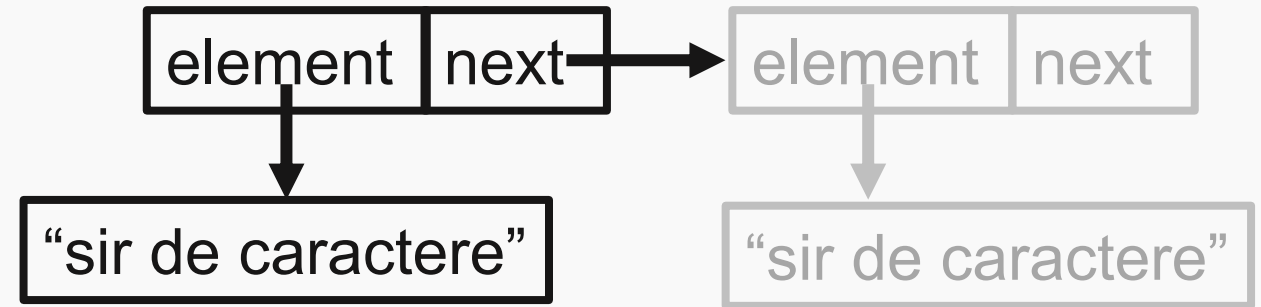
Struct

```
struct name {  
    type1 name1;  
    type2 name2;  
    ...  
};  
typedef struct name {  
    type1 name1;  
    type2 name2;  
    ...  
} newName;
```



Liste înlanțuite

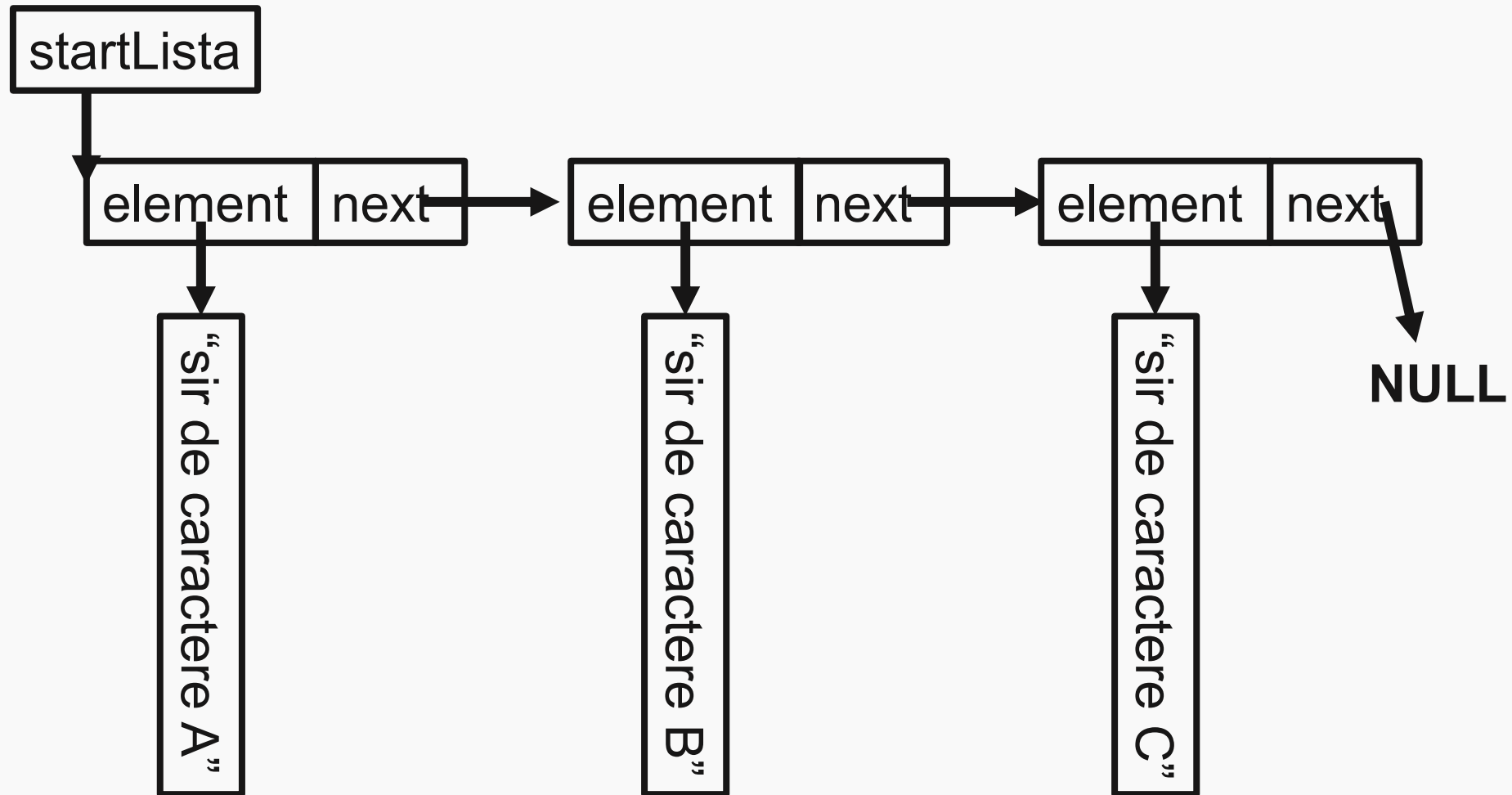
```
struct node {  
    char* element;  
    struct node* next;  
};
```



```
typedef struct node {  
    char* element;  
    struct node* next;  
}listNode;
```

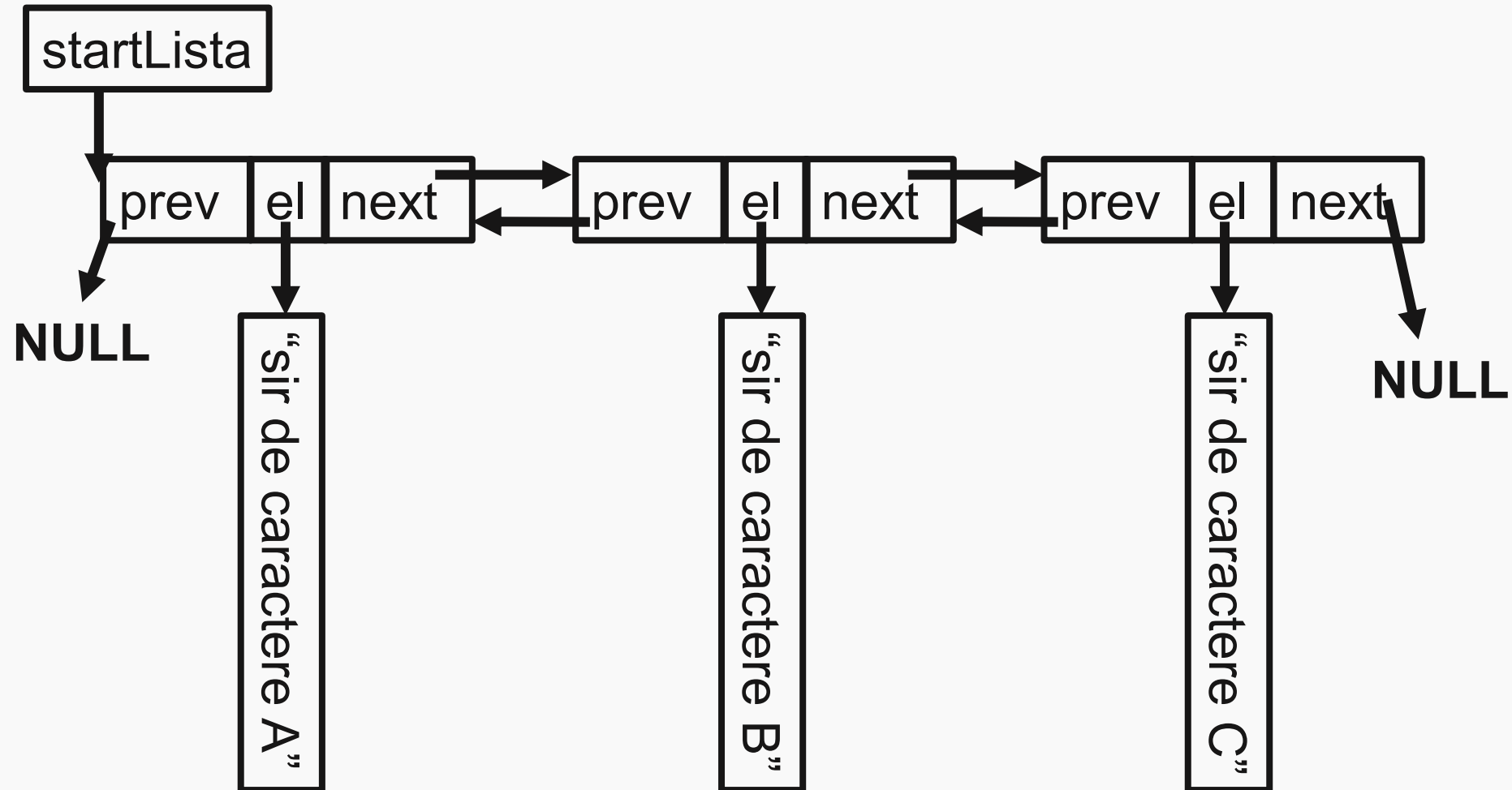


Liste înlănțuite



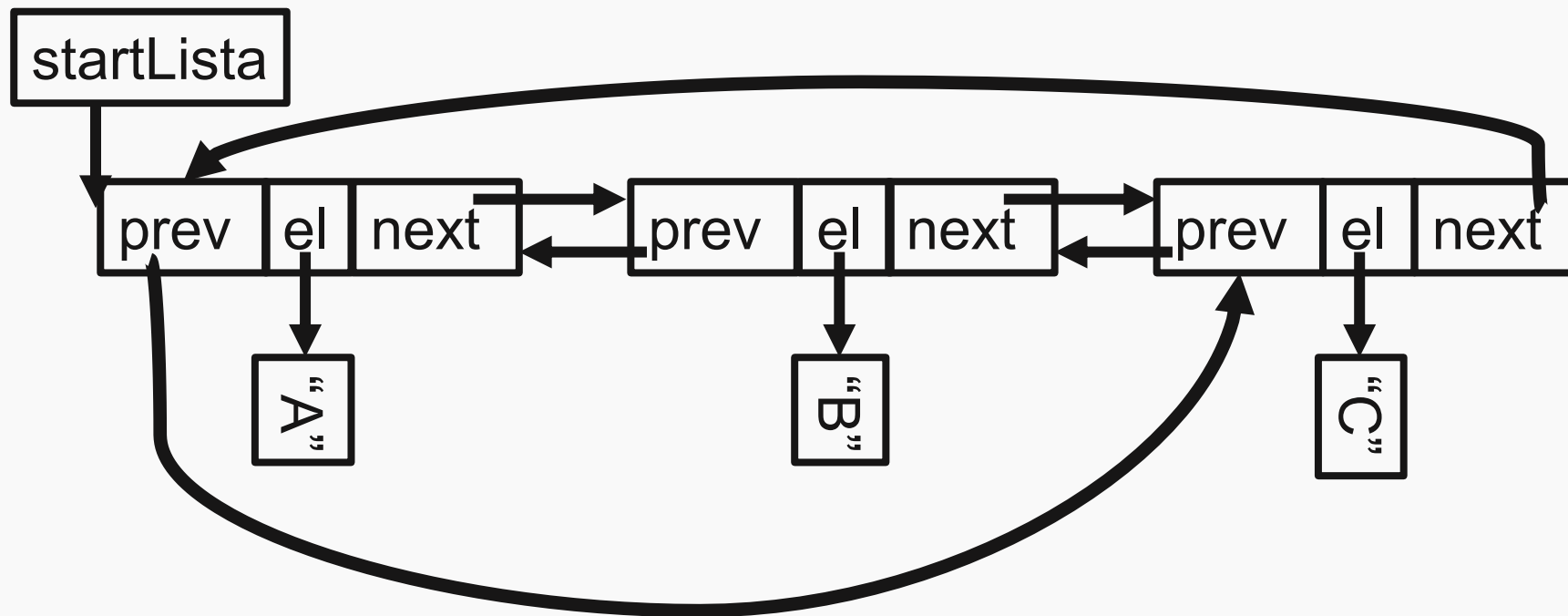


Liste dublu-înlanțuite





Liste circulare





Operații cu liste

- Accesarea unui element
- Inserare element
- Ștergere element



Accesarea unui element din listă

```
listNode* getNode(listNode* listNode, int poz) {  
    for (int i = 0; i < poz; i++) {  
        listNode = listNode->next;  
        if (listNode->next == NULL)  
            break;  
    }  
    return listNode;  
}
```



Inserare element început de listă

```
void insertNodeHeadOfList(listNode** listStart, char* element) {  
    listNode* node = (listNode*)malloc(sizeof(listNode));  
    if (node == NULL) {  
        printf("ERROR: CAN NOT ALLOCATE RAM\n");  
        return;  
    }  
    node->next = *listStart;  
    node->element = element;  
    *listStart = node;  
}
```



Inserare element în listă la o poziție dată

```
void insertNodeInList(listNode** listStart, char* element, int poz) {  
    if (poz == 0) {  
        insertNodeHeadOfList(listStart, element);  
        return;  
    }  
    listNode* node = (listNode*)malloc(sizeof(listNode));  
    if (node == NULL) {  
        printf("ERROR: CAN NOT ALLOCATE RAM\n");  
        return;  
    }  
  
    listNode* aux = getNode(*listStart, poz - 1);  
  
    node->next = aux->next;  
    node->element = element;  
    aux->next = node;  
}
```




Ștergere element început de listă

```
void removeNodeHeadOfList(listNode** listStart) {  
    // free((*listStart)->element);  
    if (*listStart == NULL)  
        return;  
    listNode* aux = (*listStart);  
    *listStart = (*listStart)->next;  
    free(aux);  
}
```



Ștergere element din listă de la o poziție dată

```
void removeNodeFromList(listNode** listStart, int poz) {  
    if (poz == 0) {  
        removeNodeHeadOfList(listStart);  
        return;  
    }  
    listNode* aux = getNode(*listStart, poz-1);  
    if (aux->next != NULL) {  
        listNode* aux1 = aux->next;  
        aux->next = aux->next->next;  
        // free(aux1->element);  
        free(aux1);  
    }  
}
```



Complexități operații cu liste

- Accesarea unui element - $O(N)$
- Inserare element la capăt - $O(1)$
- Inserare element la o poziție - $O(N)$
 - Accesare + inserare
- Ștergere element la capăt - $O(1)$
- Ștergere element de la o poziție - $O(N)$
 - Accesare + ștergere



Liste vs Vectori

	Vector	Listă
Complexitate acces	$O(1)$	$O(N)$
Complexitate inserție	$O(N)$	$O(N)$
Complexitate inserție capete	$O(N)/O(1)$	$O(1)$
Complexitate ștergere	$O(N)$	$O(N)$
Complexitate ștergere capete	$O(N)/O(1)$	$O(1)$
Alocare spațiu	Doar la început	Oricând și oricât



De ce la liste avem inserție/ștergere $O(1)$ la ambele capete?



De ce la liste avem inserție/ștergere $O(1)$ la ambele capete?

Putem să avem pointeri HeadList și EndList



Vectori Alocați Dinamic



Vectori Alocați Dinamic

- O mărime de start – poate fi și un element.
- Când vectorul este plin și se încearcă un insert, se dublează dimensiunea sa.

1	2	3	4
---	---	---	---

push(**5**)

1	2	3	4	5			
---	---	---	---	---	--	--	--



Vectori Alocați Dinamic

Push()	Copiere	Mărime vector
1	0	1
2	1	2
3	2	4
4	0	4
5	4	8
6	0	8
7	0	8
8	0	8
9	8	16
10	0	16



Complexitate amortizată inserție vector dinamic

- 3 push 2 + 1 copieri
- 5 push 4 + 2 + 1 copieri
- 9 push 8 + 4 + 2 + 1 copieri
- ...
- $2^n + 1$ push $2^n + 2^{n-1} + 2^{n-2} + \dots + 2 + 1 = 2^{n+1} - 1$ copieri

$$\lim_{n \rightarrow \infty} \frac{2^{n+1} - 1}{2^{n+1}} = 2 = O(1)$$



Self-organizing lists



Self-organizing lists

- Premisă: Nu toate elementele sunt echiprobabile la căutare.
 - ▣ Unele elemente sunt accesate mult mai des decât altele.
- Dacă elementele sunt **echiprobabile** la căutare:
 - ▣ Probabilitatea de a găsi un element p_i este egală cu probabilitatea de a găsi orice alt element p_j

$$\sum_{i=0}^N p_i = 1 \Rightarrow p_i = \frac{1}{N}$$

- ▣ Timpul mediu de găsim a unui element este

$$\sum_{i=0}^N i * p_i = \frac{1}{N} O(N^2) = O(N)$$



Ordinea listei contează

- Presupunem că pentru un $i < j$ avem $p_i < p_j$
- $C = 1 p_1 + \dots + i \textcolor{teal}{p}_i + j \textcolor{red}{p}_j + \dots + N p_N$
- Dacă interschimbăm elementele de pe pozițiile i și j :
- $C' = 1 p_1 + \dots + i \textcolor{red}{p}_j + j \textcolor{teal}{p}_i + \dots + N p_N$

$$\begin{aligned} C' - C &= (i \textcolor{red}{p}_j + j \textcolor{teal}{p}_i) - (i \textcolor{teal}{p}_i + j \textcolor{red}{p}_j) = \\ &= i(\textcolor{red}{p}_j - \textcolor{teal}{p}_i) - j(\textcolor{teal}{p}_i - \textcolor{red}{p}_j) = (i - j)(\textcolor{red}{p}_j - \textcolor{teal}{p}_i) < 0 \Rightarrow \\ &\quad C > C' \end{aligned}$$



Self-organizing lists

probabilități descrescătoare

$$p_i = \frac{1}{2^i}$$

$$\begin{aligned} C &= 1 p_1 + \dots + N p_N = 1 \frac{1}{2} + \dots + N \frac{1}{2^N} = \\ &= \sum_{i=1}^N \frac{i}{2^i} = 2 \quad (n \rightarrow \infty) \end{aligned}$$



Mecanisme de ordonare automată

- Sortare la intervale prestabilite de timp
- Când un element este accesat acesta se mută în capul listei *moveToFront*
- Când un element este accesat acesta își schimbă locul cu precedentul *Transpose*
- Numărăm fiecare accesare a unui element și schimbăm ordinea pentru a păstra ordonare dupa aceste valori.



Coadă de priorități



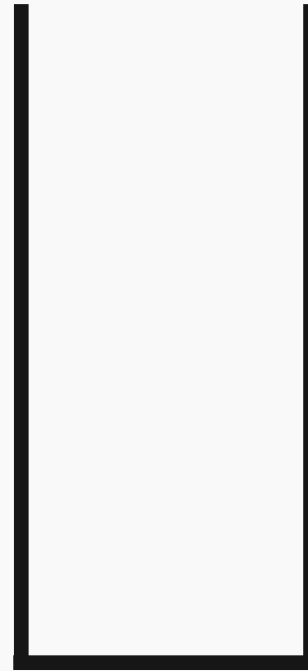
Stack - Stivă



Stack - Stivă

LIFO – Last In First Out

push(1)

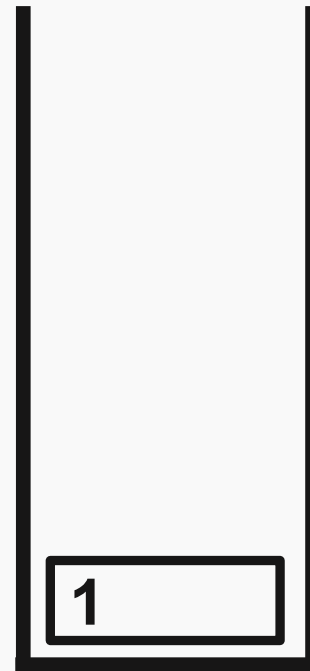




Stack - Stivă

LIFO – Last In First Out

push(2)

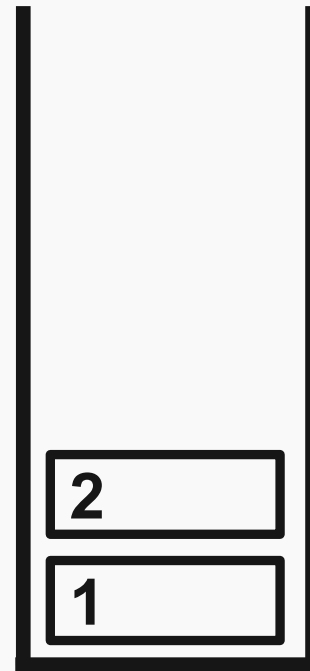




Stack - Stivă

LIFO – Last In First Out

push(3)

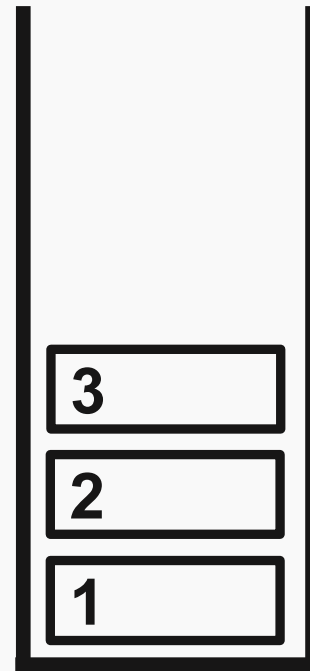




Stack - Stivă

LIFO – Last In First Out

push(4)

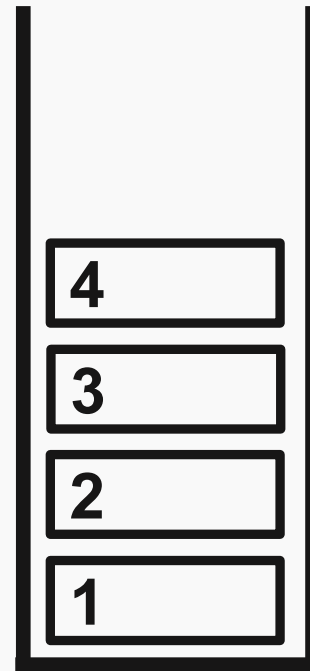




Stack - Stivă

LIFO – Last In First Out

push(**5**)

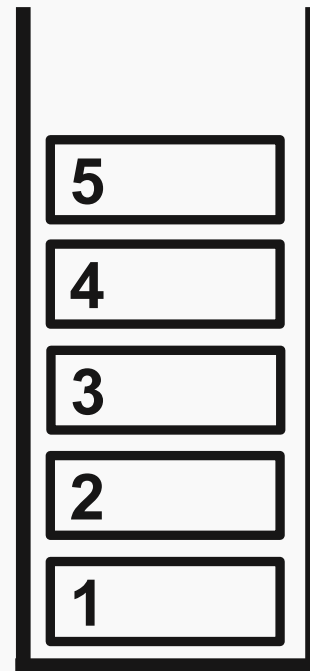




Stack - Stivă

LIFO – Last In First Out

pop()

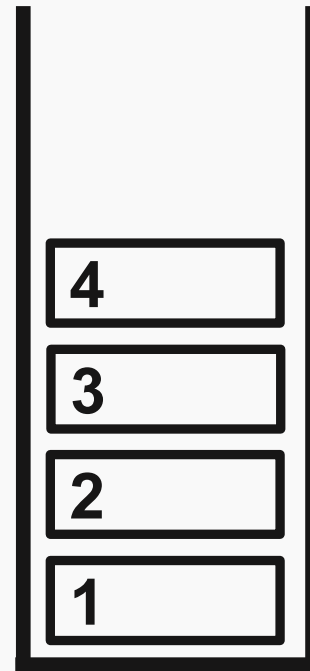




Stack - Stivă

LIFO – Last In First Out

5 pop()

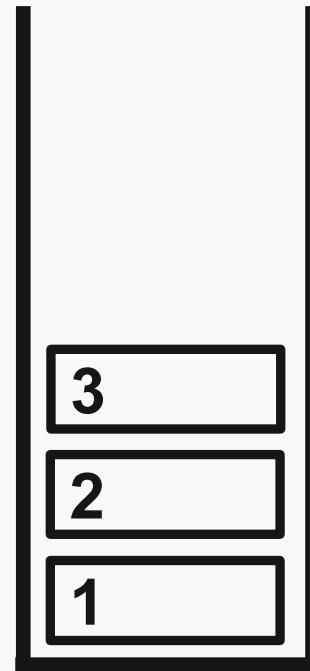




Stack - Stivă

LIFO – Last In First Out

4 pop()

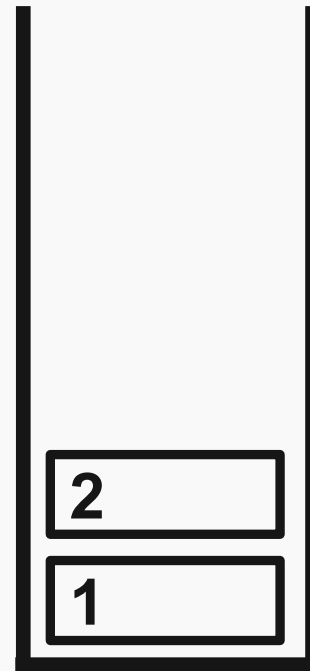




Stack - Stivă

LIFO – Last In First Out

3

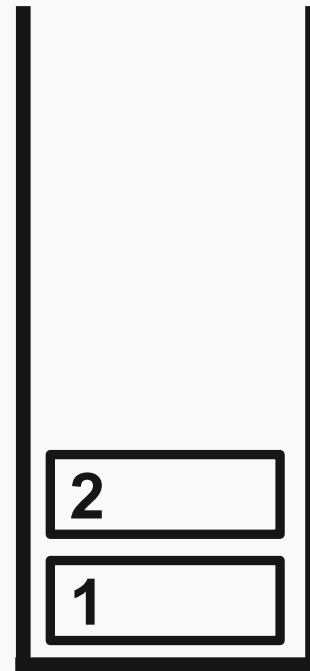




Stack - Stivă

LIFO – Last In First Out

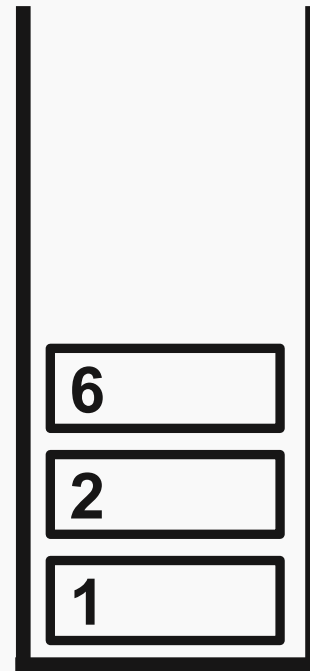
push(6)





Stack - Stivă

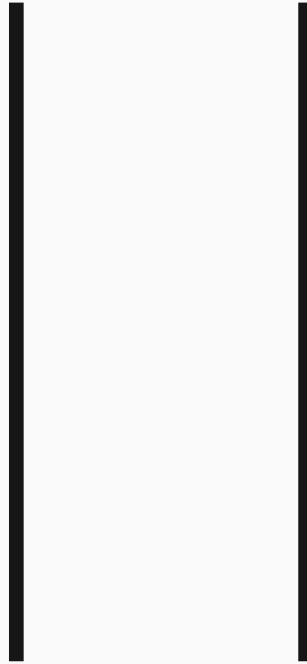
LIFO – Last In First Out





Queue - Coadă

FIFO – First In First Out

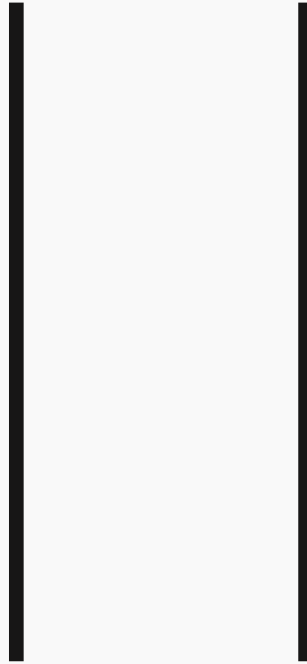




Queue - Coadă

FIFO – First In First Out

push(1)

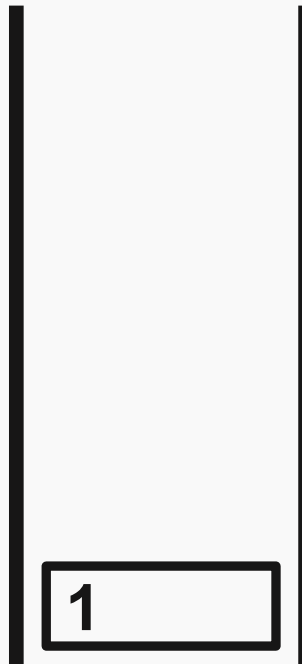




Queue - Coadă

FIFO – First In First Out

push()

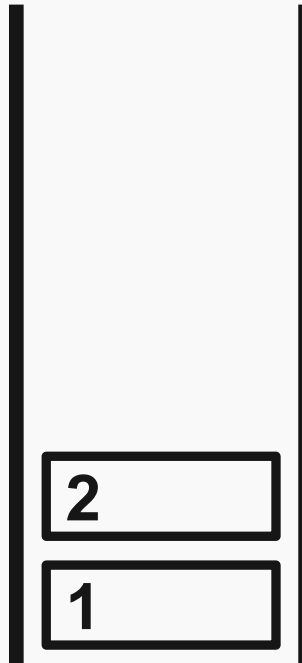




Queue - Coadă

FIFO – First In First Out

push(3)

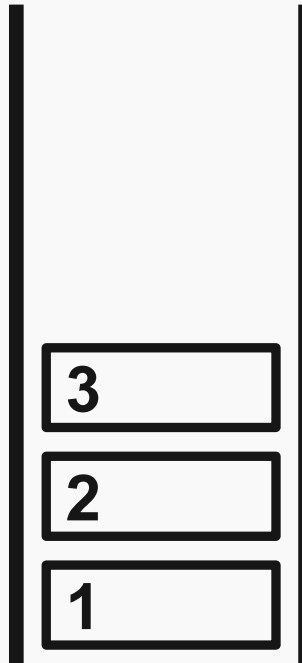




Queue - Coadă

FIFO – First In First Out

push()

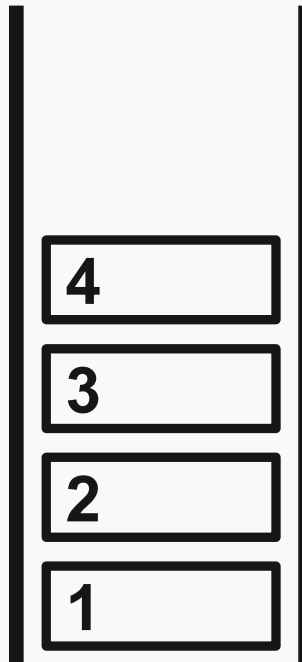




Queue - Coadă

FIFO – First In First Out

push(5)

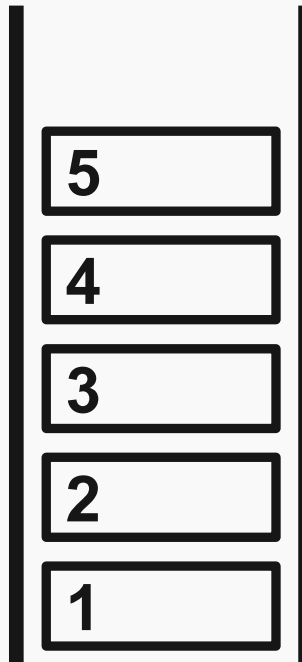




Queue - Coadă

FIFO – First In First Out

pop()

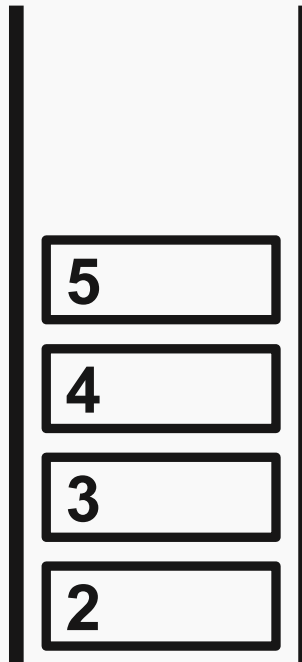




Queue - Coadă

FIFO – First In First Out

1 pop()

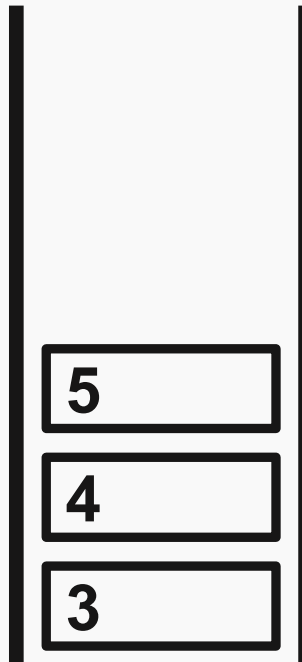




Queue - Coadă

FIFO – First In First Out

2 pop()

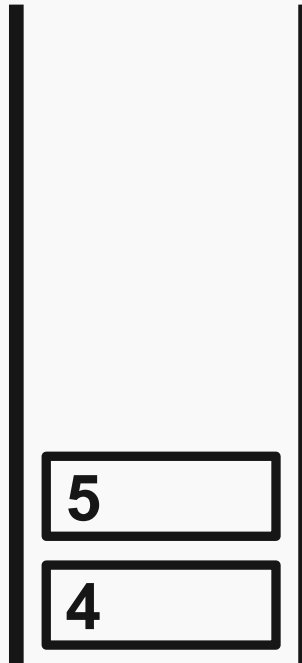




Queue - Coadă

FIFO – First In First Out

3

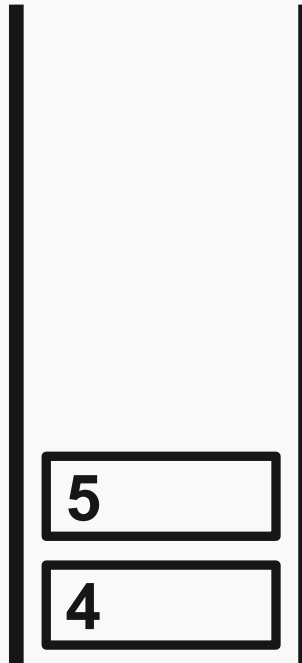




Queue - Coadă

FIFO – First In First Out

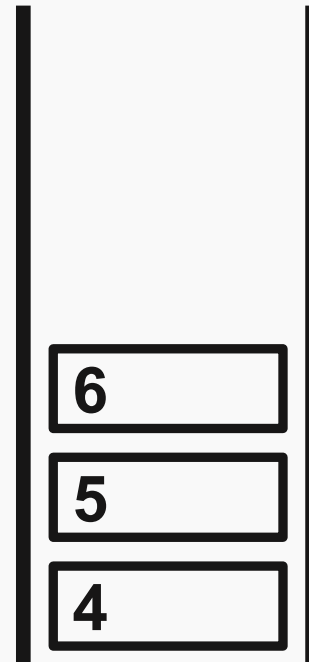
push()





Queue - Coadă

FIFO – First In First Out





Corectitudinea unui expresii

- $[(1+2)-3*(2+1)]$
- $\{[(())]\}$