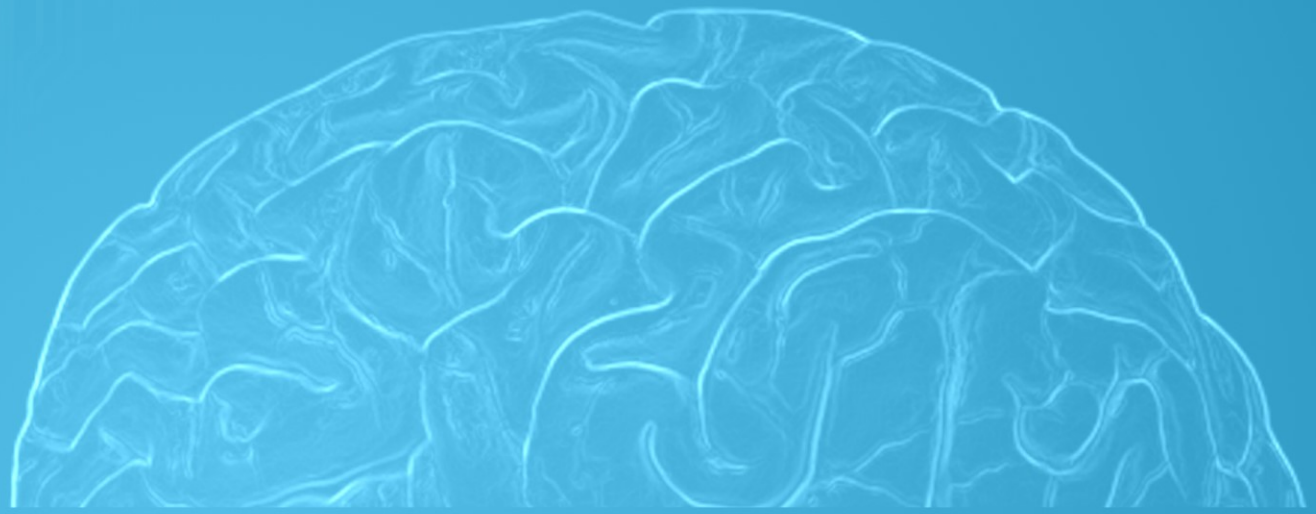
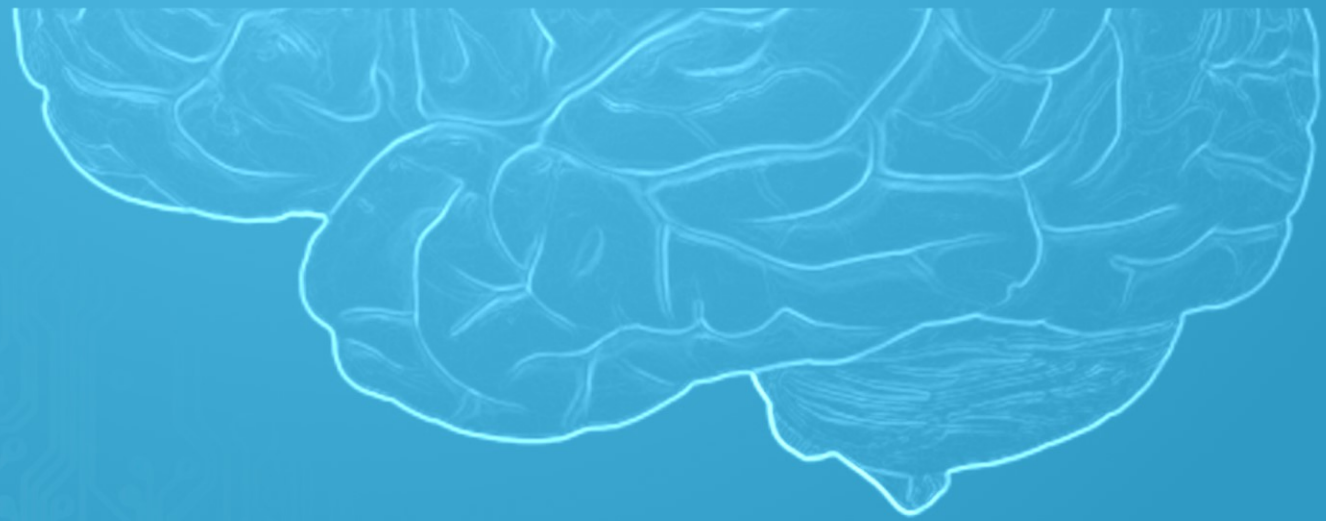


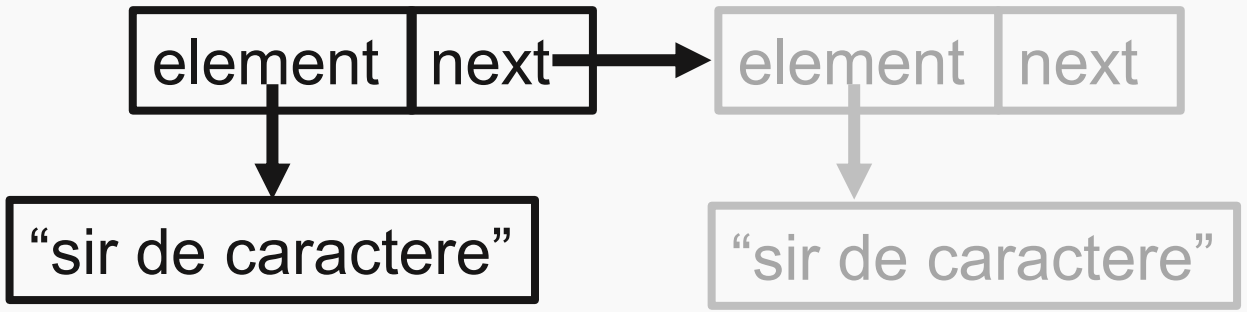
Structuri de date și algoritmi
Liste, Stive, Cozi - 2

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



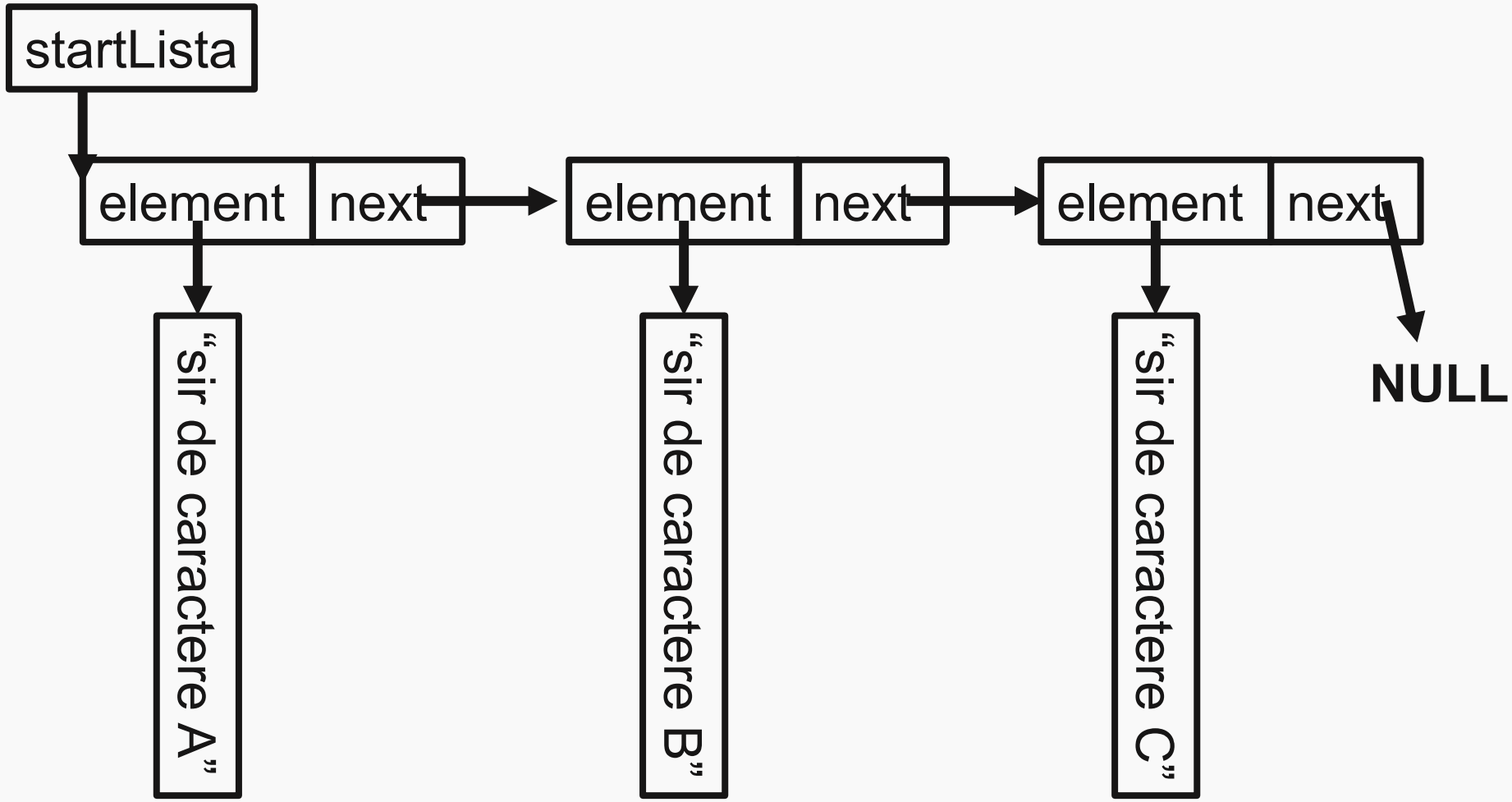
Liste înlănțuite

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




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

Liste înlănțuite



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.

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 $2^n + 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = 2^{n+1} - 1$ copieri

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

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 găsire 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 $p_i < p_j$
- $C = 1\,p_1 + \ \ldots + i\,\boldsymbol{p_i} + j\,\boldsymbol{p_j} + \ \ldots + Np_N$

- Dacă interschimbăm elementele de pe pozițiile i și j:
- $C' = 1\,p_1 + \ \ldots + i\,\boldsymbol{p_j} + j\,\boldsymbol{p_i} + \ \ldots + Np_N$

$$C' - C = \left(i\,\boldsymbol{p_j} + j\,\boldsymbol{p_i}\right) - \left(i\,\boldsymbol{p_i} + j\,\boldsymbol{p_j}\right) =$$

$$= i\left(\boldsymbol{p_j} - \boldsymbol{p_i}\right) - j\left(\boldsymbol{p_i} - \boldsymbol{p_j}\right) = (i-j)\left(\boldsymbol{p_j} - \boldsymbol{p_i}\right) < 0 \Rightarrow C > C'$$

Self-organizing lists
probabilități descrescătoare

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

$$C = 1\,p_1 + \ \ldots + Np_N = 1\frac{1}{2} + \ \ldots + N\frac{1}{2^N} =$$

$$= \sum_{i=1}^{N} \frac{i}{2^i} = 2\ (\mathrm{n} \to \infty)$$

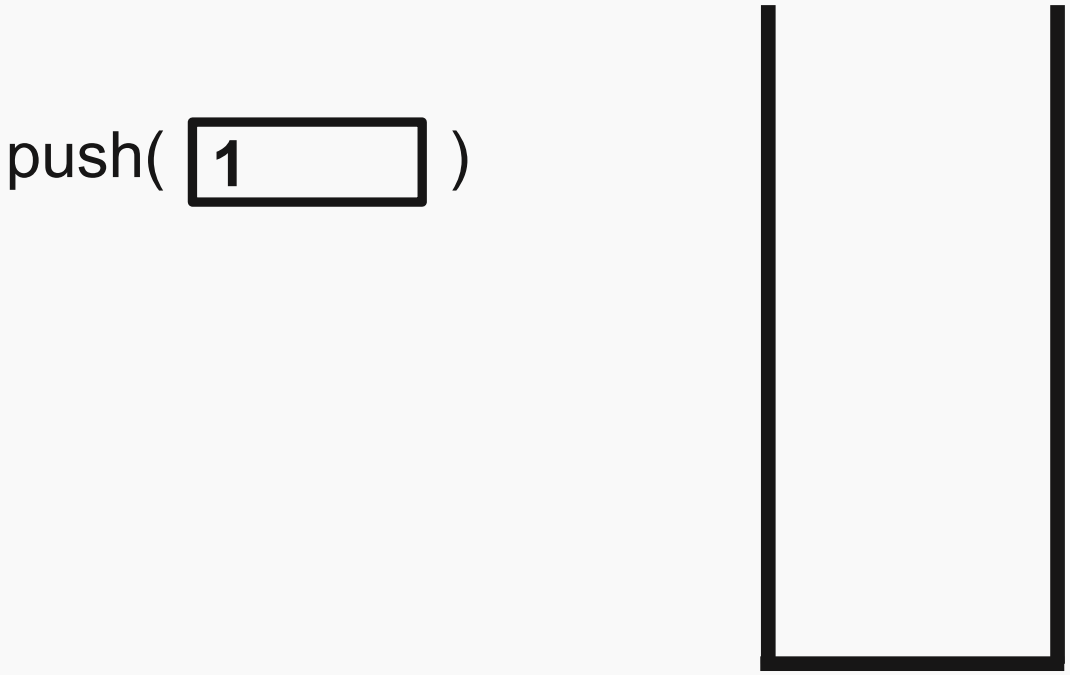
Mecanisme de ordonare automată

- 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.

Stack - Stivă

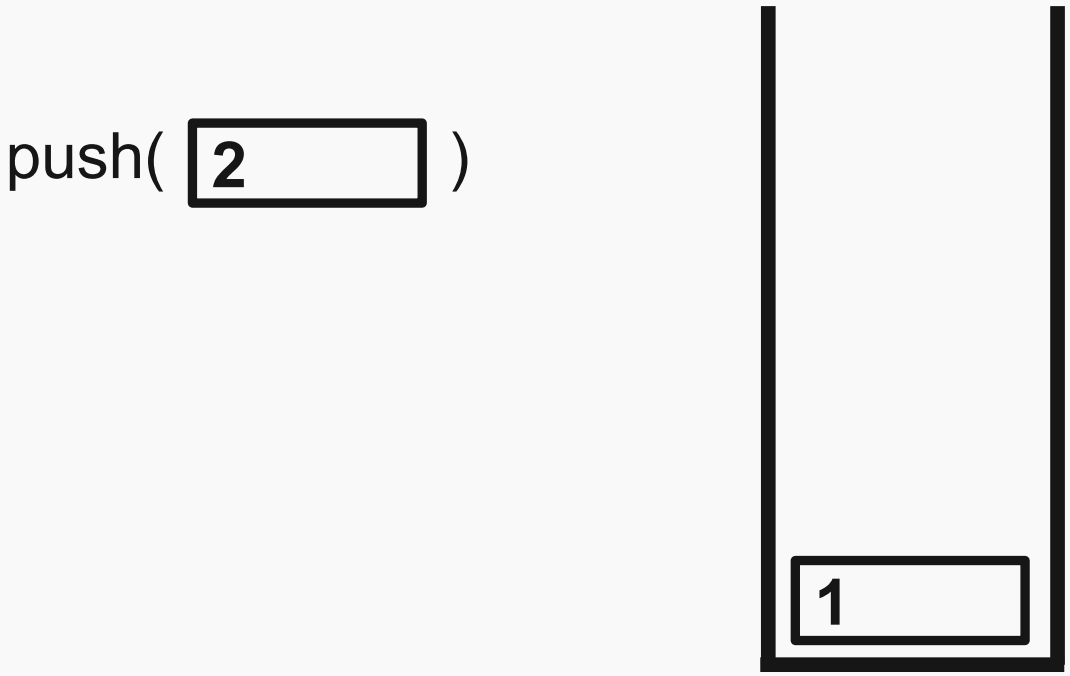
Stack - Stivă

LIFO – Last In First Out



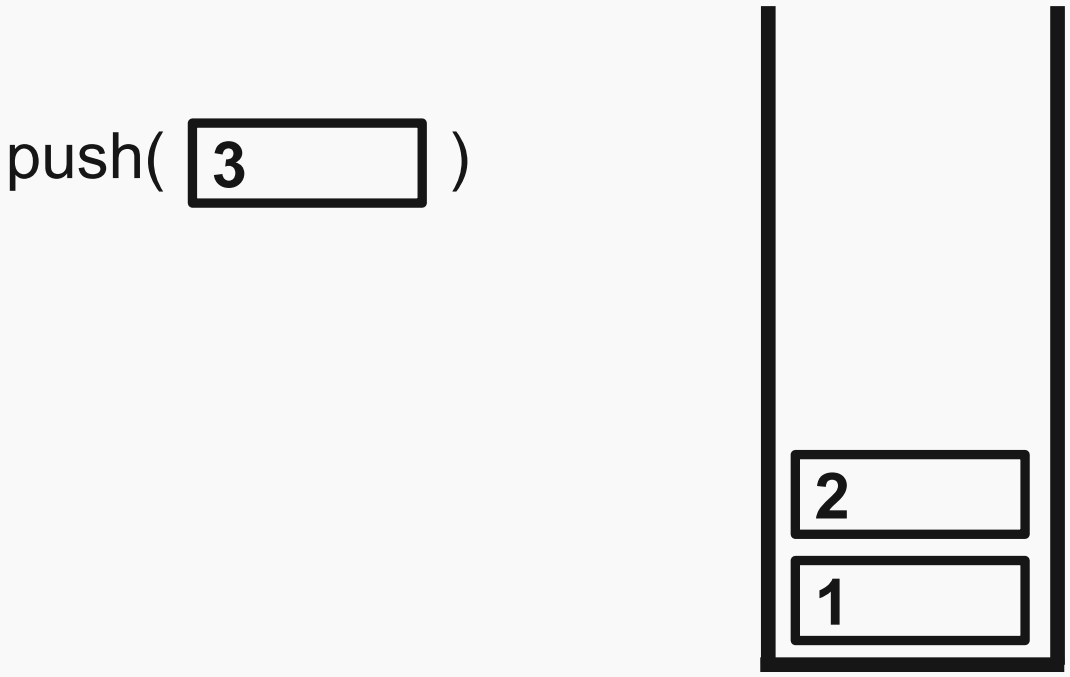
Stack - Stivă

LIFO – Last In First Out



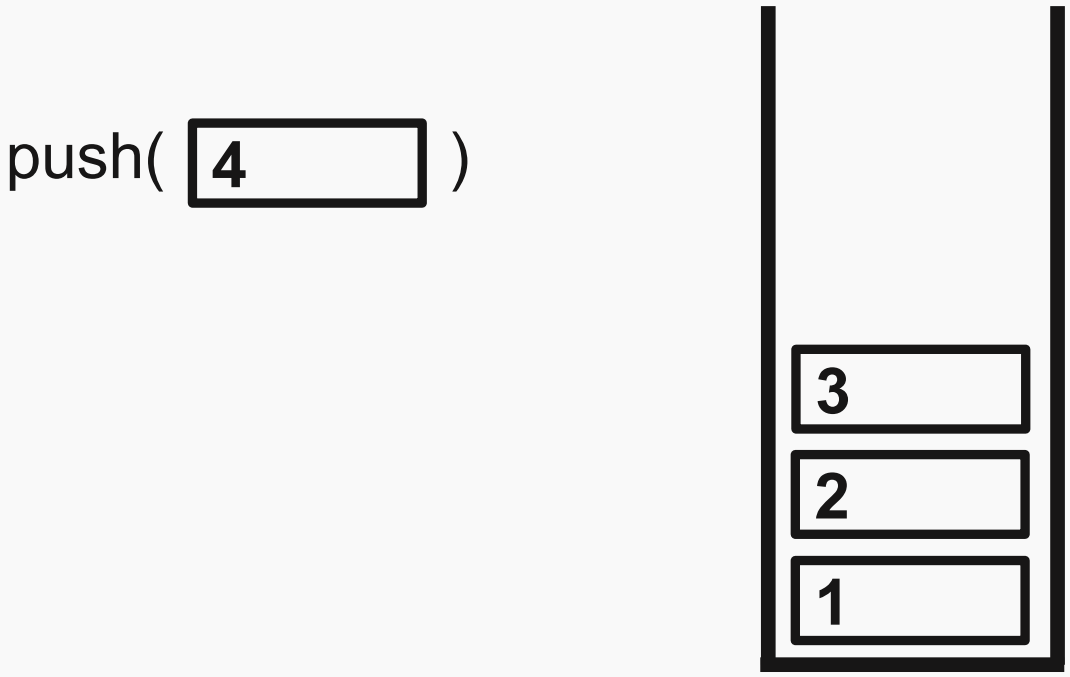
Stack - Stivă

LIFO – Last In First Out



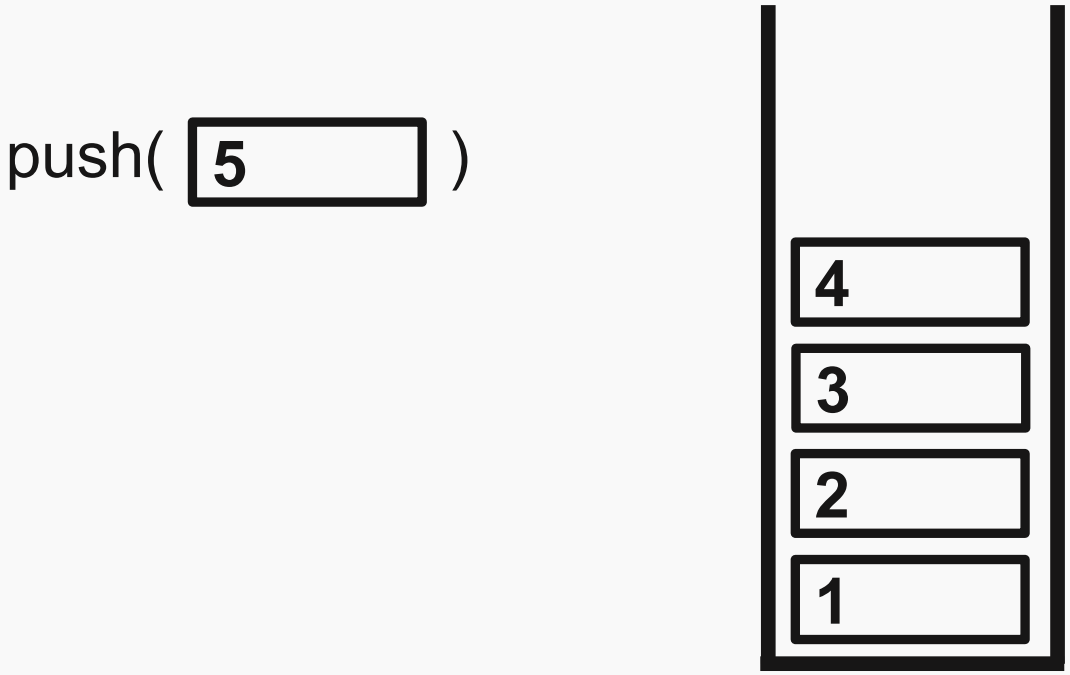
Stack - Stivă

LIFO – Last In First Out



Stack - Stivă

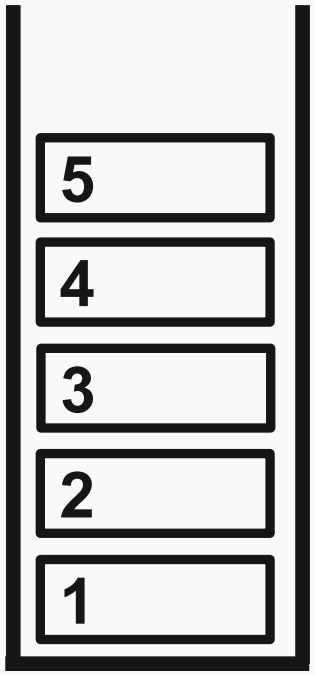
LIFO – Last In First Out



Stack - Stivă

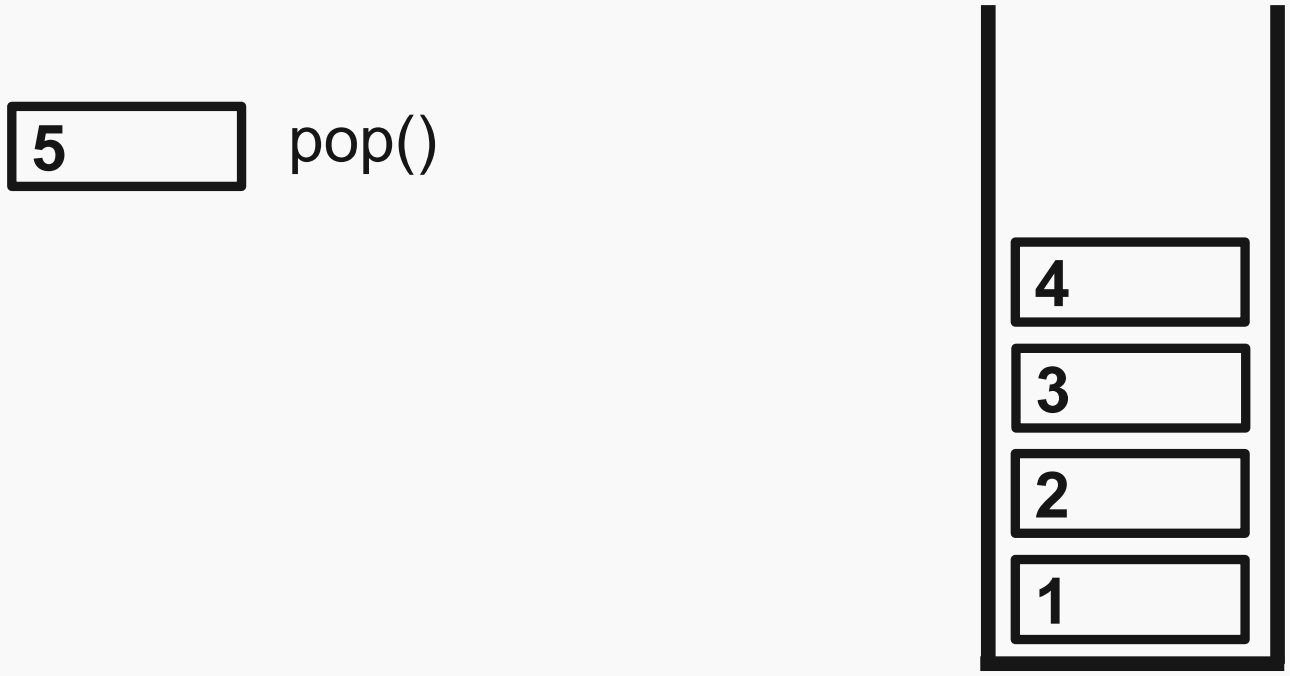
LIFO – Last In First Out

pop()



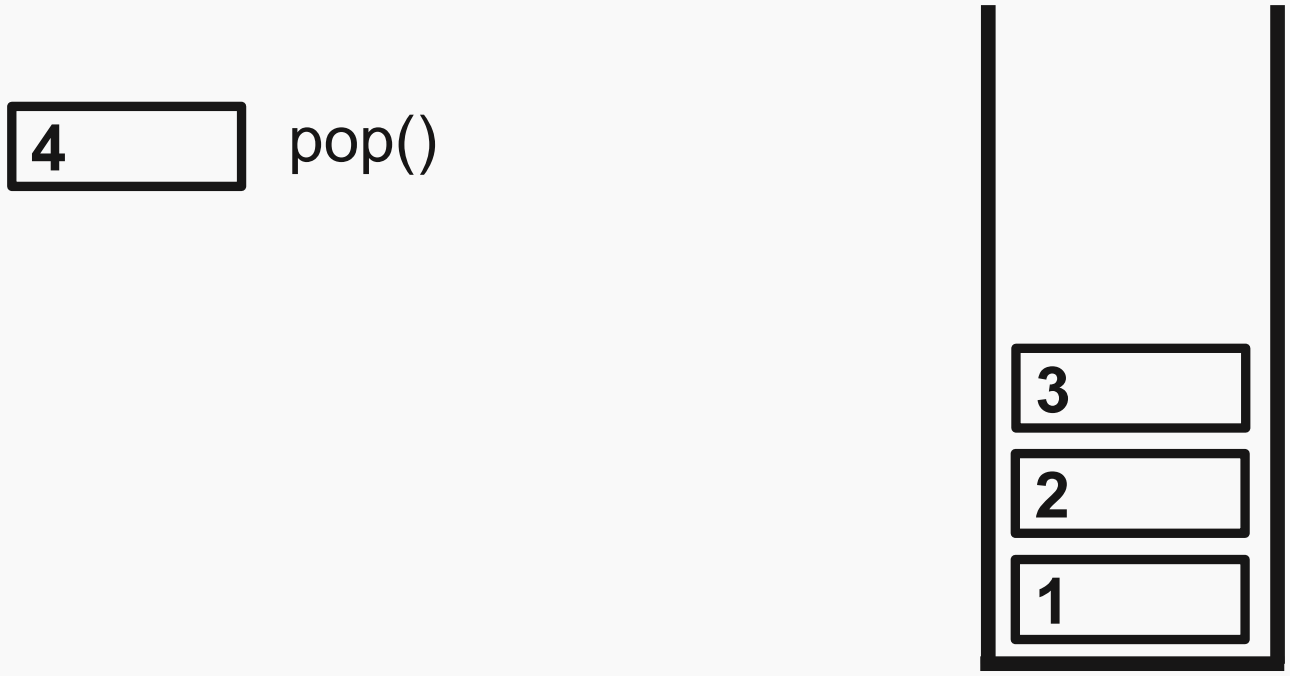
Stack - Stivă

LIFO – Last In First Out



Stack - Stivă

LIFO – Last In First Out

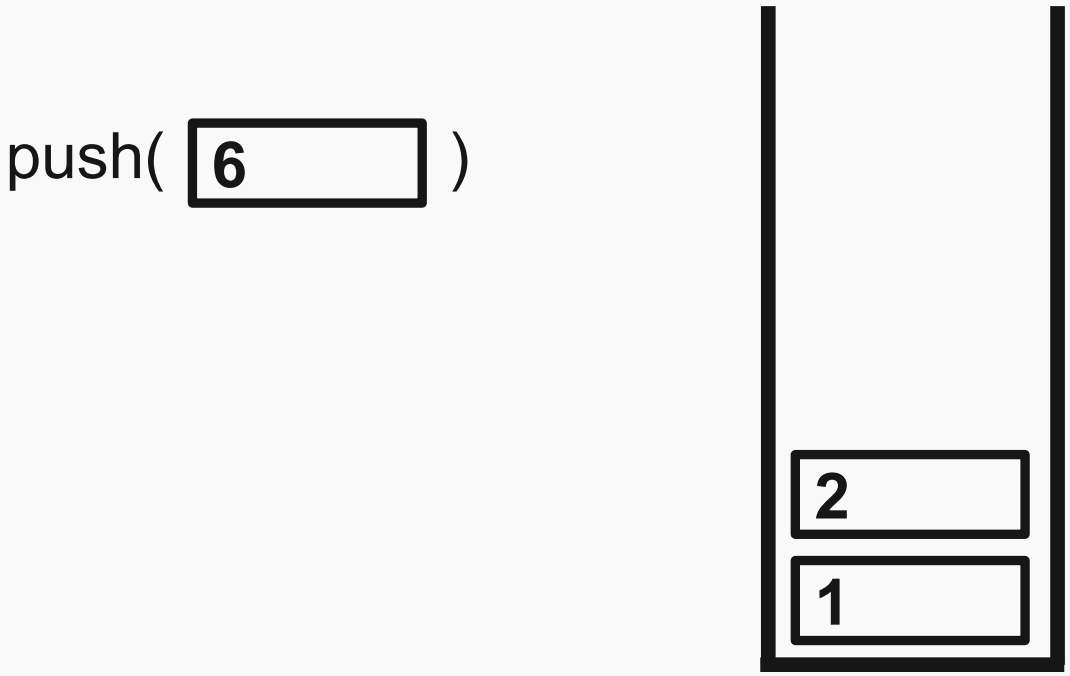


Stack - Stivă

LIFO – Last In First Out

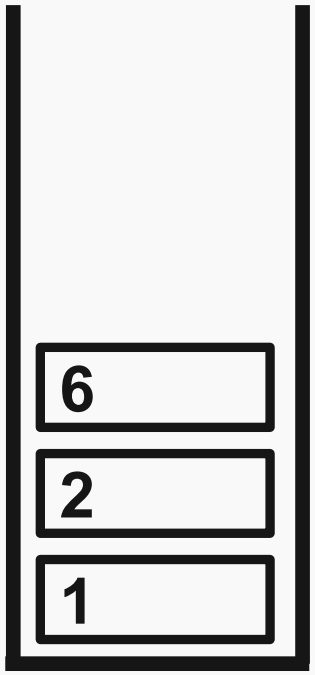
Stack - Stivă

LIFO – Last In First Out



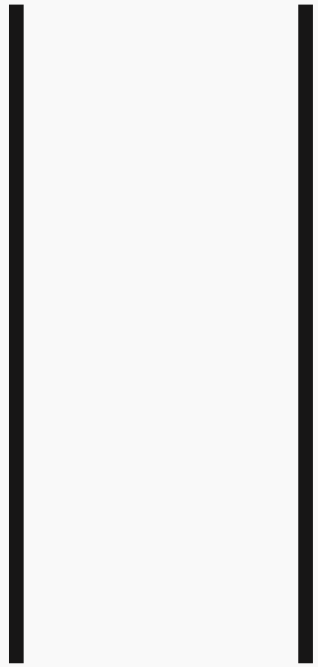
Stack - Stivă

LIFO – Last In First Out



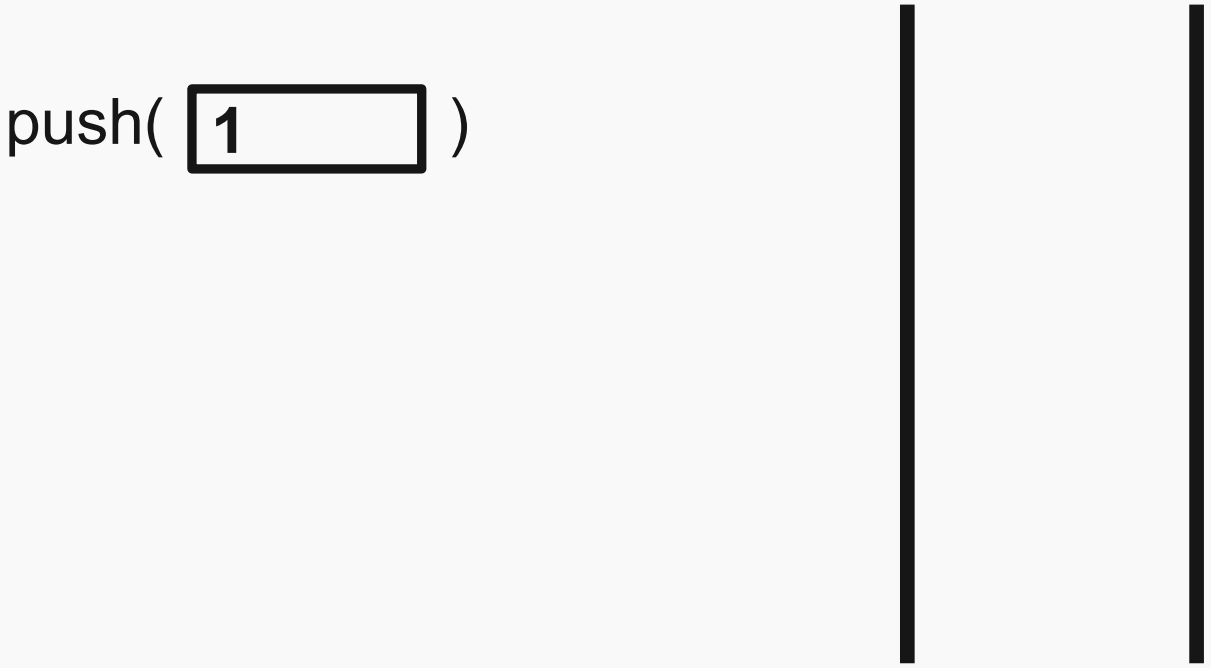
Queue - Coadă

FIFO – First In First Out



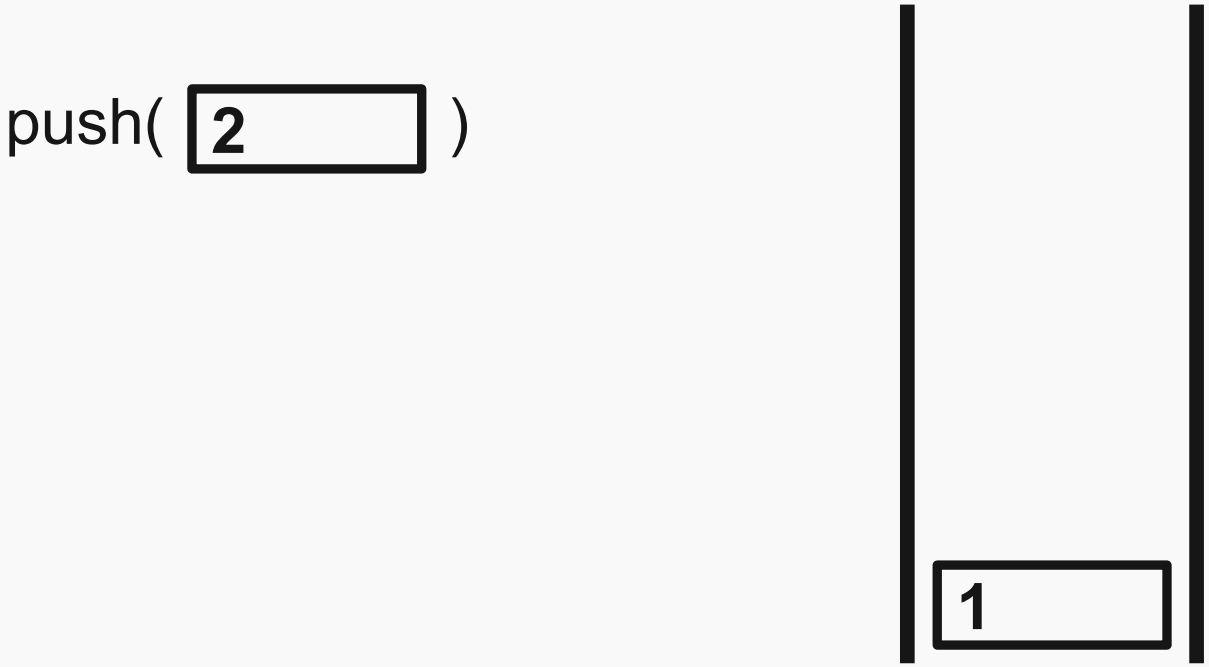
Queue - Coadă

FIFO – First In First Out



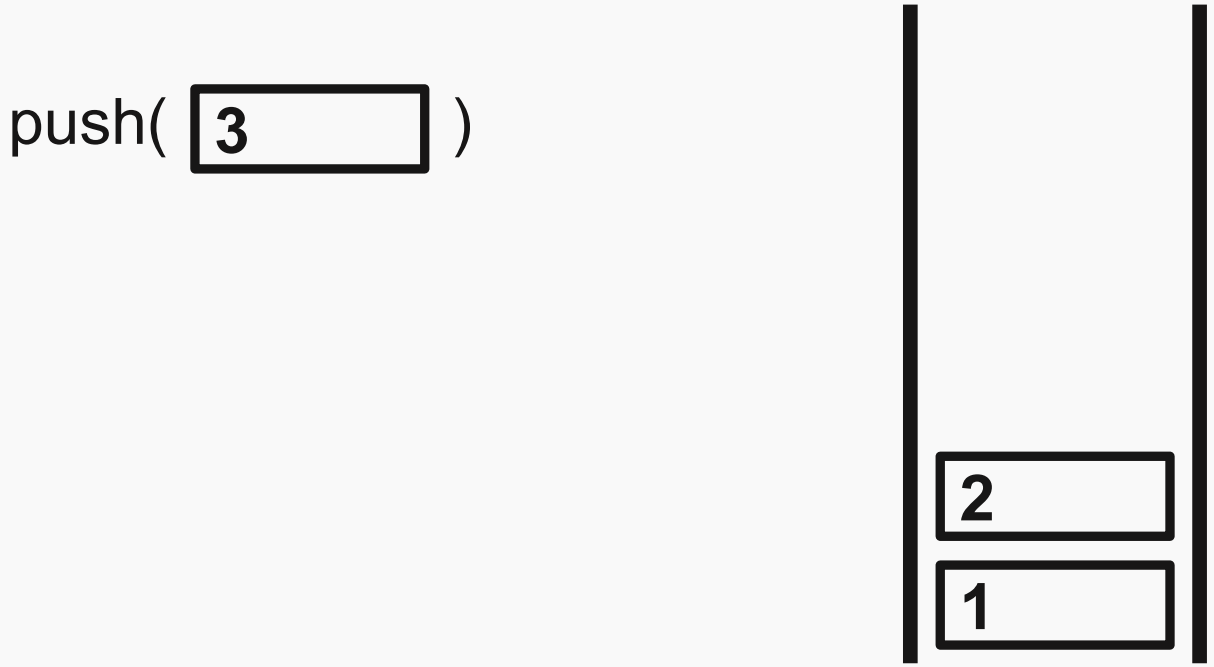
Queue - Coadă

FIFO – First In First Out



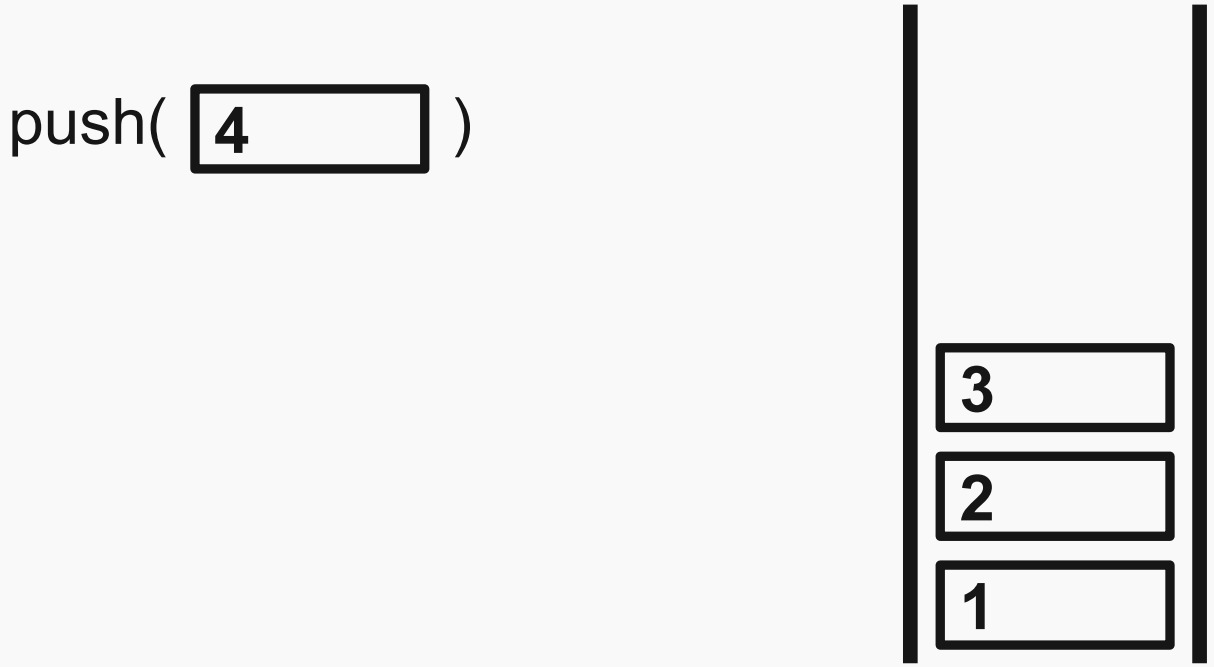
Queue - Coadă

FIFO – First In First Out



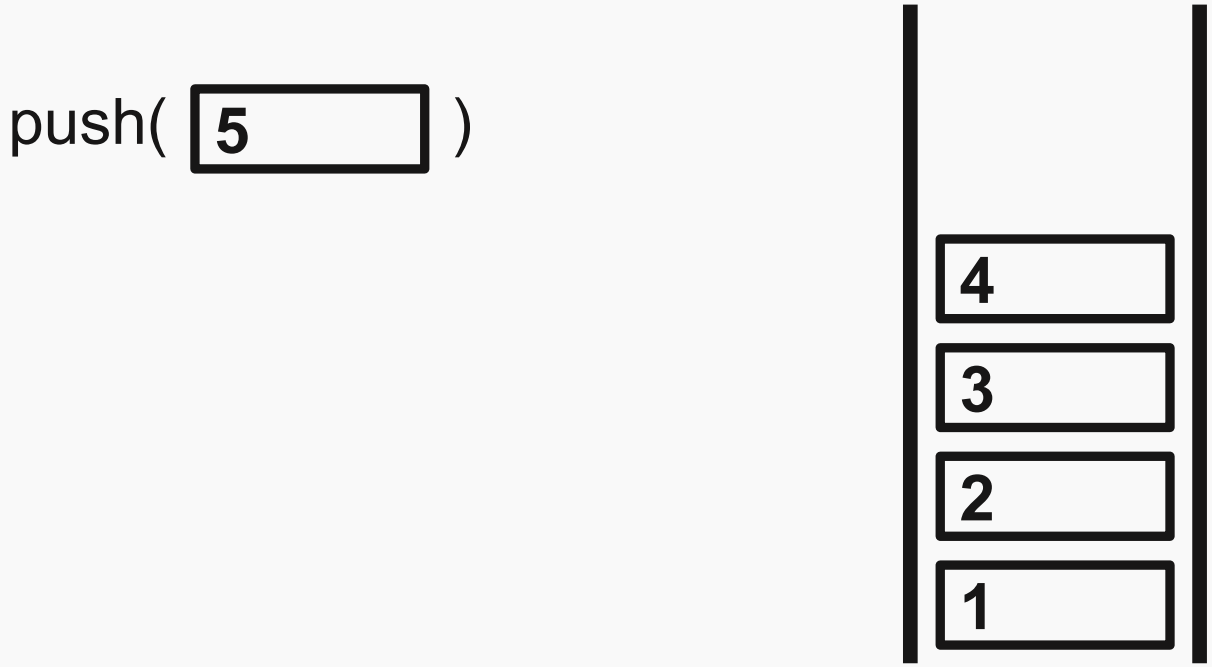
Queue - Coadă

FIFO – First In First Out



Queue - Coadă

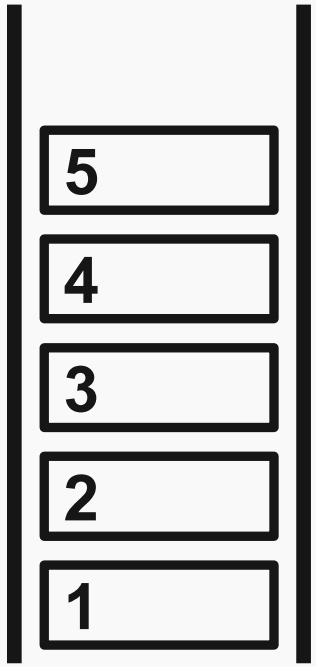
FIFO – First In First Out



Queue - Coadă

FIFO – First In First Out

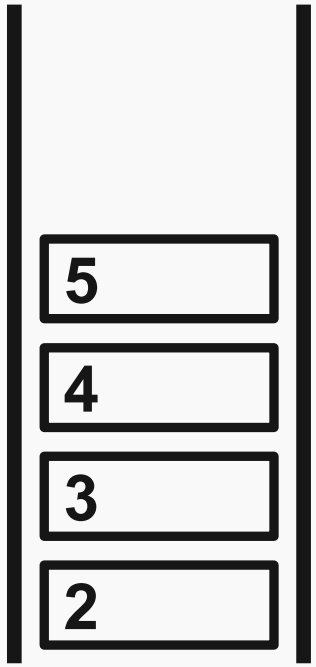
pop()



Queue - Coadă

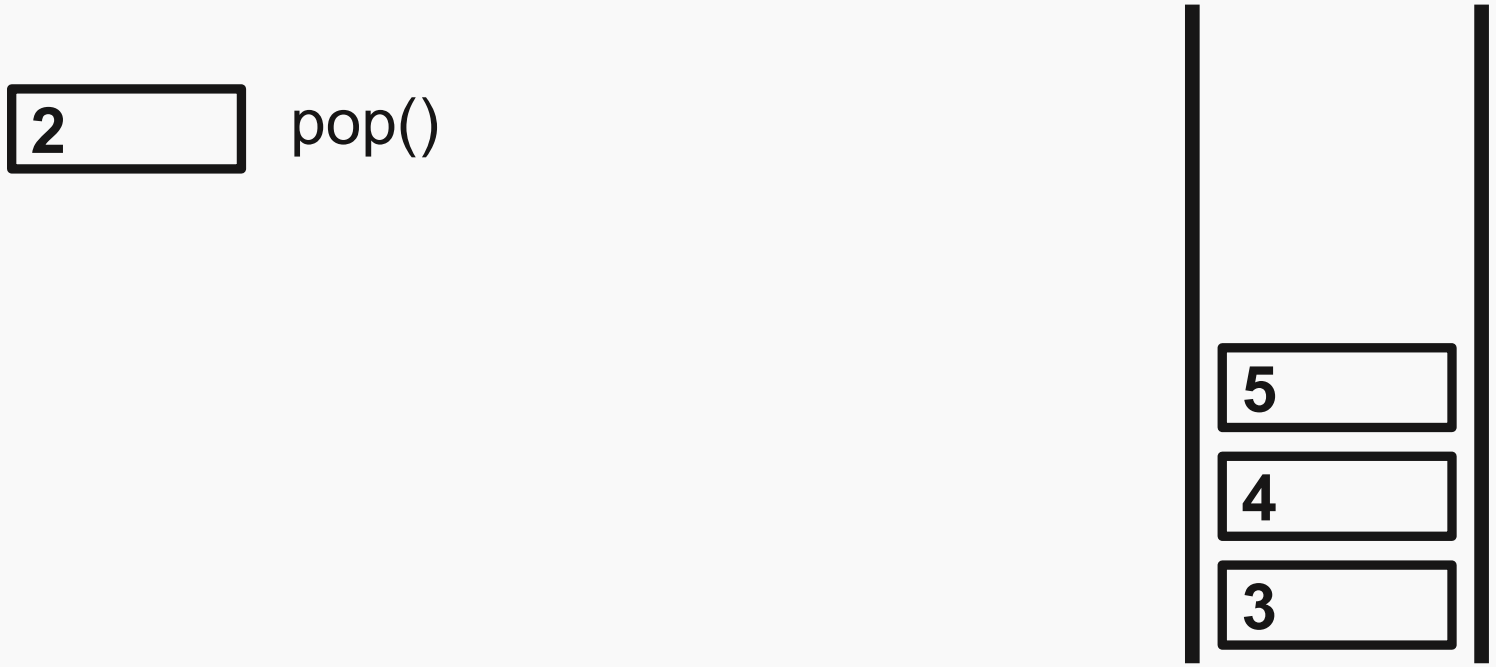
FIFO – First In First Out

pop()



Queue - Coadă

FIFO – First In First Out

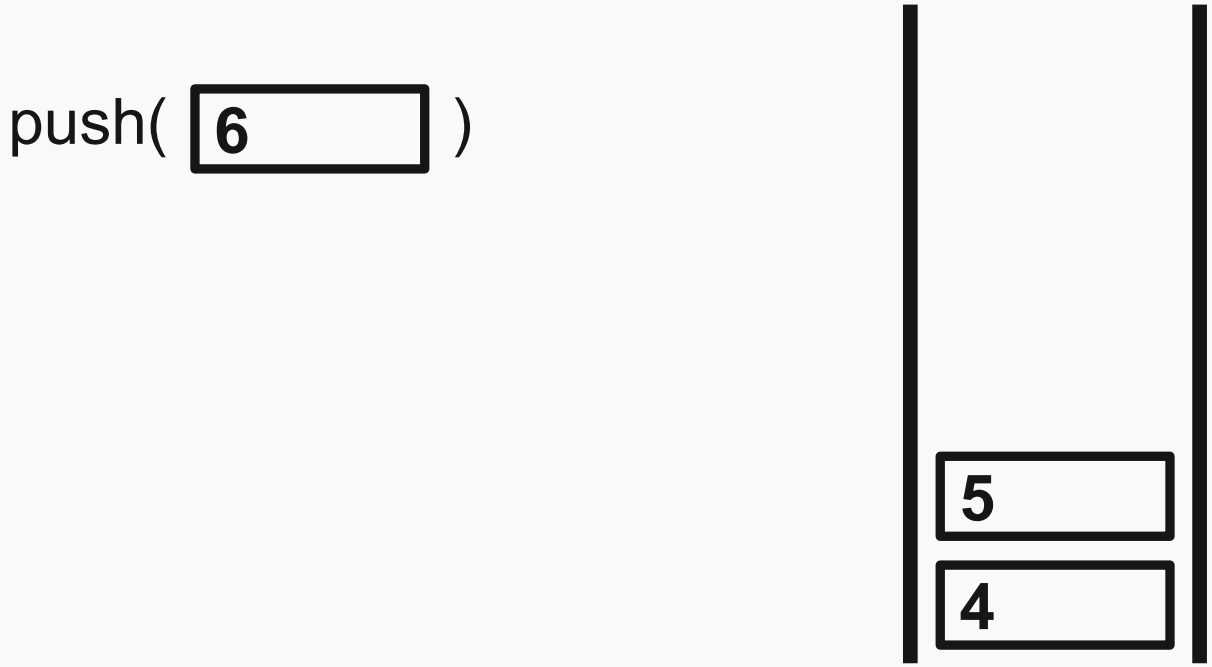


Queue - Coadă

FIFO – First In First Out

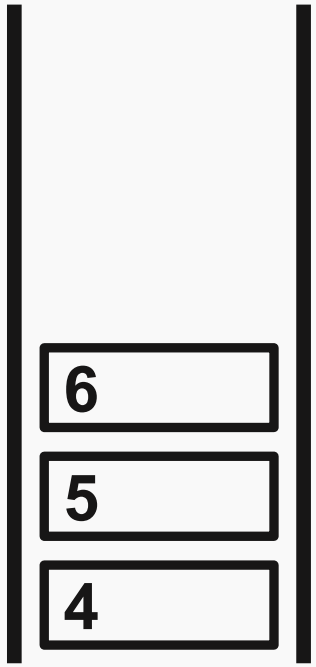
Queue - Coadă

FIFO – First In First Out



Queue - Coadă

FIFO – First In First Out

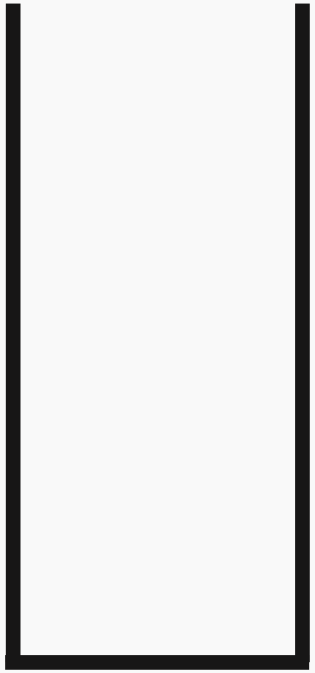


Coadă de priorități

Inversarea unui șir

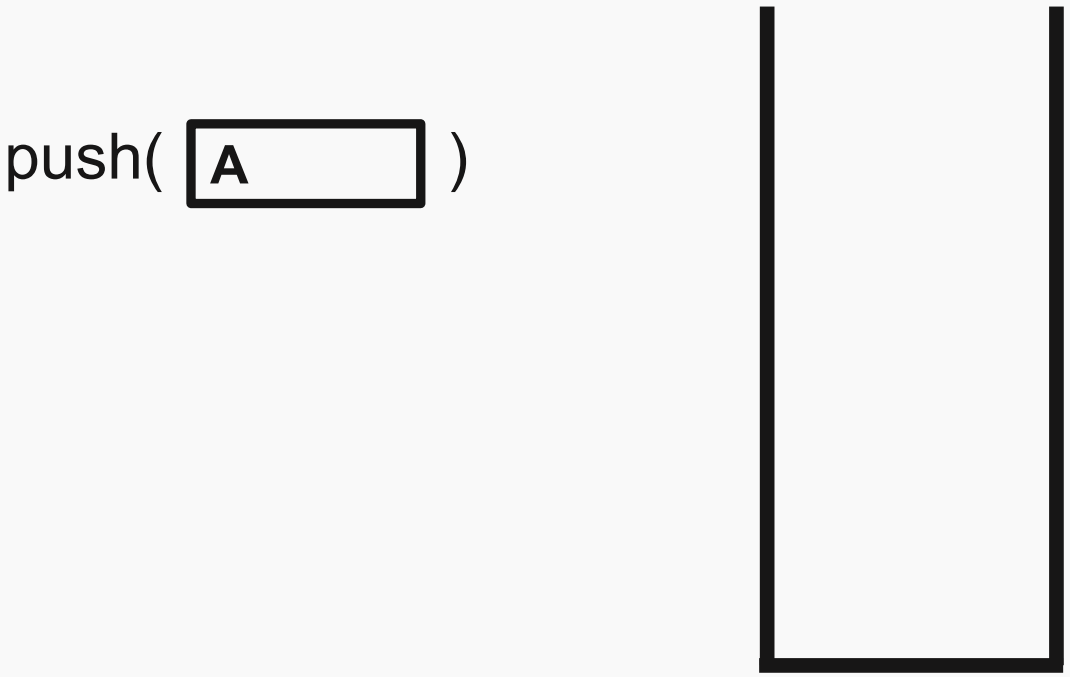
Inversarea unui șir

ABCDEF



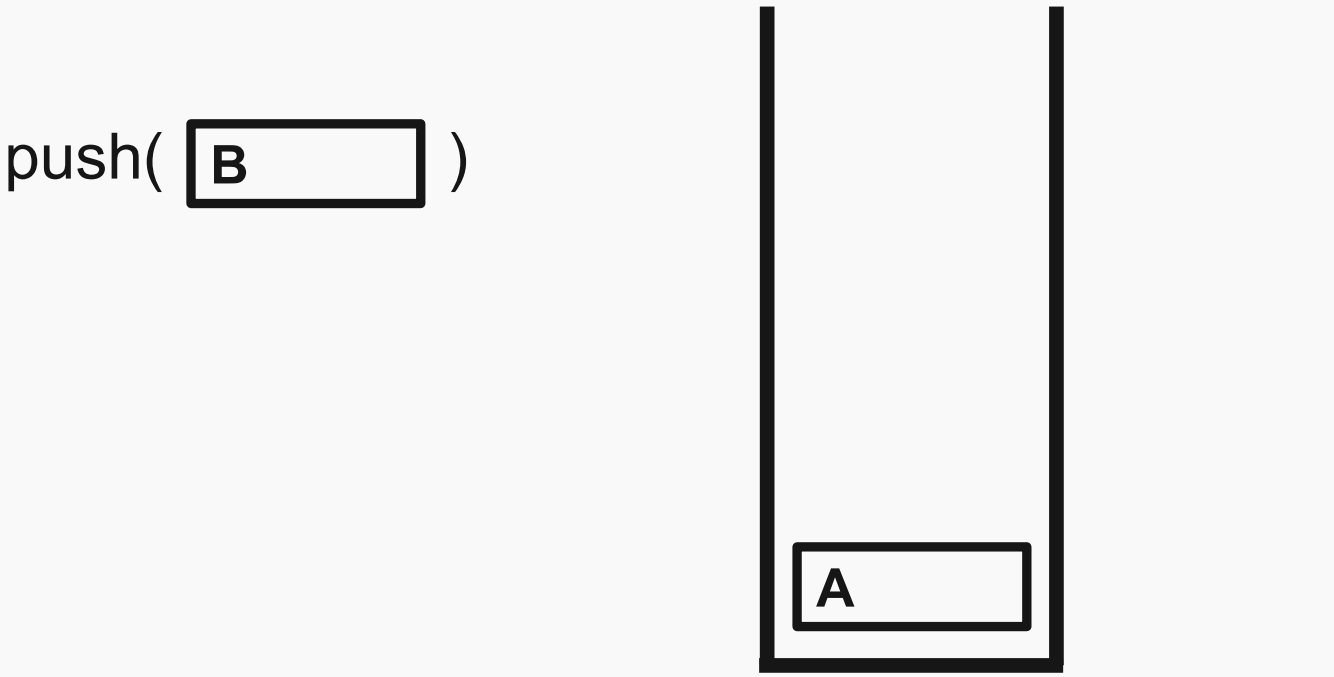
Inversarea unui șir

BCDEF



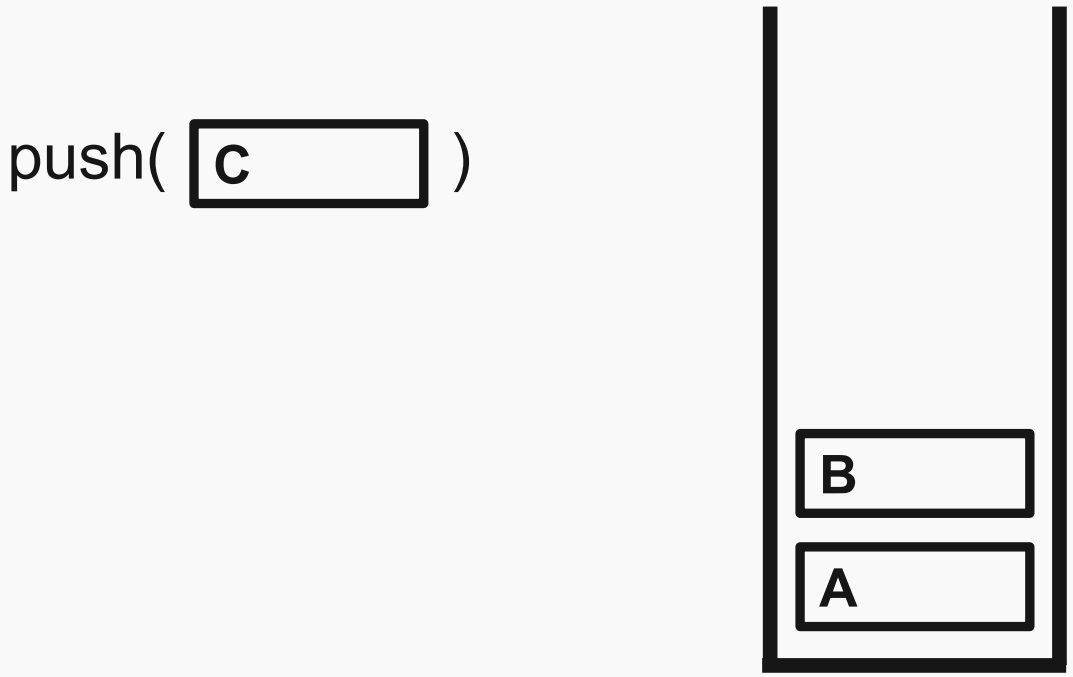
Inversarea unui șir

CDEF



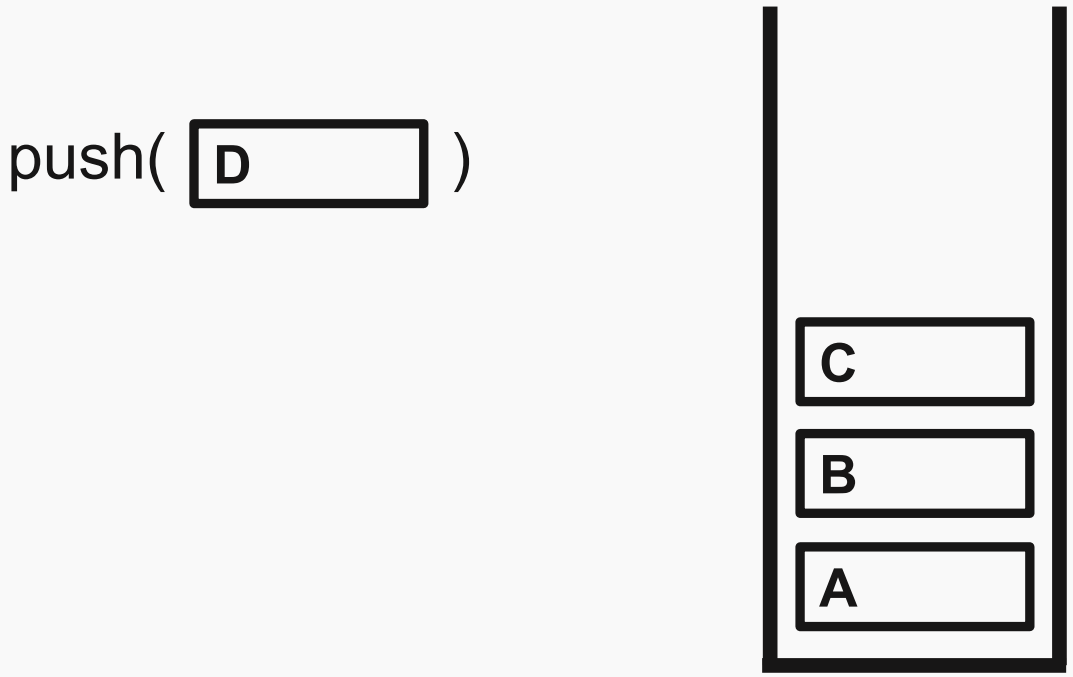
Inversarea unui șir

DEF



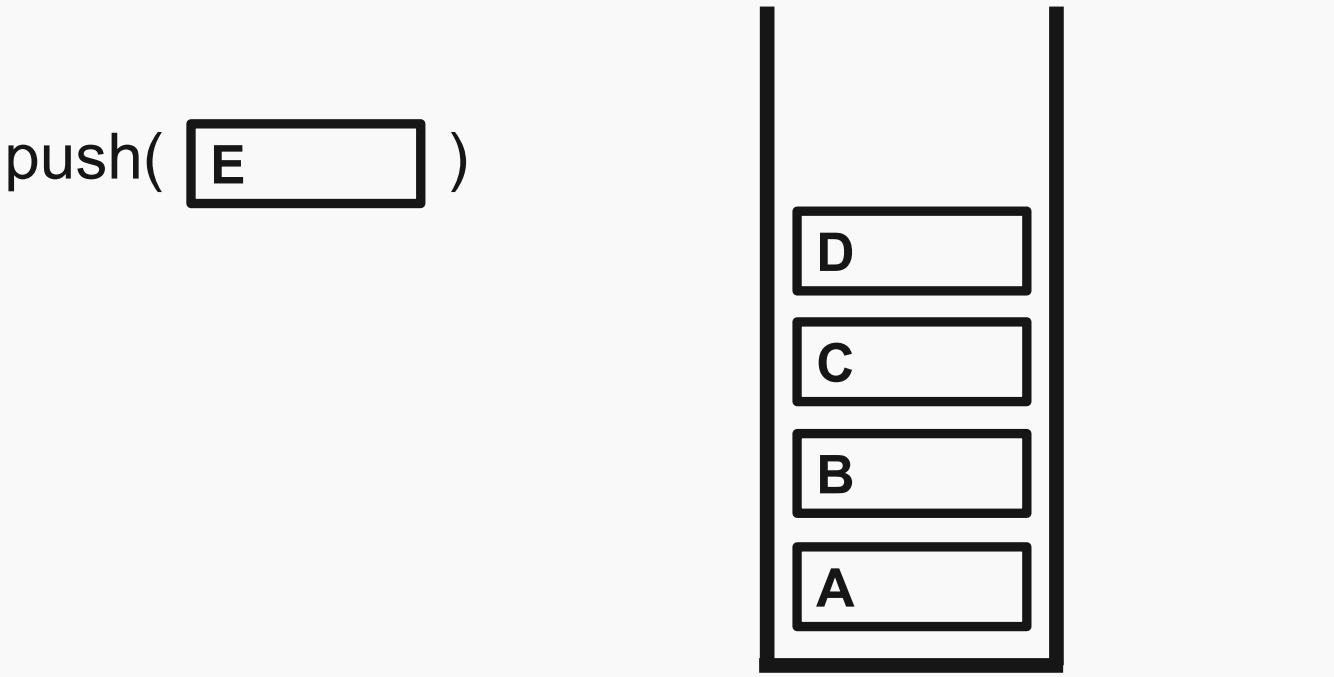
Inversarea unui șir

EF

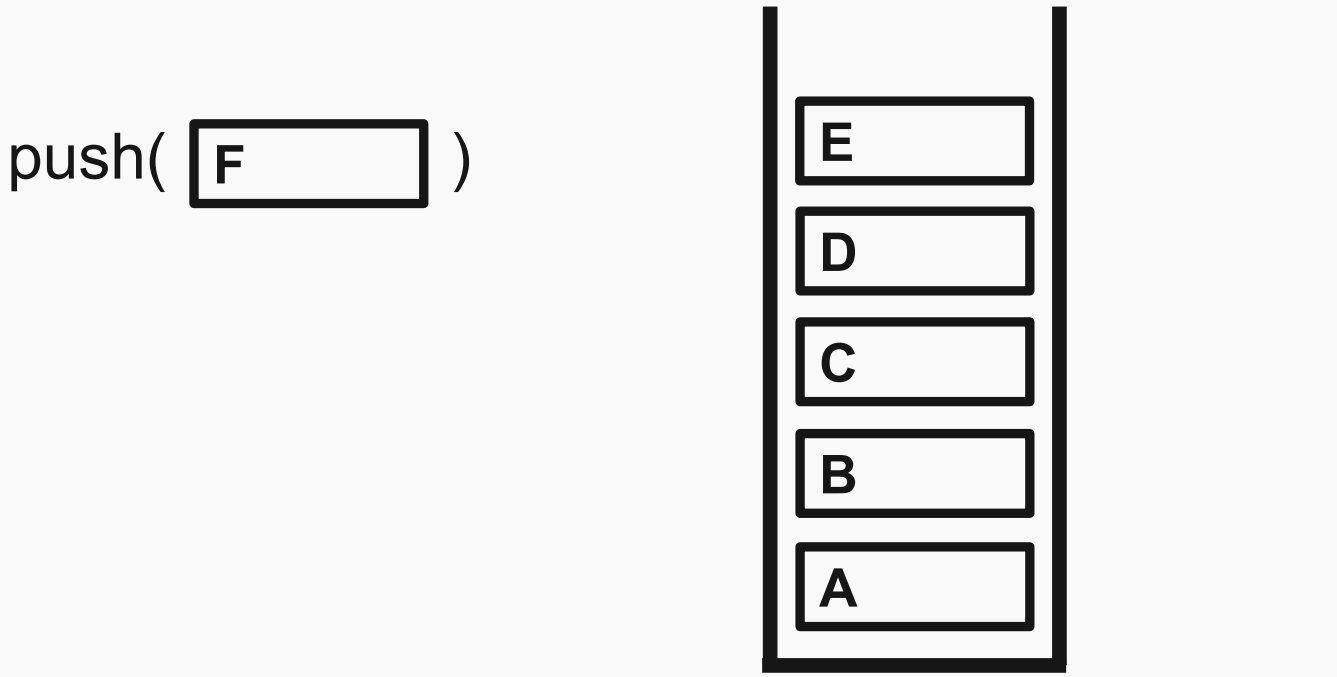


Inversarea unui șir

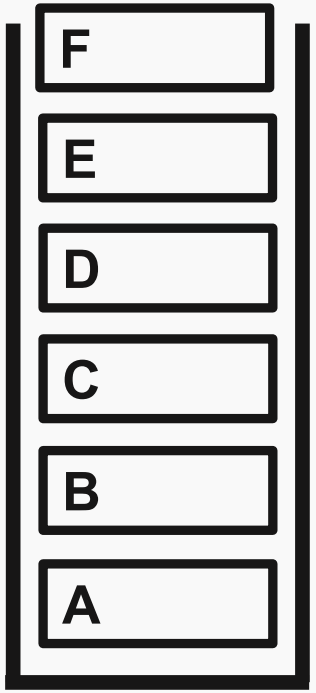
F



Inversarea unui șir

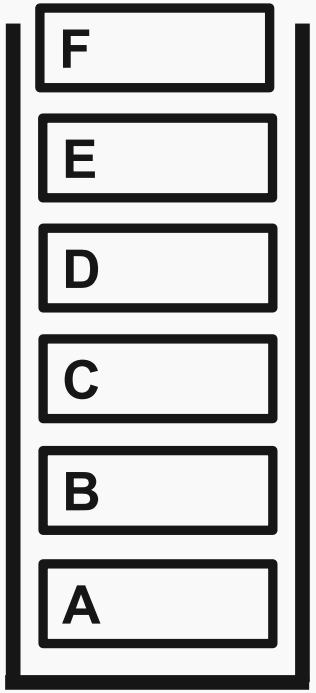


Inversarea unui șir

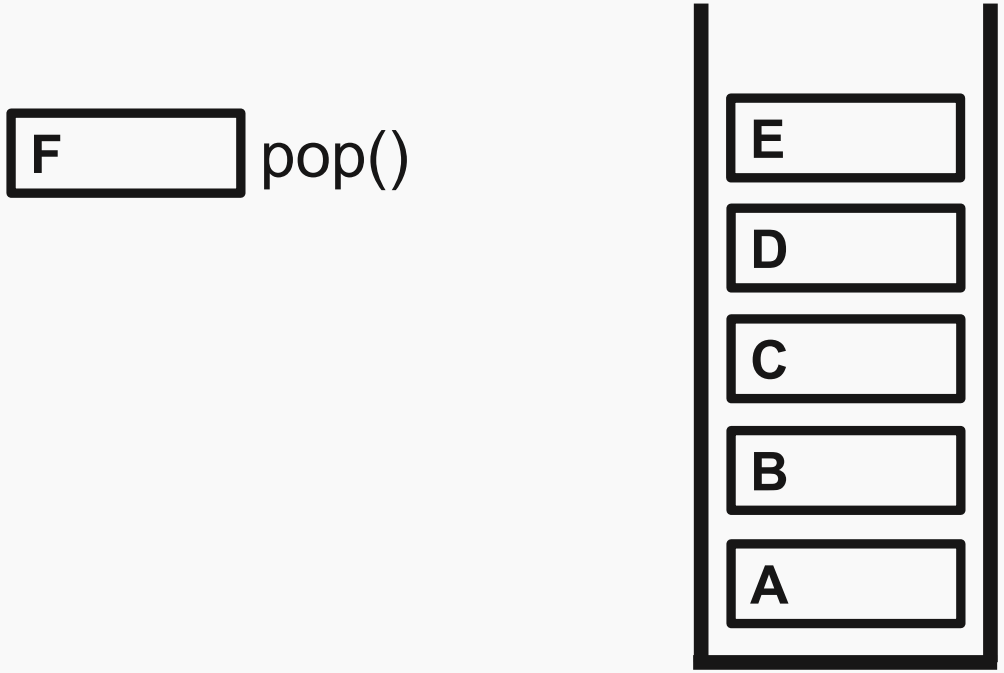


Inversarea unui șir

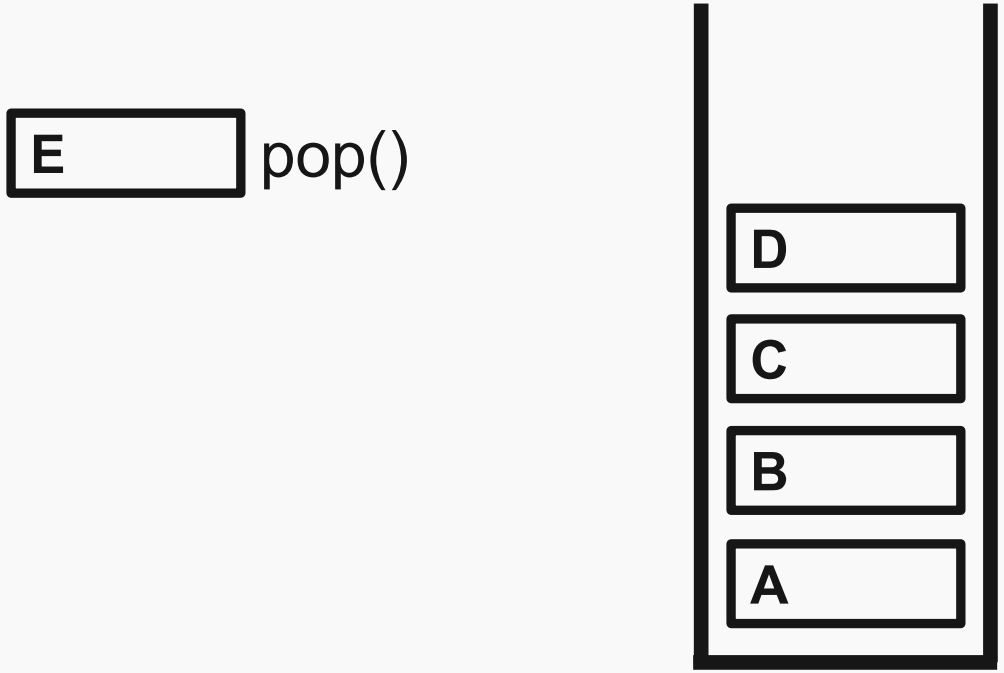
pop()



Inversarea unui șir

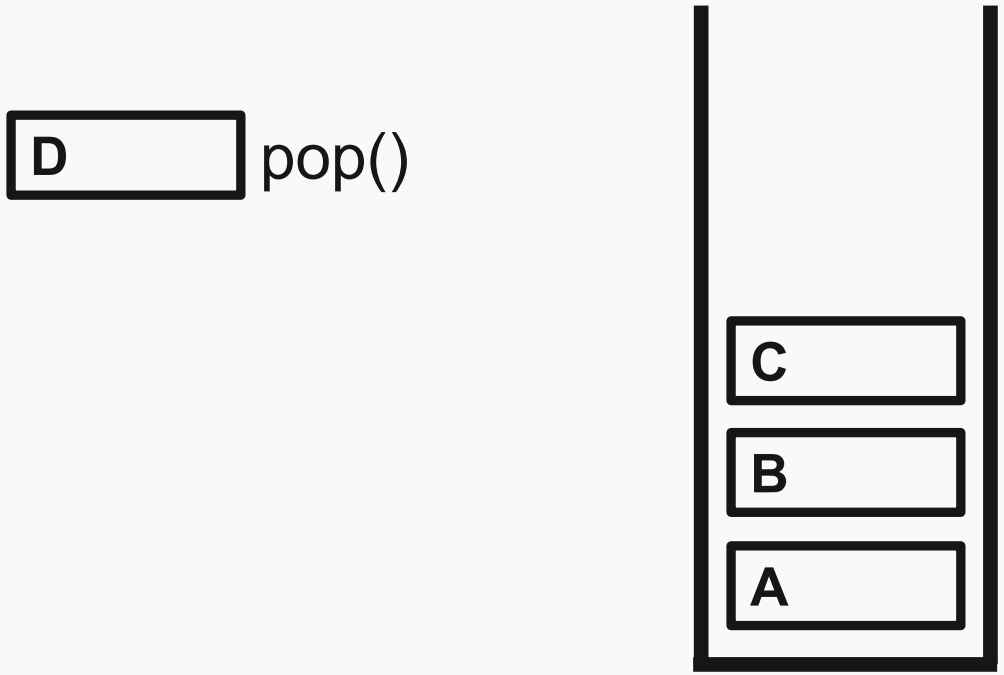


Inversarea unui șir



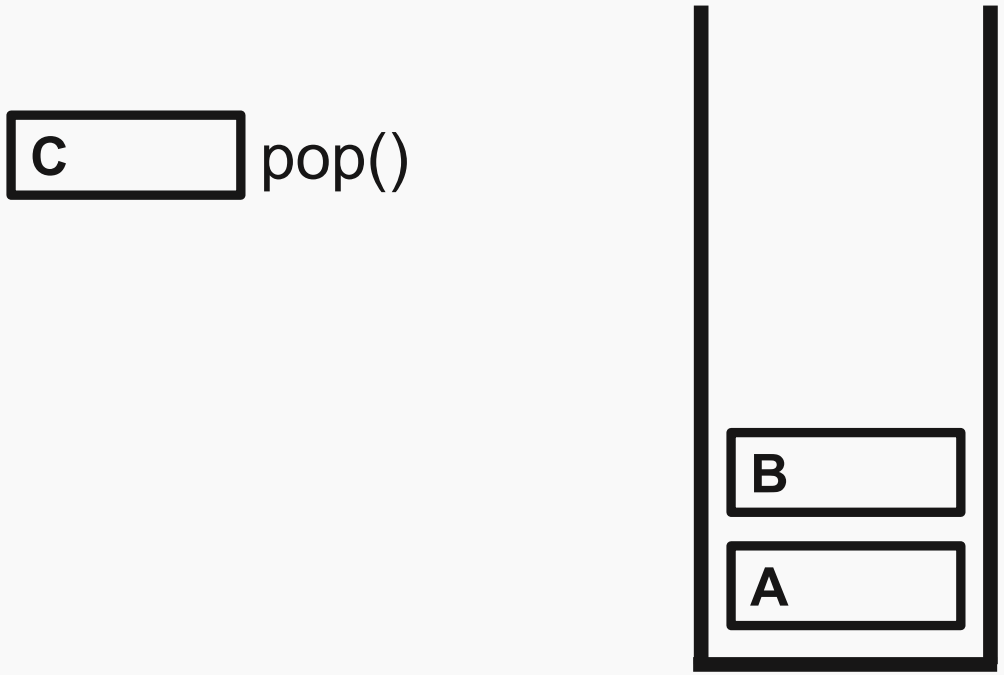
Afișare: F

Inversarea unui șir



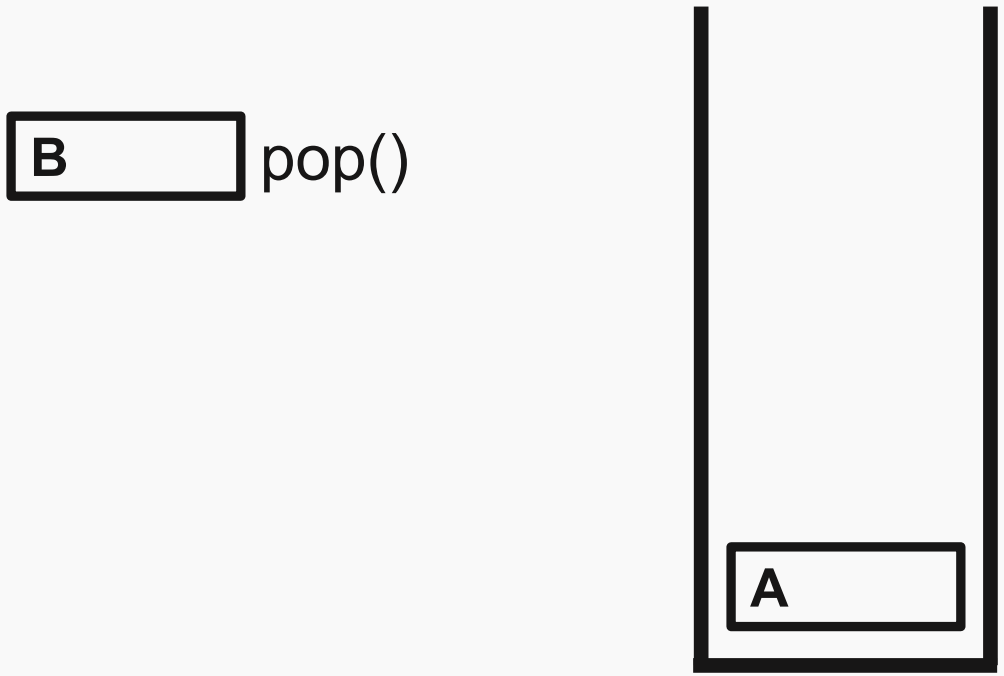
Afișare: FE

Inversarea unui șir



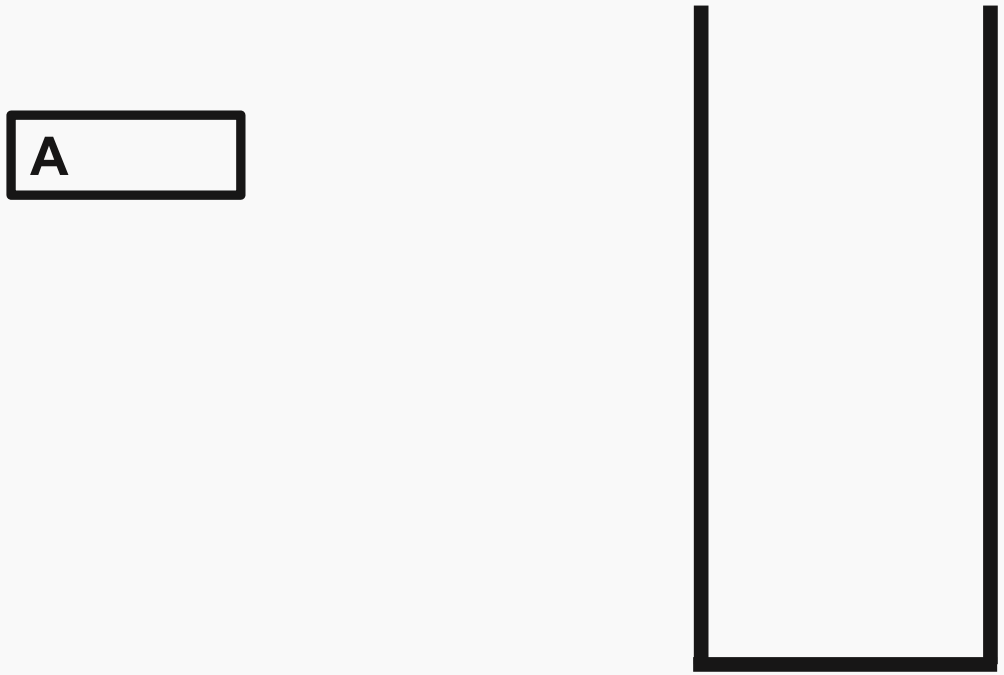
Afișare: FED

Inversarea unui șir



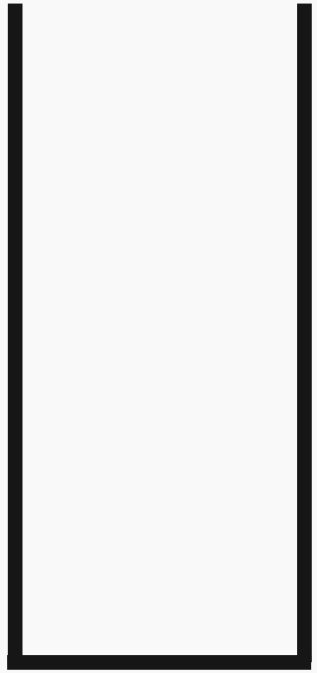
Afișare: FEDC

Inversarea unui șir



Afișare: FEDCB

Inversarea unui șir



Afișare: FEDCBA

Corectitudinea parantezelor unui expresii

1. [(1+2)-3*(2+1)]
2. {[()]}
3. [(()]
4. [)(]
5. [(1+)*2(3)]
6. {1+2+3+[2*3]*(1+2)}
7. {1+2+3+[2*3*(1+2])}
8. {1+2+3+[2*3]*(1+2)))

- Care expresii sunt corecte?

Corectitudinea parantezelor unui expresii

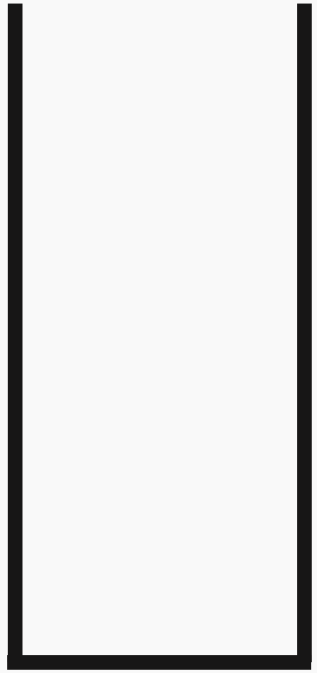
Parcurgem șirul de caractere

- Ignorăm orice caracter cu excepția [ ] ( ) { }
- Pentru caracterele ([{ adăugăm pe acestea pe stivă
- Pentru caracterele)] } scoatem capul stivei și verificăm dacă se potrivește
  - **Dacă nu avem ce scoate expresia este greșită**
  - **Dacă nu expresia este greșită**

Când terminăm șirul știm că expresia este corectă

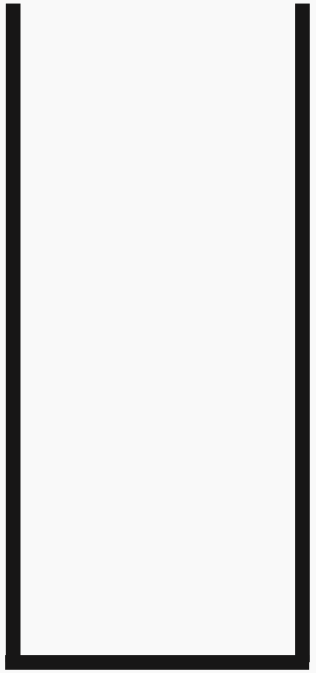
Corectitudinea parantezelor unui expresii

[(1+2)-3*(2+1)]



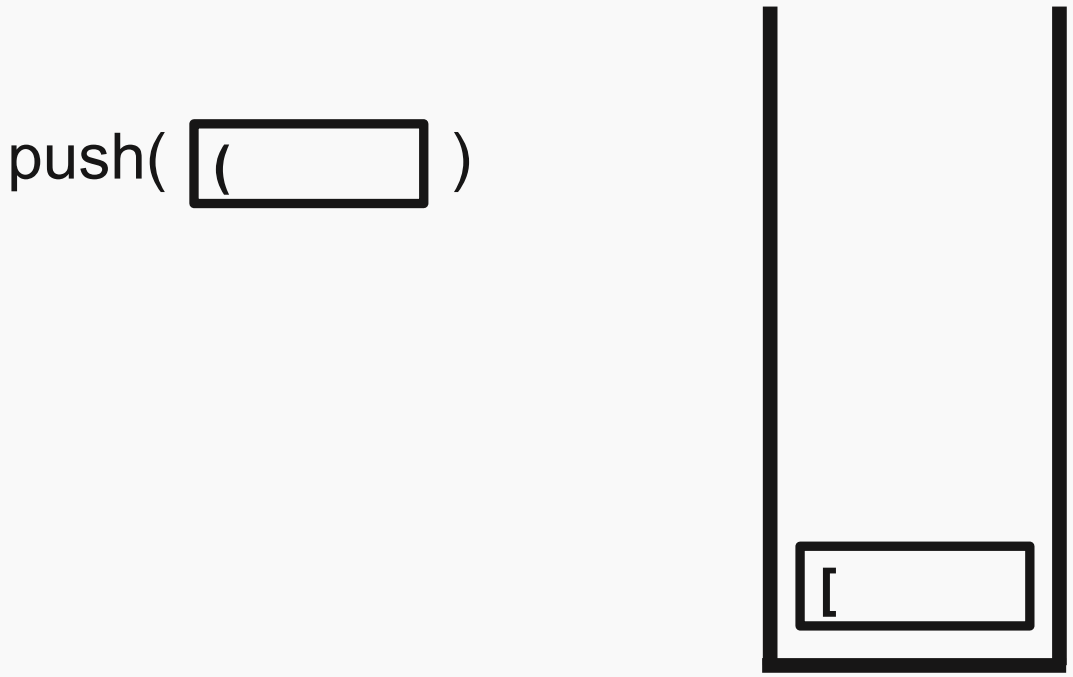
Corectitudinea parantezelor unui expresii

(1+2)-3*(2+1)]



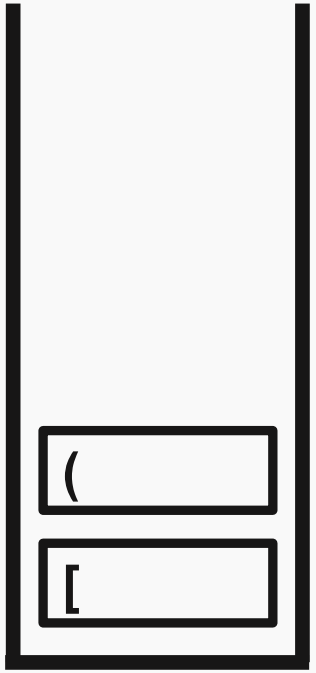
Corectitudinea parantezelor unui expresii

1+2)-3*(2+1)]



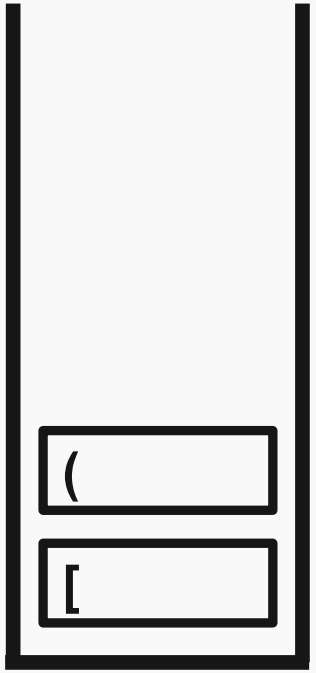
Corectitudinea parantezelor unui expresii

1+2)-3*(2+1)]



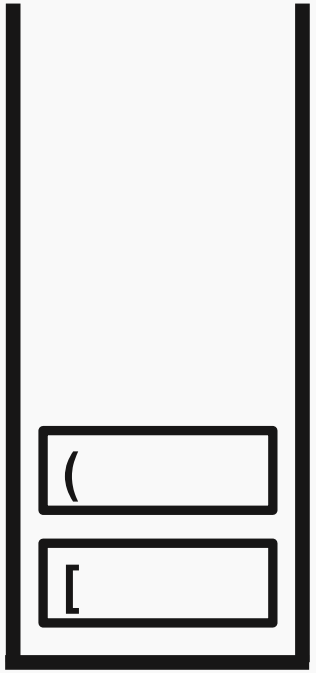
Corectitudinea parantezelor unui expresii

+2)-3*(2+1)]



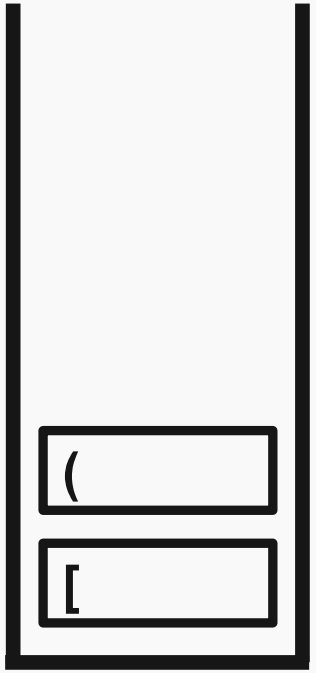
Corectitudinea parantezelor unui expresii

2)-3*(2+1)]



Corectitudinea parantezelor unui expresii

)-3*(2+1)]

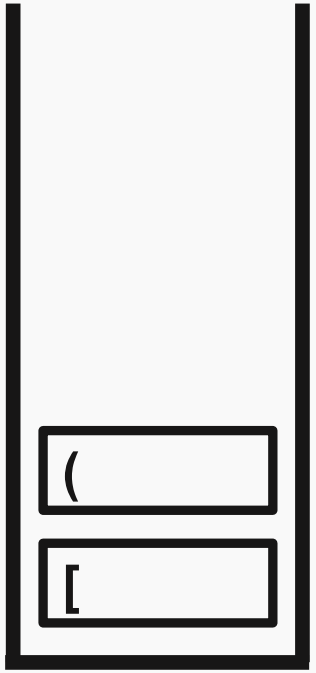


Corectitudinea parantezelor unui expresii

-3*(2+1)]

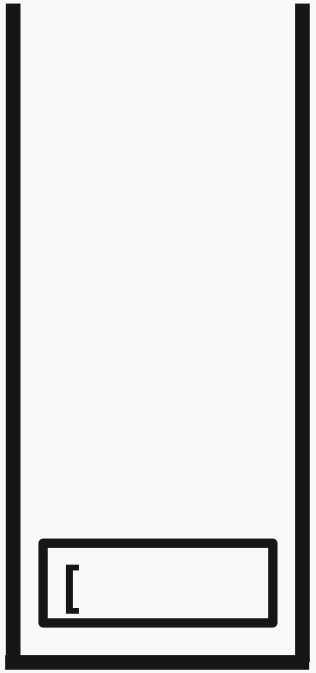
pop()

)



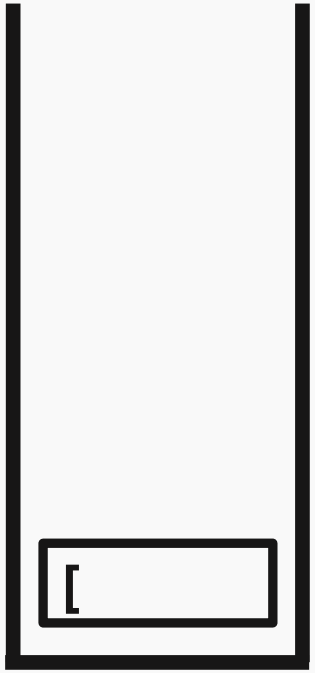
Corectitudinea parantezelor unui expresii

-3*(2+1)]



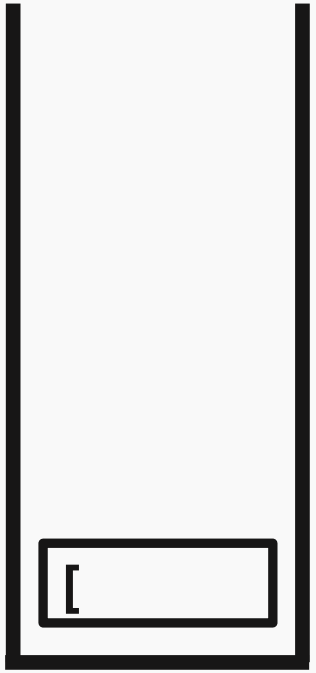
Corectitudinea parantezelor unui expresii

3*(2+1)]



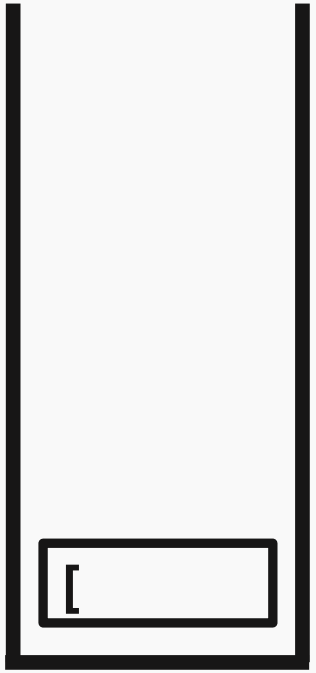
Corectitudinea parantezelor unui expresii

*(2+1)]



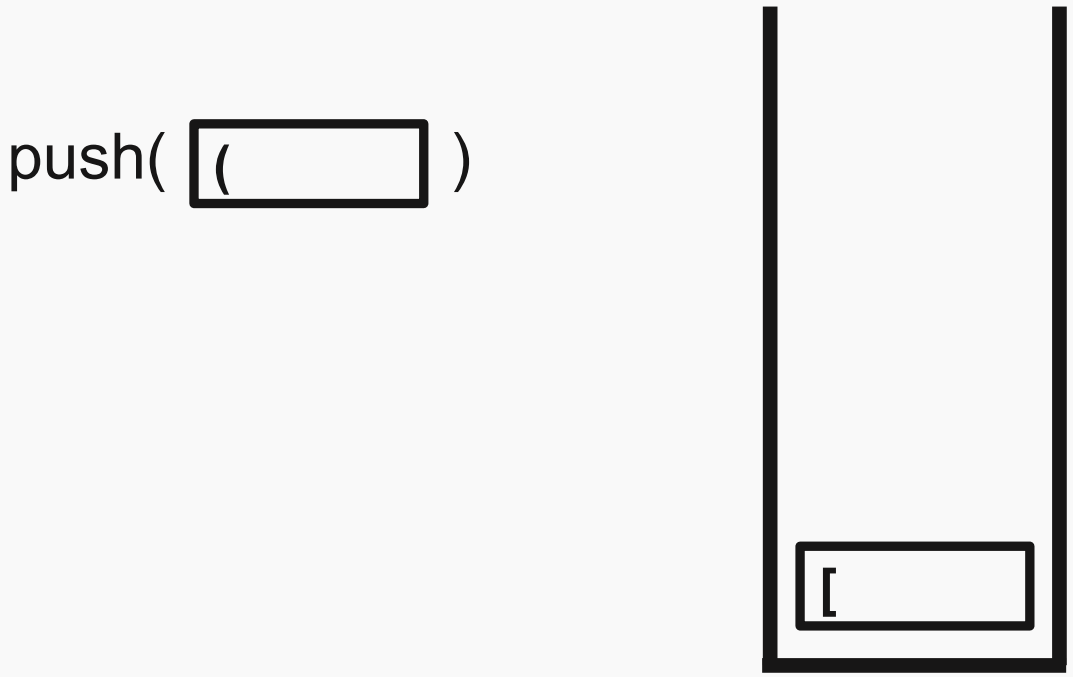
Corectitudinea parantezelor unui expresii

(2+1)]



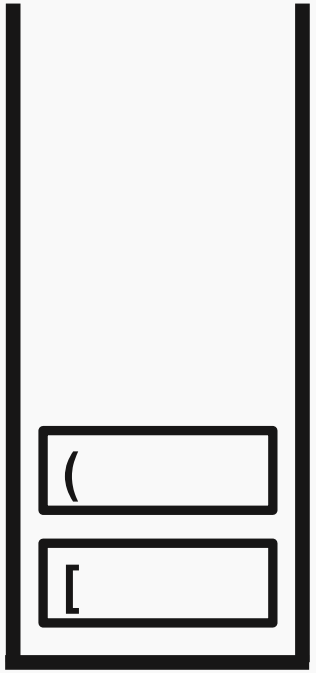
Corectitudinea parantezelor unui expresii

2+1)]



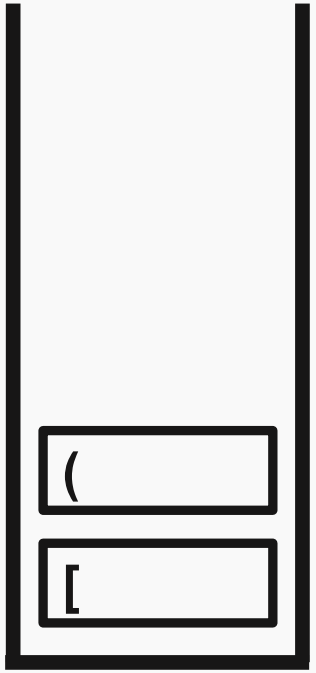
Corectitudinea parantezelor unui expresii

2+1)]



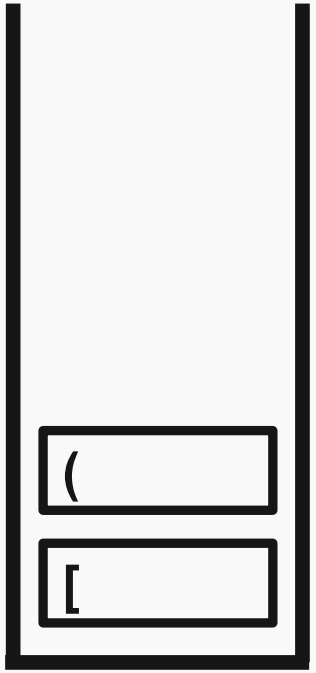
Corectitudinea parantezelor unui expresii

+1)]



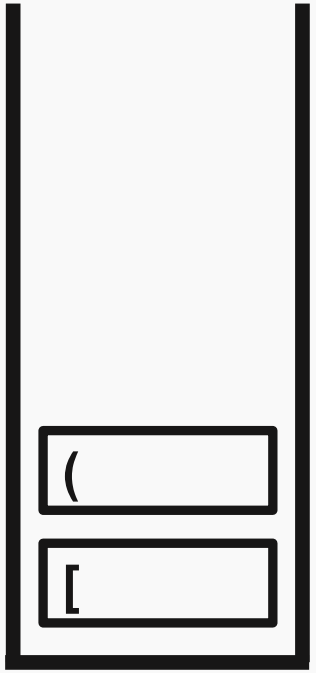
Corectitudinea parantezelor unui expresii

1)]



Corectitudinea parantezelor unui expresii

)]

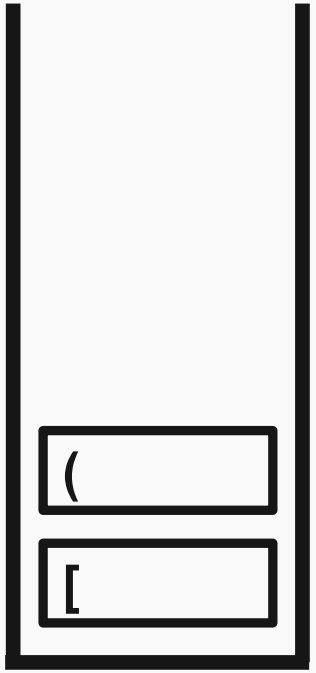


Corectitudinea parantezelor unui expresii

]

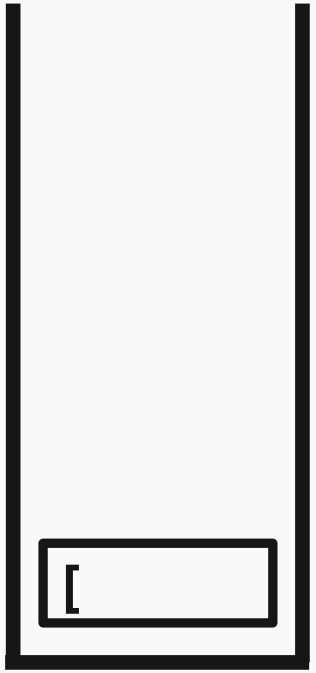
pop()

)



Corectitudinea parantezelor unui expresii

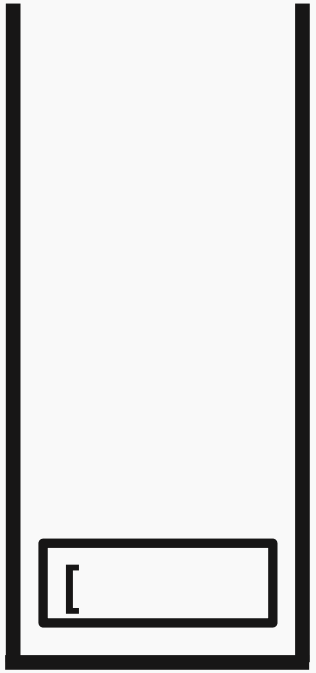
]



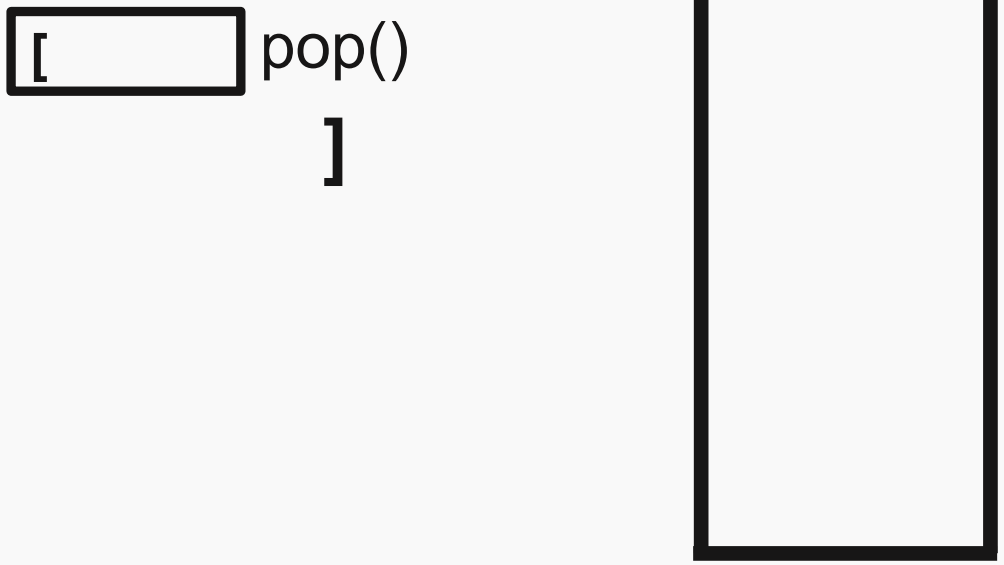
Corectitudinea parantezelor unui expresii

pop()

]

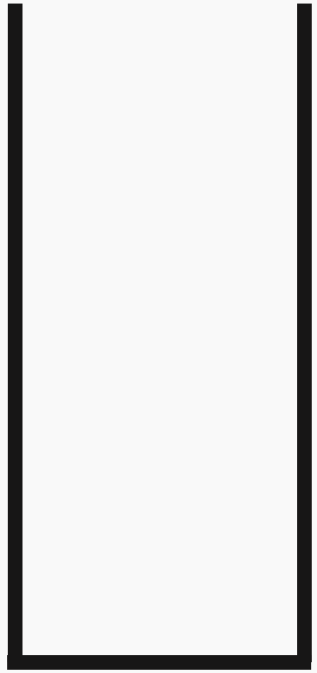


Corectitudinea parantezelor unui expresii



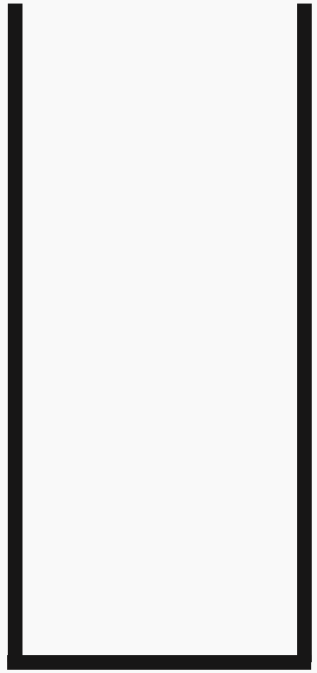
Corectitudinea parantezelor unui expresii

[(1+2)-3*(2+1)] **Corectă**



Corectitudinea parantezelor unui expresii

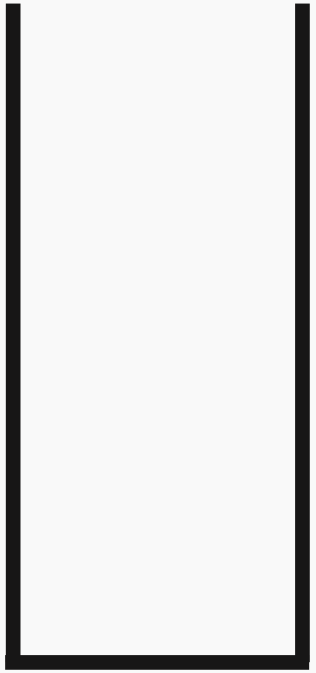
[)(]



Corectitudinea parantezelor unui expresii

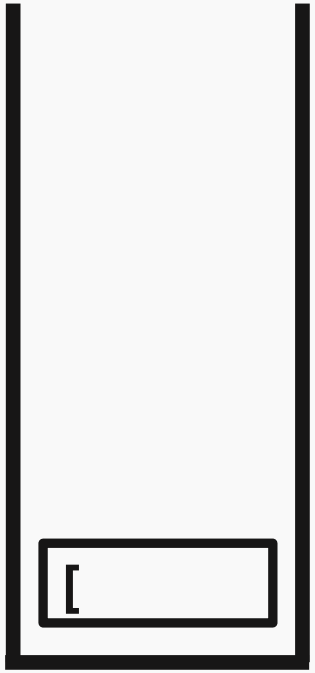
)(]

push([ )



Corectitudinea parantezelor unui expresii

)(]

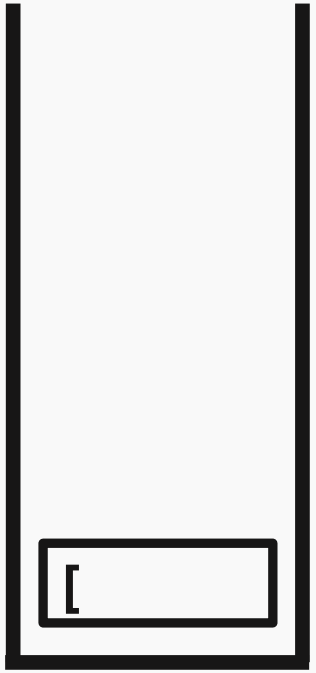


Corectitudinea parantezelor unui expresii

(]

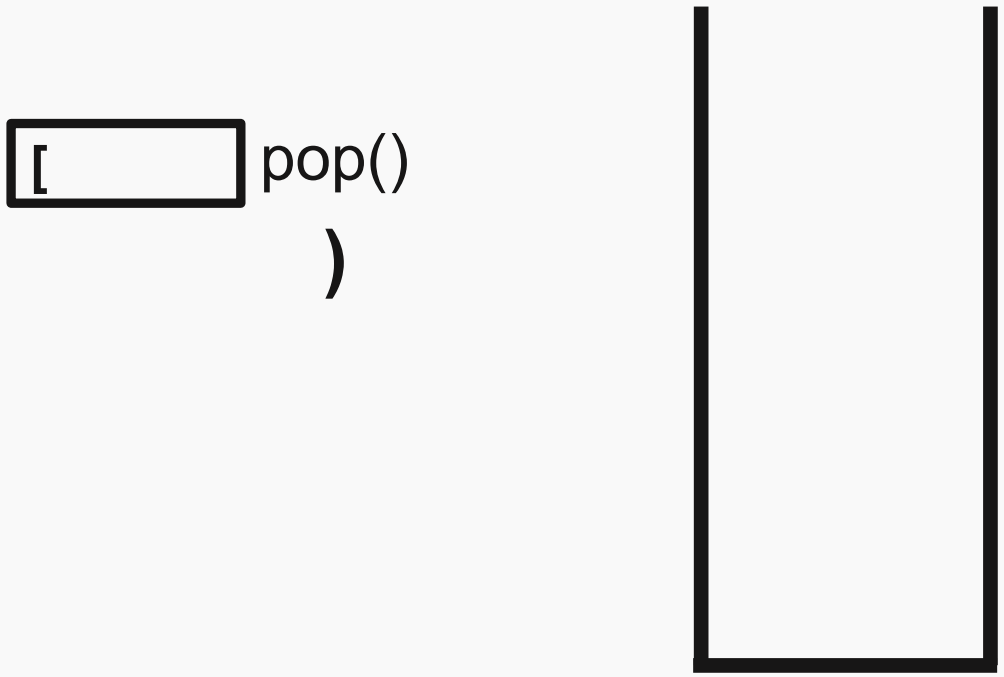
pop()

)



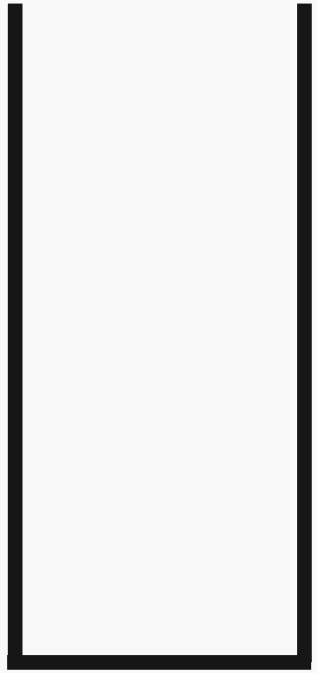
Corectitudinea parantezelor unui expresii

(] Incorect



Corectitudinea parantezelor unui expresii

[)(] Incorect



Stiva programului

http://pythontutor.com/c.html#mode=display

Ce problemă observați?

```c
#include<stdlib.h>

int fun1() {
    return 1;
}

void fun2() {
    int d = 0;
    fun1();
    return;
}
```

```c
int main() {
    int a = 5;
    int v[4] = { 0,1,2,3 };
    int* vv = (int*)malloc(5);
    vv[3] = 5;
    fun2();
    a = fun1();
    return 0;
}
```

Stiva programului

http://pythontutor.com/c.html#mode=display

```c
#include<stdlib.h>

int fun1() {
    return 1;
}

void fun2() {
    int d = 0;
    fun1();
    return;
}
```

```c
int main() {
    int a = 5;
    int v[4] = { 0,1,2,3 };
    int* vv = (int*)malloc(5 * sizeof(int));
    vv[3] = 5;
    fun2();
    a = fun1();
    return 0;
}
```

Stiva programului

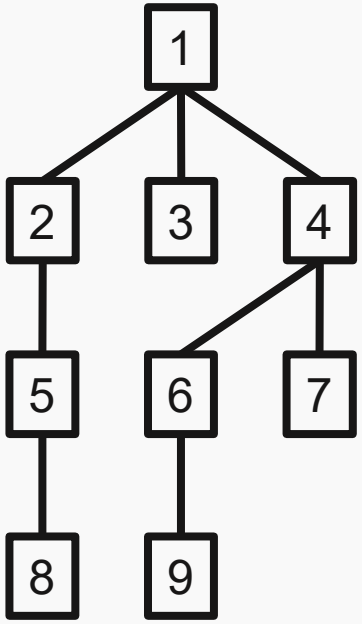
Ce valori vor fi afișate de fun2?

```c
#include<stdio.h>
#define N 5

void fun1() {
    int v[N];
    int i;
    for (i = 0; i < N; i++) {
        v[i] = i;
        printf("fun1: %p %i\n", v + i, v[i]);
    }
}

void fun2() {
    int z[N];
    int i;
    for (i = 0; i < N; i++) {
        printf("fun2: %p %i\n", z + i, z[i]);
        z[i] = 0;
    }
}
```

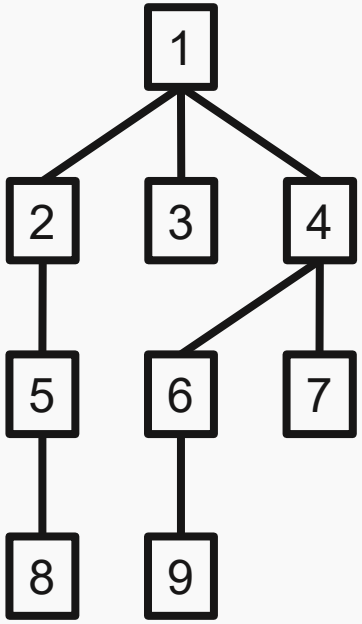
```c
int main() {
    fun1();
    fun2();
    return 0;
}
```

Implementare reală linked lists

https://gcc.gnu.org/onlinedocs/gcc-4.6.2/libstdc++/api/a00922_source.html

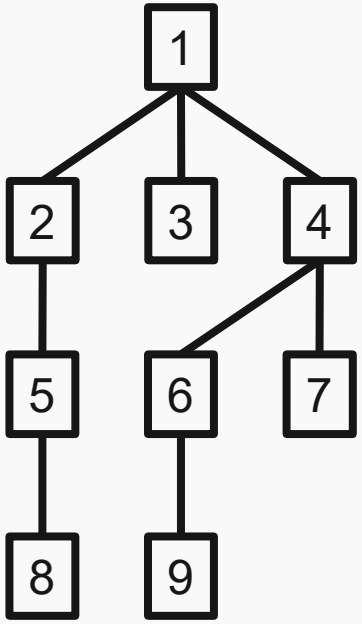


Care este rădăcina arborelui?

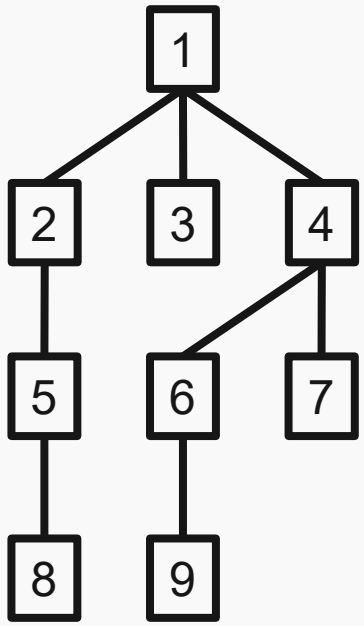


Care sunt frunzele arborelui?

Afișarea unui arbore fără recursivitate

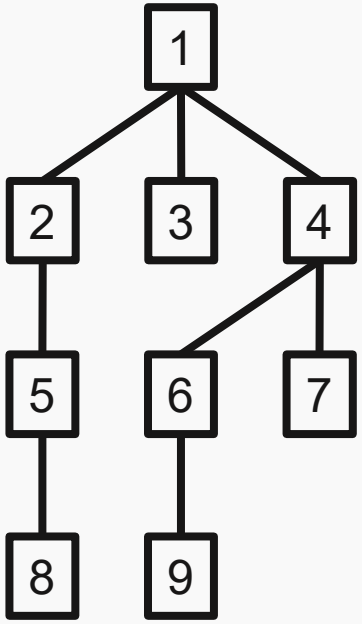


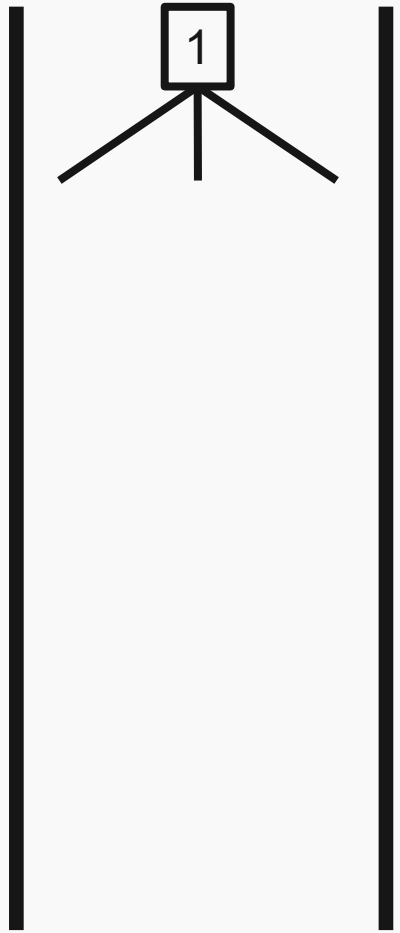
Afișarea unui arbore folosind o coadă



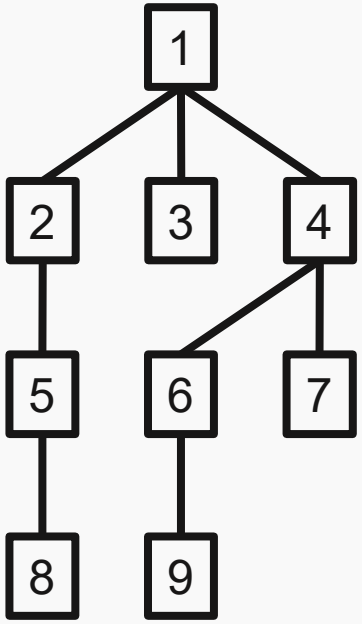
- Adăugăm rădăcina în coadă
- Până ce coada este goală
  - Scoatem un nod din coadă
  - Adăugăm toți copiii în coadă

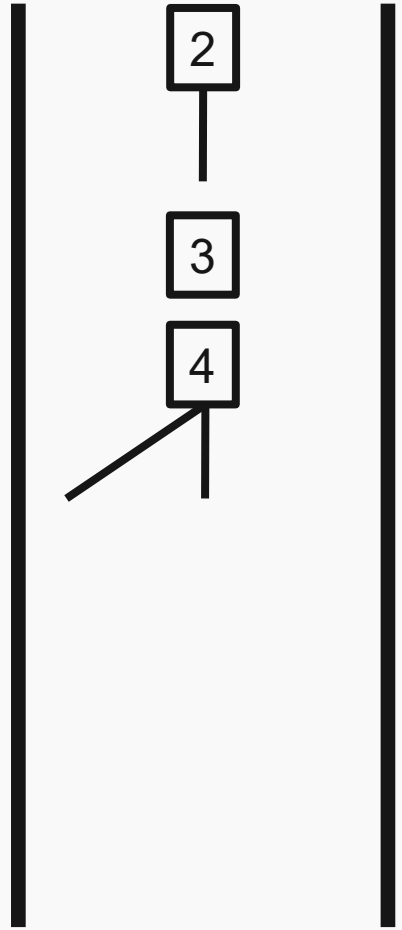
Afișarea unui arbore folosind o coadă





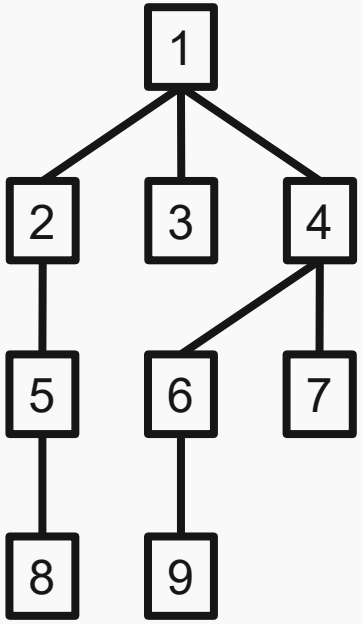
Afișarea unui arbore folosind o coadă

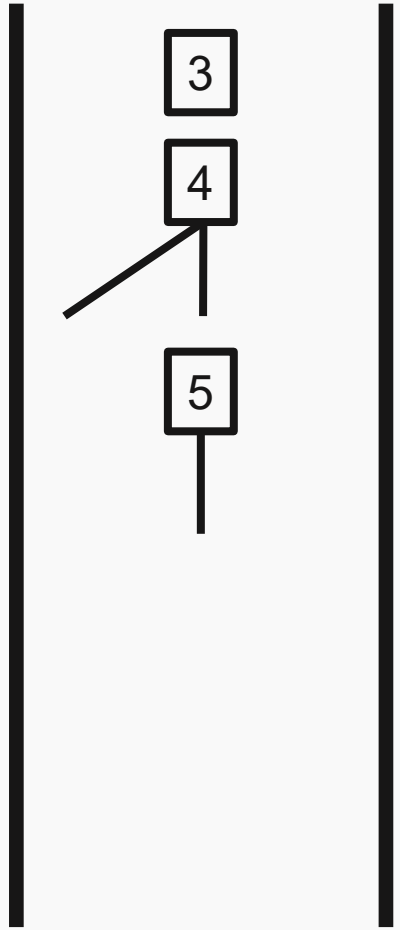




Afișare: 1

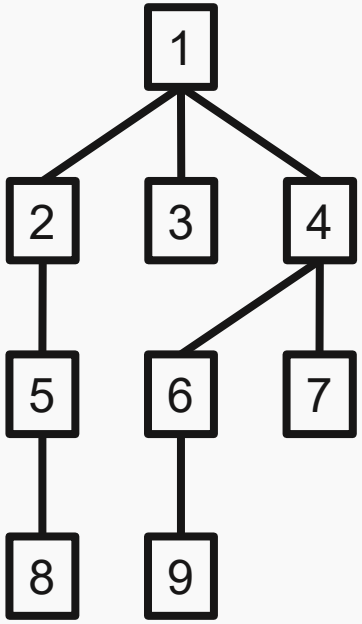
Afișarea unui arbore folosind o coadă

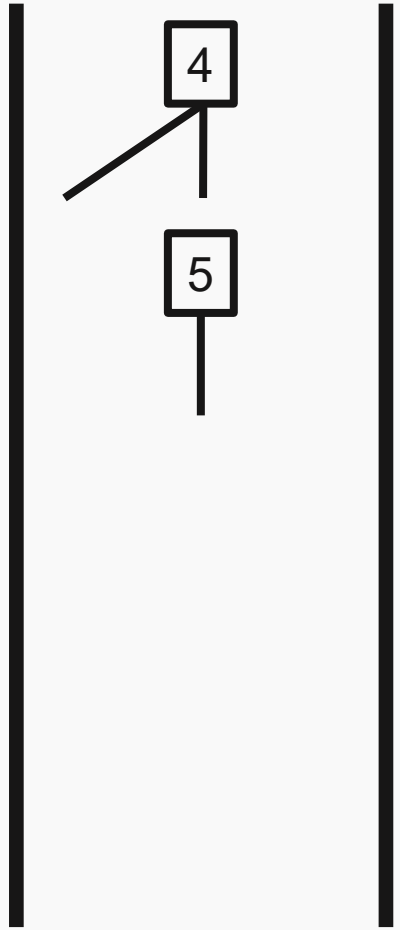




Afișare: 1 2

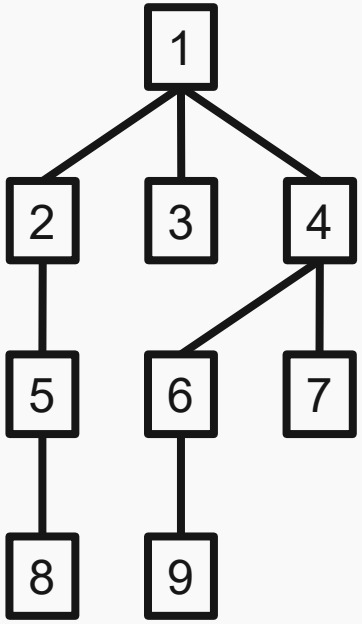
Afișarea unui arbore folosind o coadă

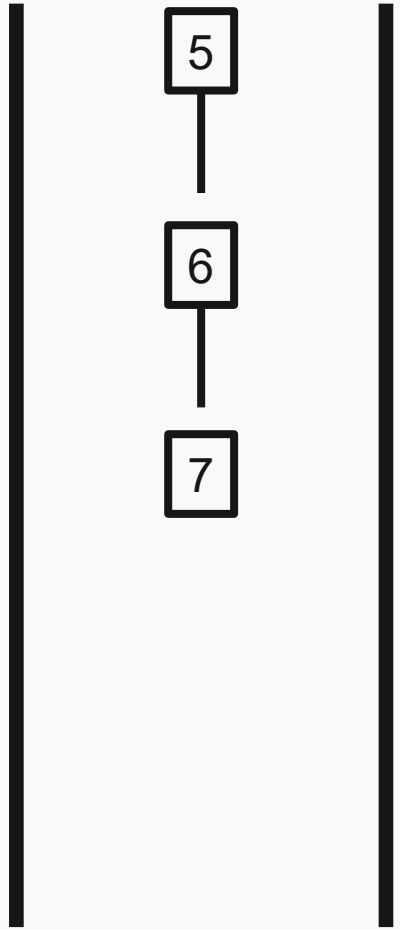




Afișare: 1 2 3

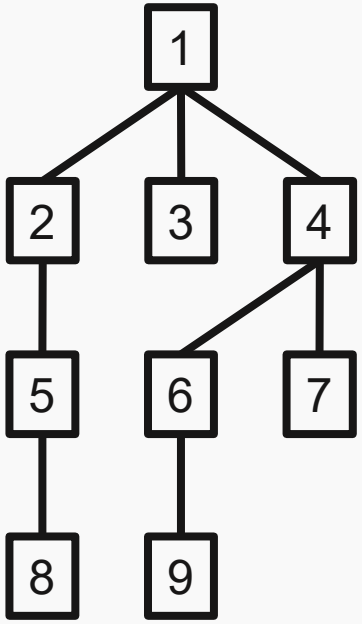
Afișarea unui arbore folosind o coadă

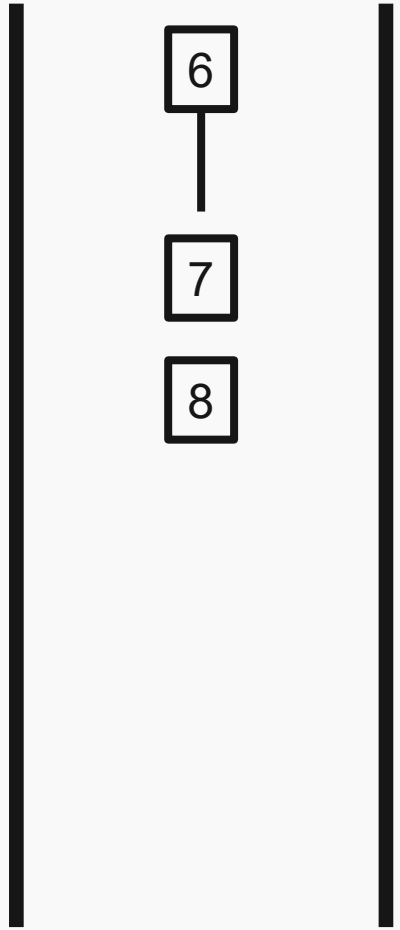




Afișare: 1 2 3 4

Afișarea unui arbore folosind o coadă

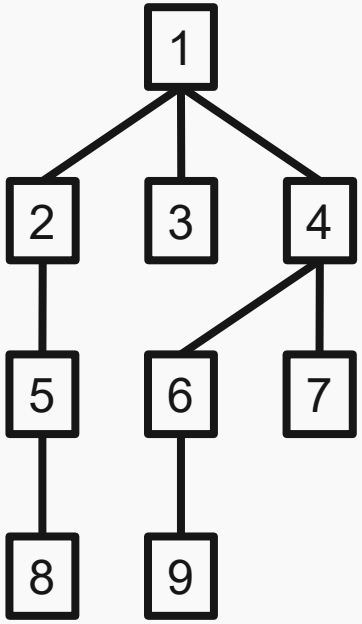


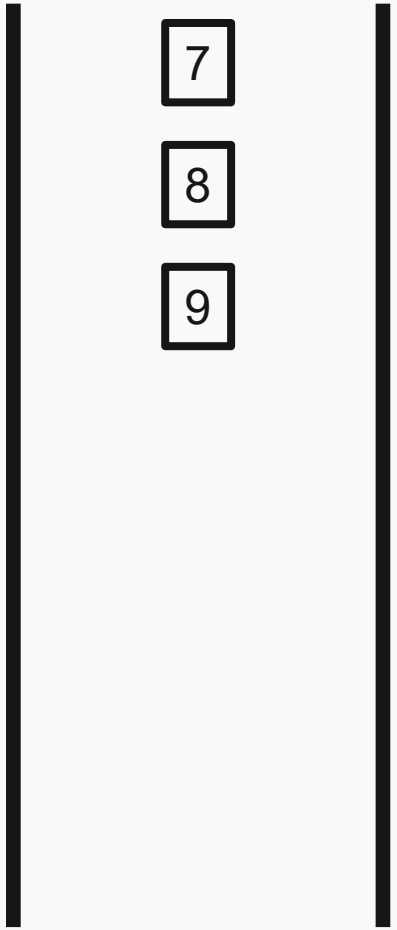


Afișare:

1	2	3	4	5

Afișarea unui arbore folosind o coadă

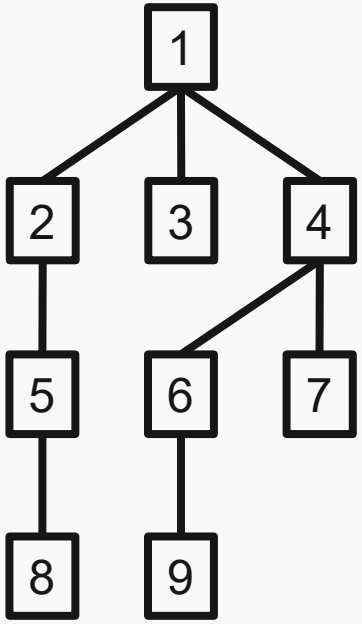


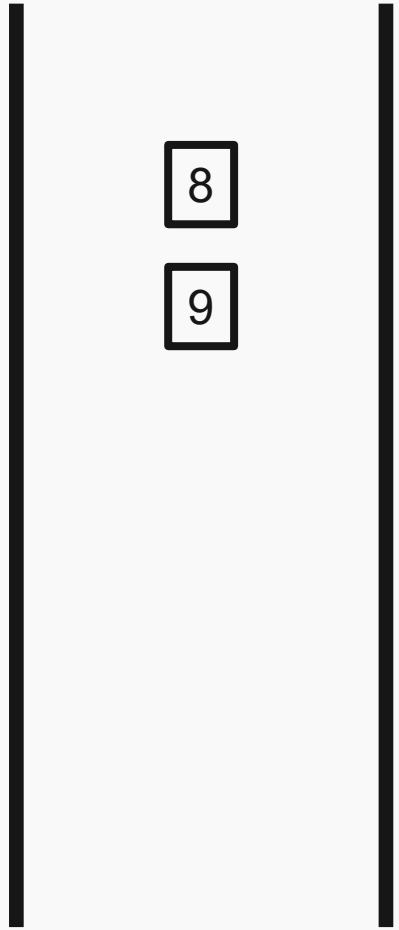


Afișare:

1	2	3	4	5	6

Afișarea unui arbore folosind o coadă

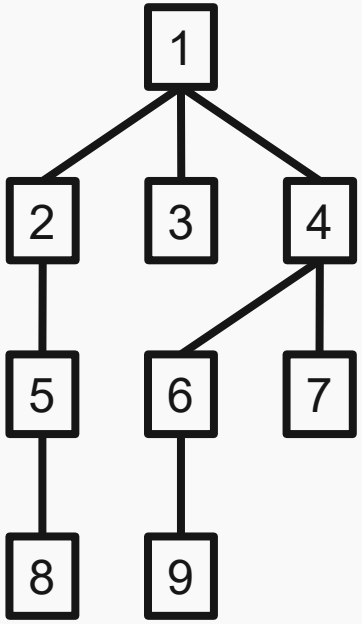


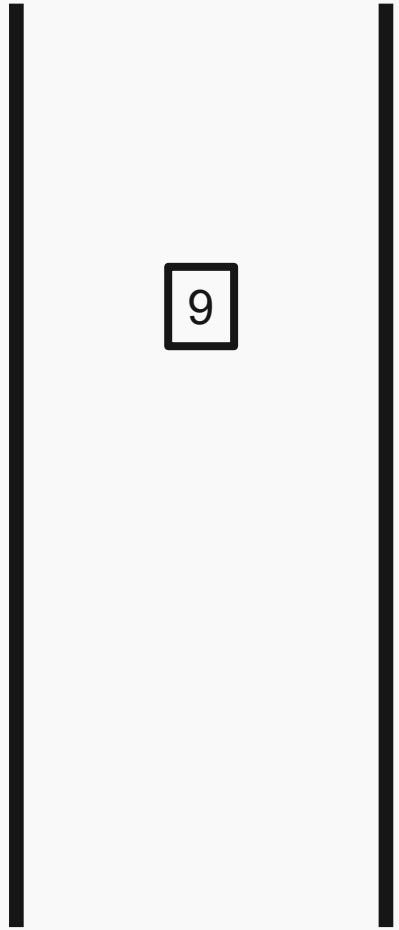


Afișare:

1	2	3	4	5	6	7

Afișarea unui arbore folosind o coadă

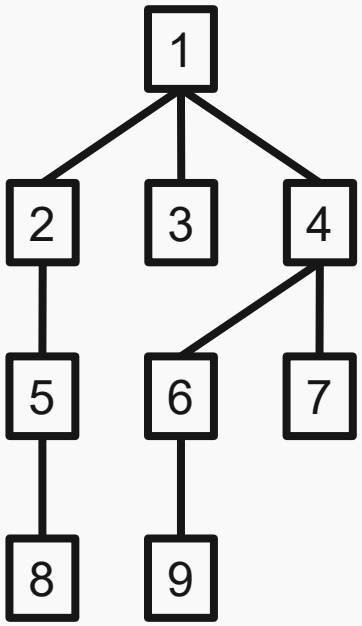


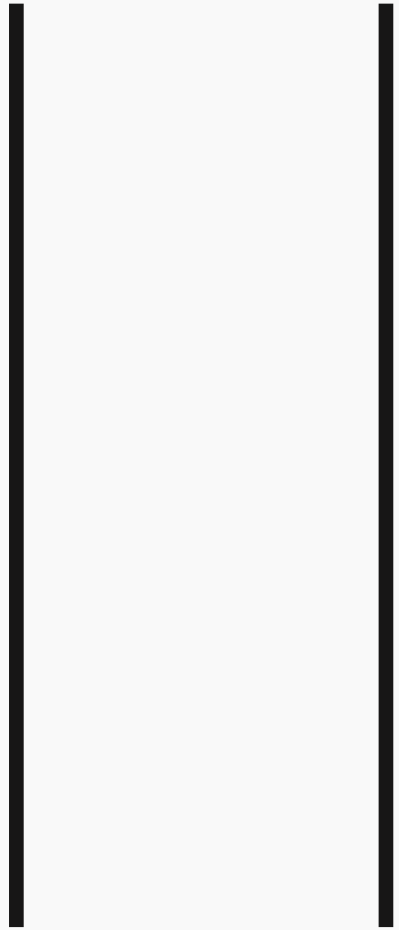


Afișare:

1	2	3	4	5	6	7	8

Afișarea unui arbore folosind o coadă

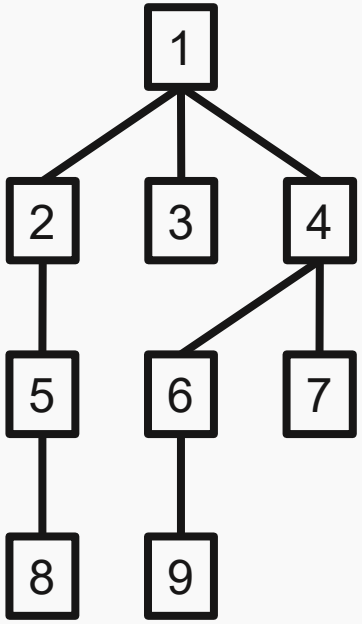




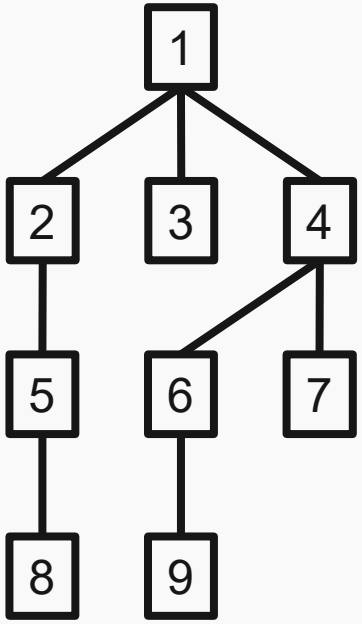
Afișare:

1	2	3	4	5	6	7	8	9

Afișarea unui arbore folosind o stivă



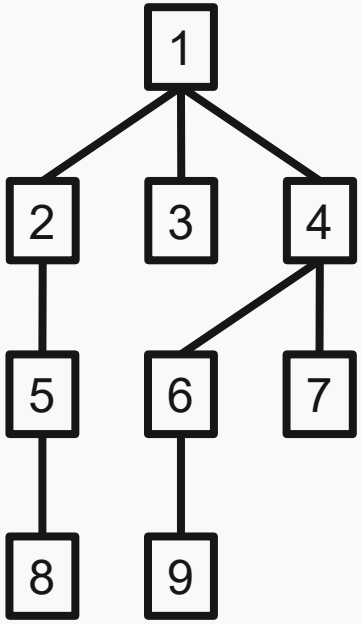
Afișarea unui arbore folosind o stivă

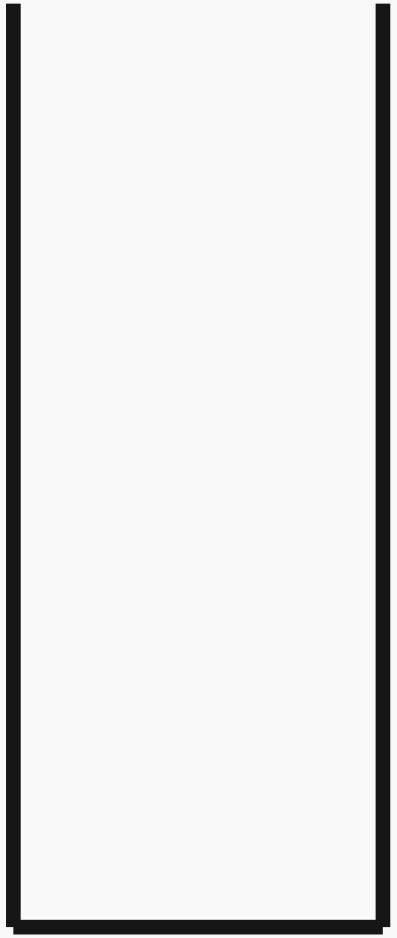


- Adăugăm rădăcina în stivă
- Până ce stiva este goală
  - Scoatem un nod din stivă
  - Adăugăm toți copiii în stivă

Cum va fi afișat arborele?

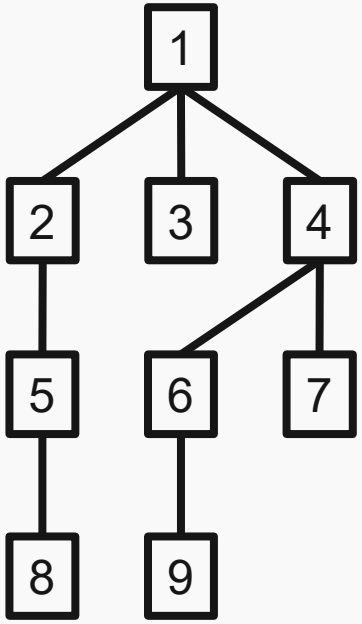
Afișarea unui arbore folosind o stivă

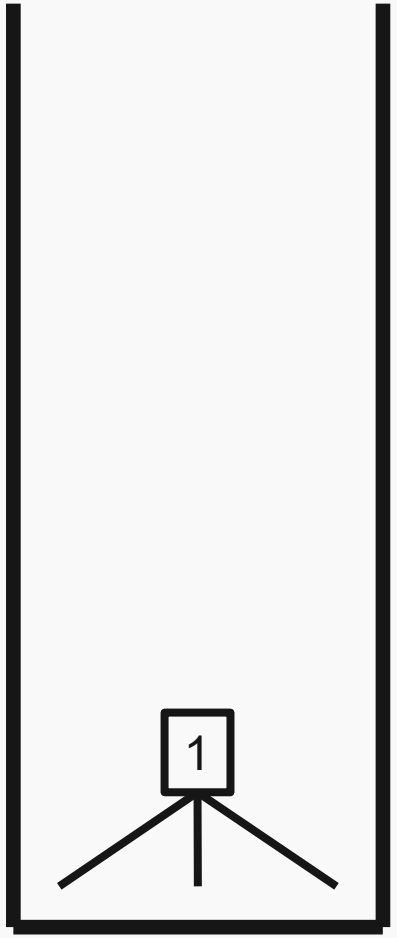




Afișare:

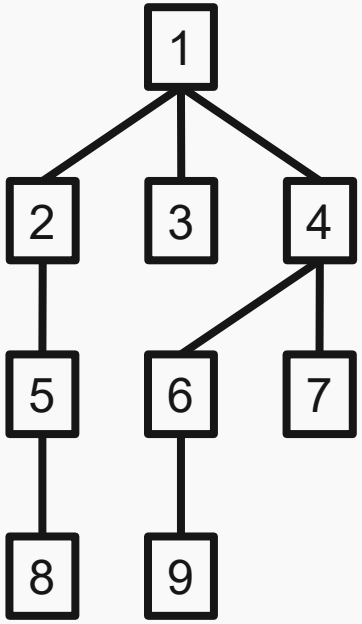
Afișarea unui arbore folosind o stivă

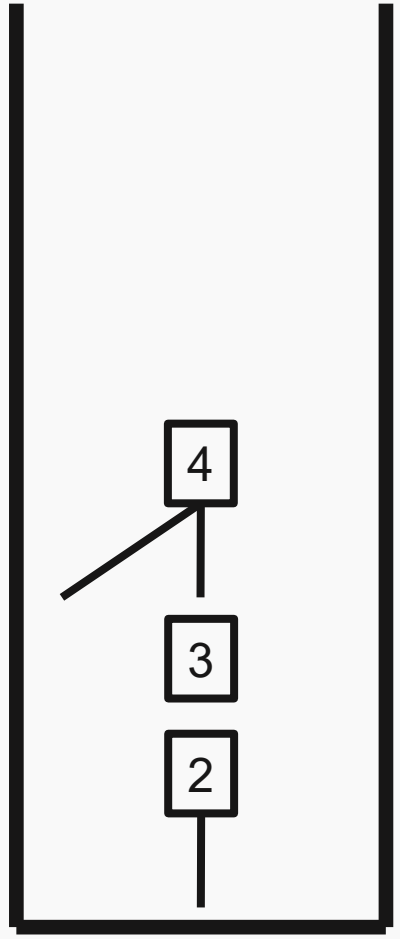




Afișare:

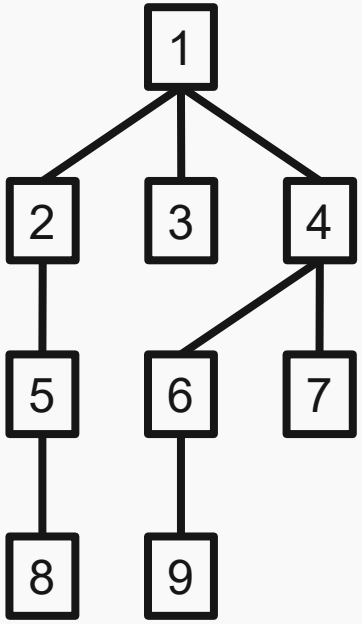
Afișarea unui arbore folosind o stivă

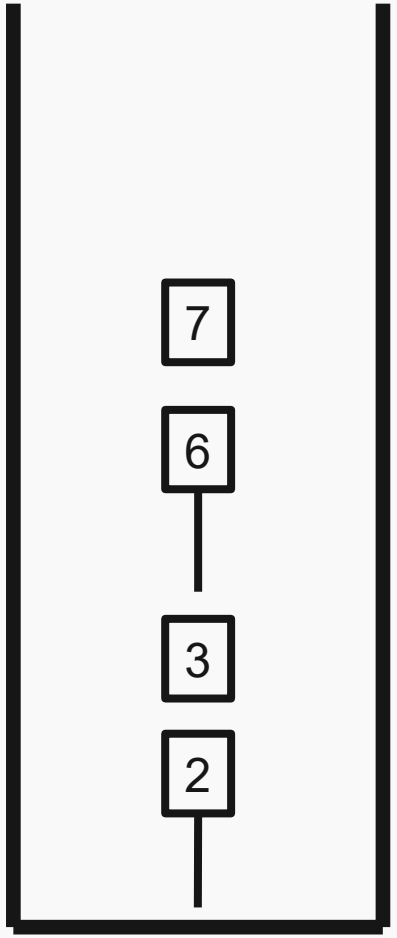




Afișare: 1

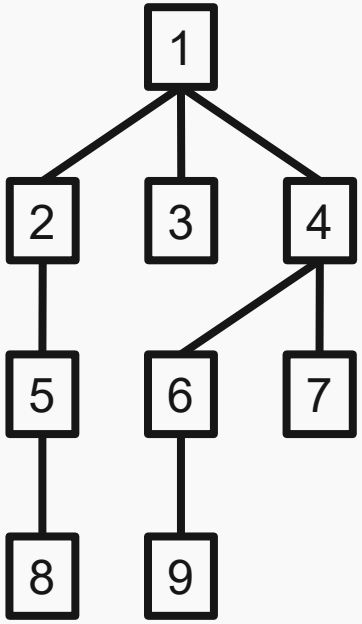
Afișarea unui arbore folosind o stivă

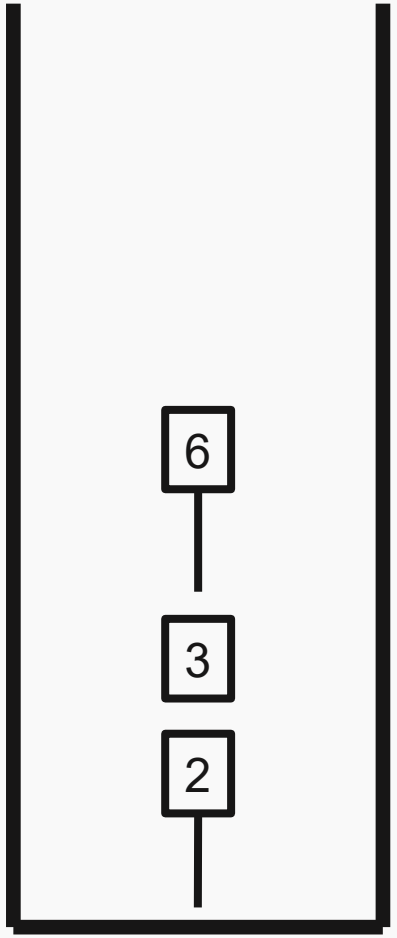




Afișare: 1 4

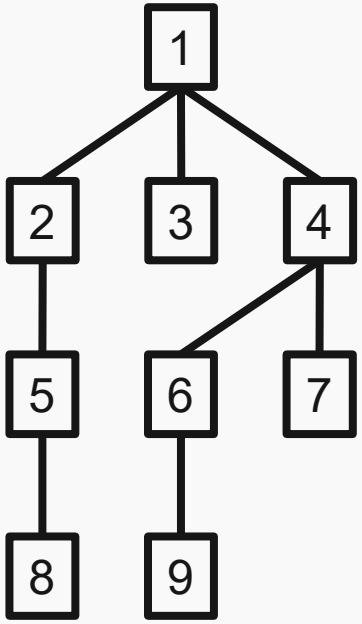
Afișarea unui arbore folosind o stivă

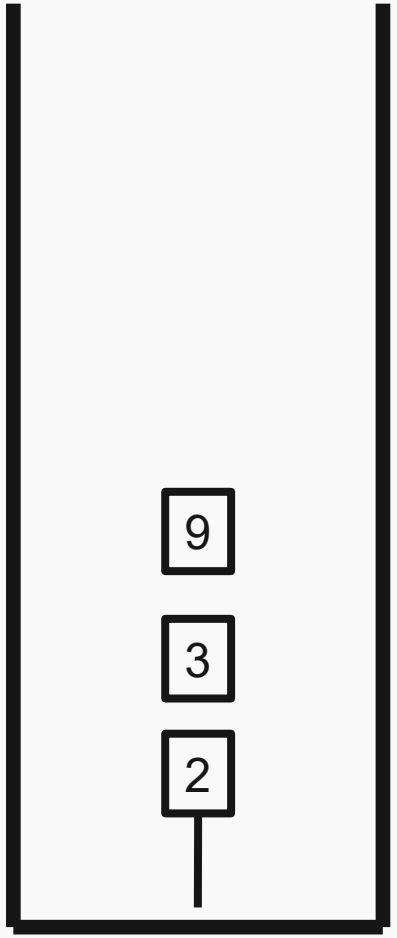




Afișare: 1 4 7

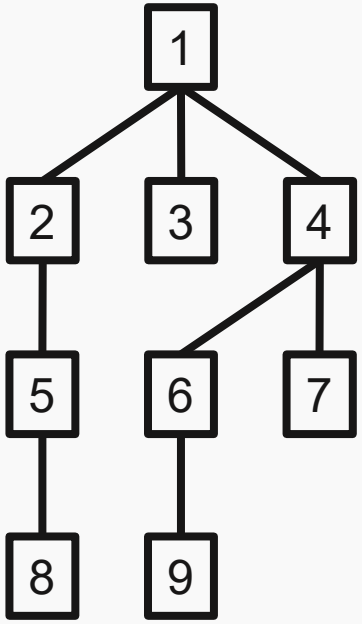
Afișarea unui arbore folosind o stivă

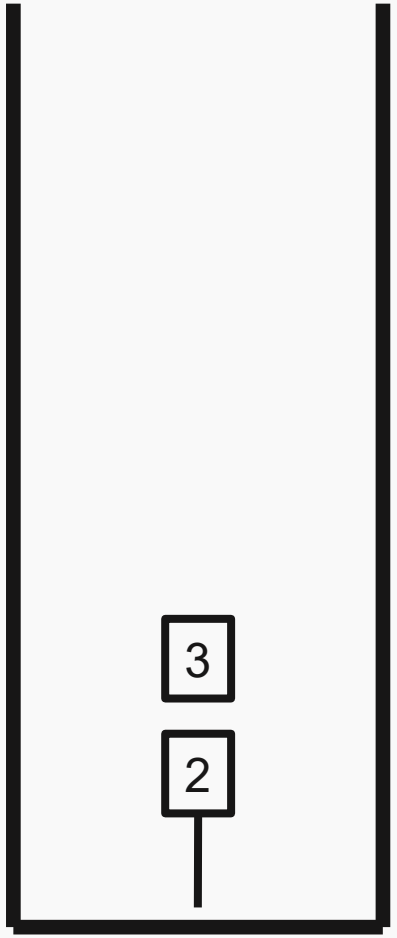




Afișare: 1 4 7 6

Afișarea unui arbore folosind o stivă

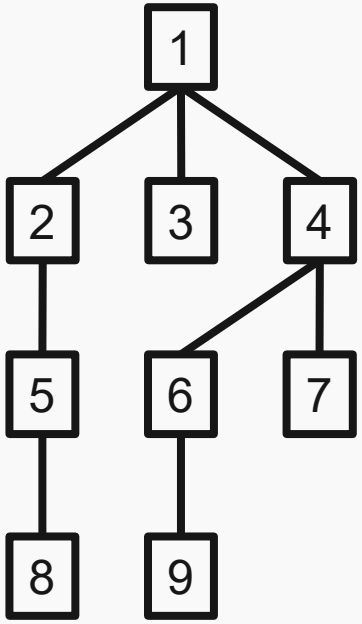


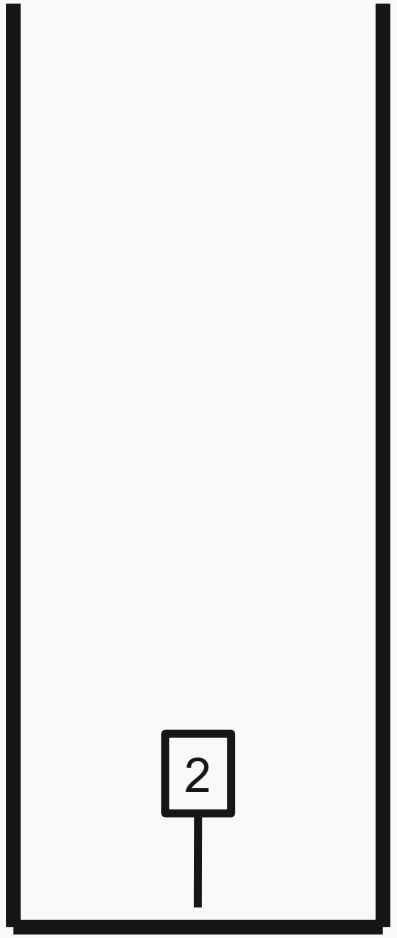


Afișare:

1	4	7	6	9

Afișarea unui arbore folosind o stivă

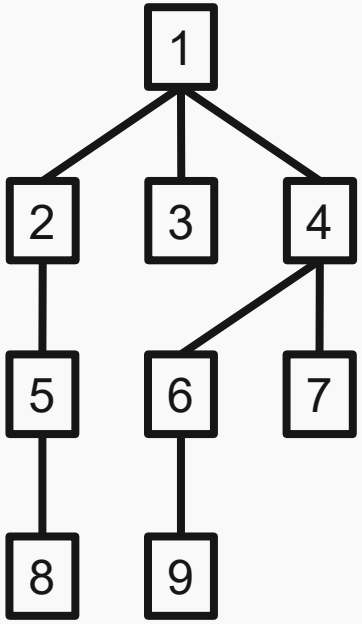


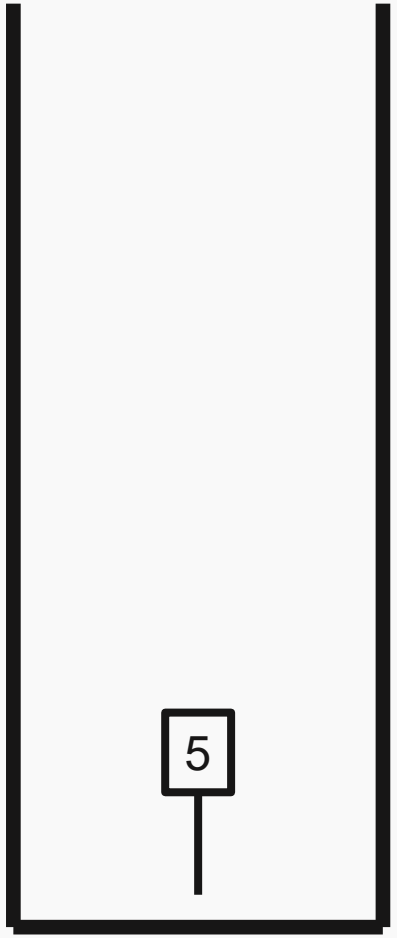


Afișare:

1	4	7	6	9	3

Afișarea unui arbore folosind o stivă

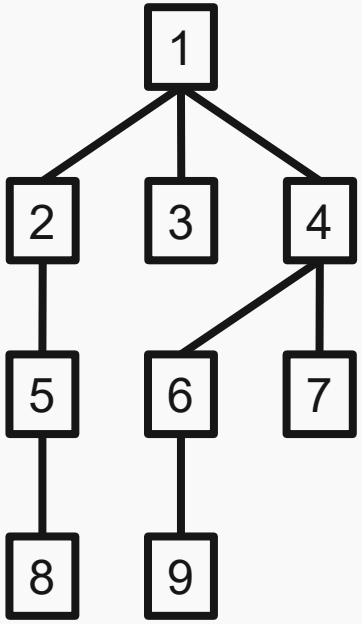


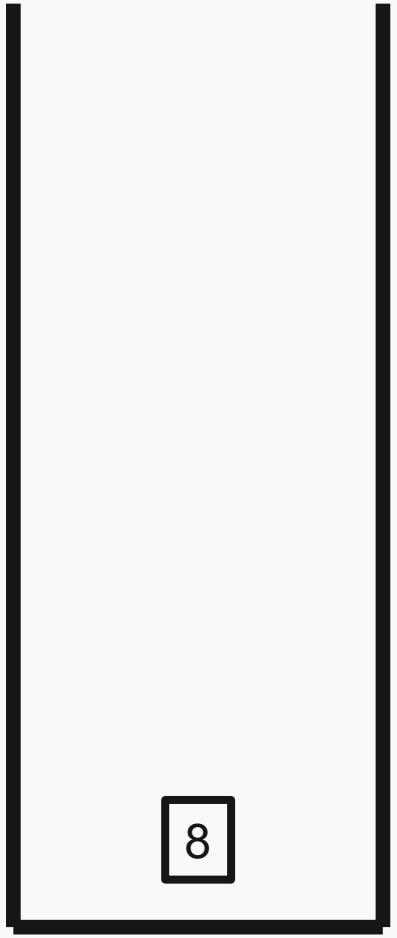


Afișare:

1	4	7	6	9	3	2

Afișarea unui arbore folosind o stivă

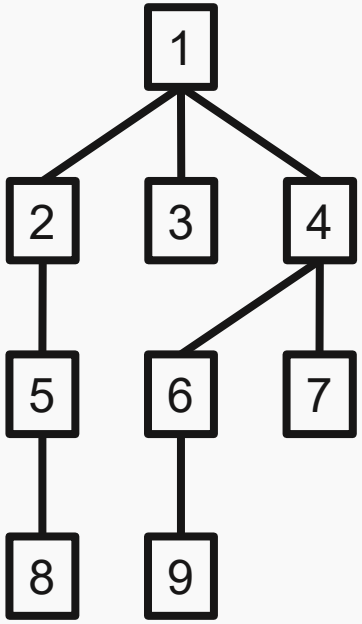


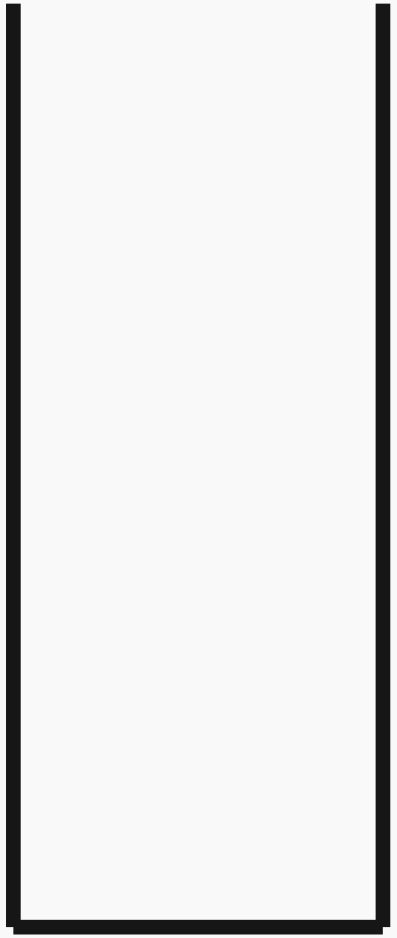


Afișare:

1	4	7	6	9	3	2	5

Afișarea unui arbore folosind o stivă





Afișare:

1	4	7	6	9	3	2	5	8

Skip list

Skip Lists

Algorithms and Data Structures

Jeffrey Vitter Editor

Skip Lists: A Probabilistic Alternative to Balanced Trees

Skip lists are data structures that use probabilistic balancing rather than strictly enforced balancing. As a result, the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

William Pugh

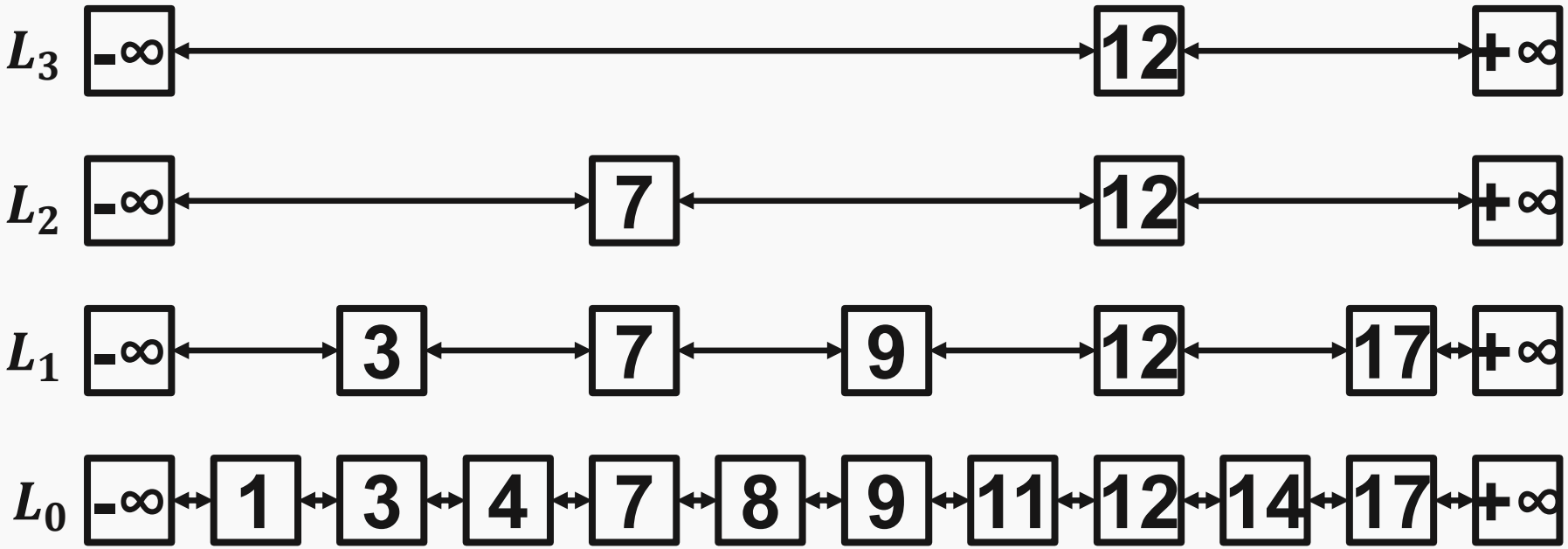
Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that perform very poorly. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input

self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of 1⅓ pointers per element (or even less) and do not require balance or priority information to be stored with each node.

SKIP LISTS

We might need to examine every node of the list when

Skip list



Skip list - search

```
for (i = top; i >= 0; i--) { //Li
   while (searchValue > iterator->next->value) {
      iterator = iterator->next;
   }
   if (searchValue == iterator->value)
      return iterator;
   iterator = iterator->down;
}
```

Skip list - remove

```
search(node);
removeFromAllLevels(node);
```

Skip list - insert

```
prevNode = search(node);
insert(prevNode->next, node, L0);
i = 1;
while (random % 2 != 0) {
      insert(node, Li)
      i++;
}
```

Skip list complexities

	Vector	Listă	Skip list
Complexitate acces	O(1)	O(N)	O(log2(N))
Complexitate inserție	O(N)	O(N)	O(log2(N))
Complexitate inserție capete	O(N)/O(1)	O(1)	O(log2(N))
Complexitate ștergere	O(N)	O(N)	O(log2(N))
Complexitate ștergere capete	O(N)/O(1)	O(1)	O(log2(N))

Bloom Filters

Bloom Filters

Space/Time Trade-offs in Hash Coding with Allowable Errors

BURTON H. BLOOM
Computer Usage Company, Newton Upper Falls, Mass.

In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a

iteratively generating hash addresses of cells. The contents of these cells are then compared with the test messages. A match indicates the test message is a member of the set; an empty cell indicates the opposite. The reader is assumed to be familiar with this and similar conventional hash-coding methods [1, 2, 3].

The new hash-coding methods to be introduced are suggested for applications in which the great majority of messages to be tested will not belong to the given set. For these applications, it is appropriate to consider as a unit of time (called *reject time*) the average time required to classify a test message as a nonmember of the given set. Furthermore, since the contents of a cell can, in general, be recognized as not matching a test message by examining only part of the message, an appropriate assumption will be introduced concerning the time required to access individual bits of the hash area.

In addition to the two computational factors, reject time and space (i.e. hash area size), this paper considers a third computational factor, allowable fraction of errors. It will be shown that allowing a small number of test messages to be falsely identified as members of the given set will permit a much smaller hash area to be used without increasing the reject time. In some practical applications,

Silly Bloom Filters

5	0000_0101	OR
1	0000_0001	OR
3	0000_0011	OR
32	0010_0000	OR
33	0010_0001	
	=	
	????_????	

Silly Bloom Filters

5	0000_0101	OR
1	0000_0001	OR
3	0000_0011	OR
32	0010_0000	OR
33	0010_0001	
		=
Rez	0010_0111	

Silly Bloom Filters

Rez	0010_0111	AND
Număr	inițial	
	=	
??	????_????	

Silly Bloom Filters

Rez	0010_0111	AND
1	0000_0001	
	=	
1	0000_0001	

Silly Bloom Filters

Rez	0010_0111	AND
5	0000_0101	
	=	
5	0000_0101	

Silly Bloom Filters

Rez	0010_0111	AND
33	0010_0001	
	=	
33	0010_0001	

Silly Bloom Filters

Rez 0010_0111 AND

x ????_????

=

x ????_????

Găsiți x diferit de numerele inițiale scris în zecimal și binar:

Confusion matrix

	Pozitive reale	Negative reale
Pozitive prezise	True pozitive	False positives
Negative prezise	False negatives	True negatives

Confusion matrix

		Fac parte din calcul inițial	NU fac parte din calcul inițial
		Pozitive reale	Negative reale
x & BF == x	Pozitive prezise	True pozitives	False positives
x & BF != x	Negative prezise	False negatives	True negatives

Ce poate întoarce un (silly) bloom filter testat pe multiple numere?

Confusion matrix – bloom filters

		Fac parte din calcul inițial	NU fac parte din calcul inițial
		Pozitive reale	Negative reale
x & BF == x	Pozitive prezise	True pozitives	False positives
x & BF != x	Negative prezise	~~False negatives~~	True negatives

Bloom filter - creare

- Un set de funcții hash (h1(); h2(); … hk())
  - **Funcțiile returnează valori între 1 si N**
- Un șir de **N** biți initializați cu **0**

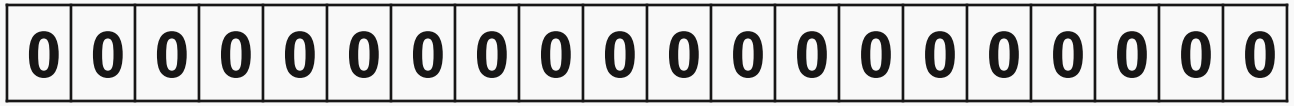
- Un input **x** pe care dorim să îl adăugăm filtrului, setăm cu **1** pozițiile hi(), oricare ar fi i.

Bloom filter - Verificare apartenență

Pentru un **y** pentru care dorim să știm dacă este în filtru.

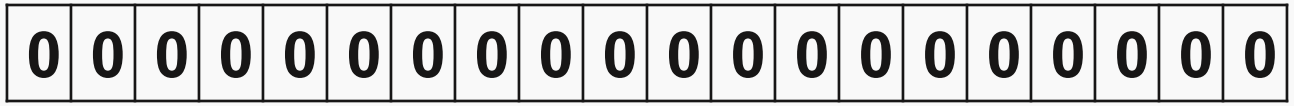
- Aplicăm toate funcțiile hi()
- Dacă pe toate pozițiile returnate găsim **1** în vectorul de biți putem considera că **y** aparține setului de numere cu care a fost creat filtrul. (Putem avea **false positives**)
- Dacă pe oricare poziție din cele returnate găsim **0** în vectorul de biți putem fi siguri că **y** nu a fost introdus în Bloom filter

Bloom filter – inițializare

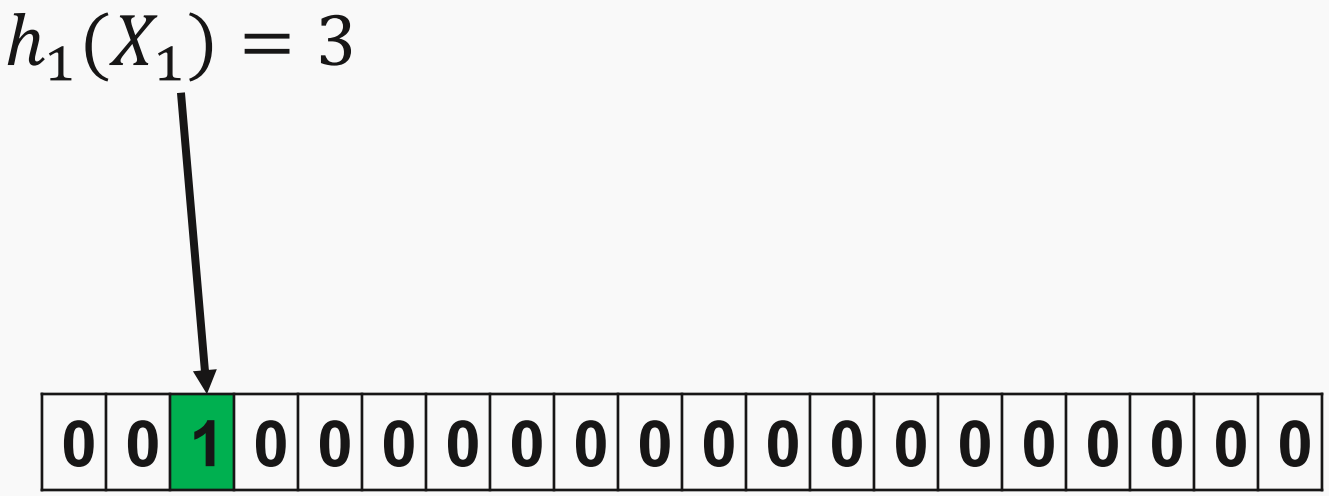


Bloom filter – inițializare

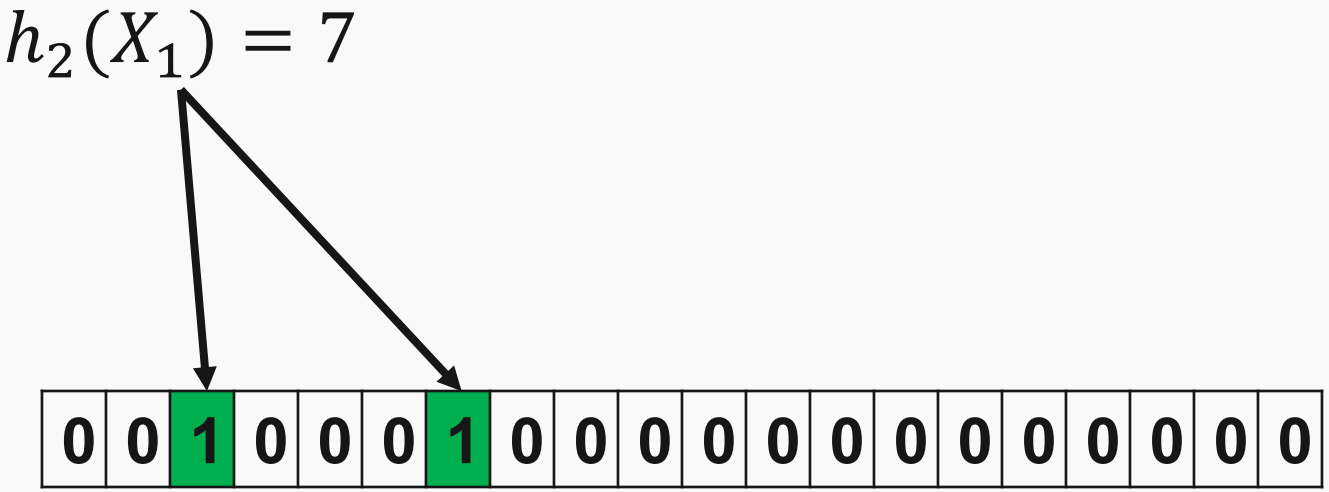
X_1



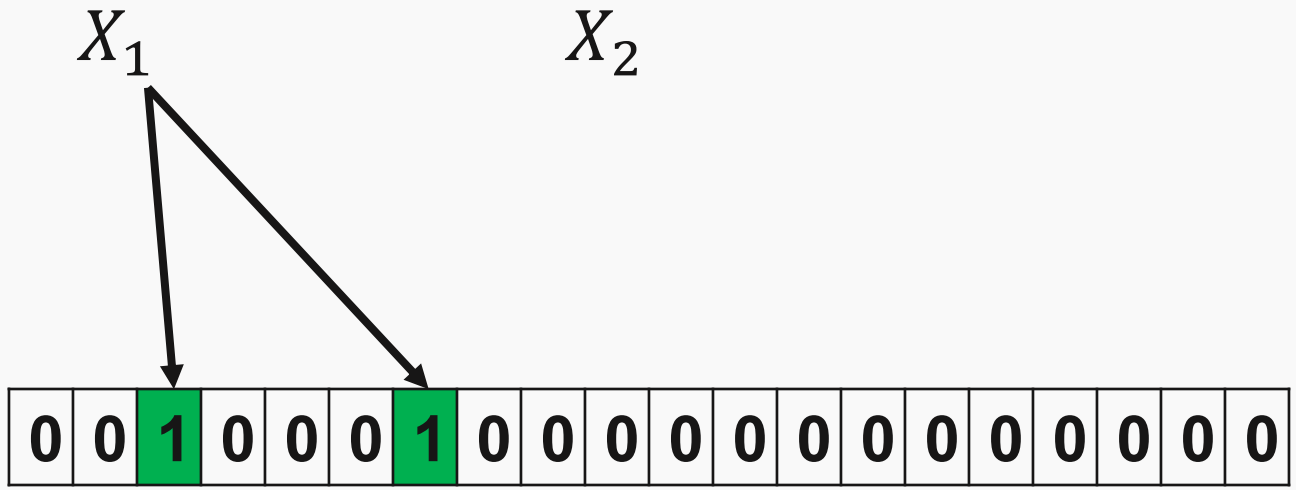
Bloom filter – inițializare



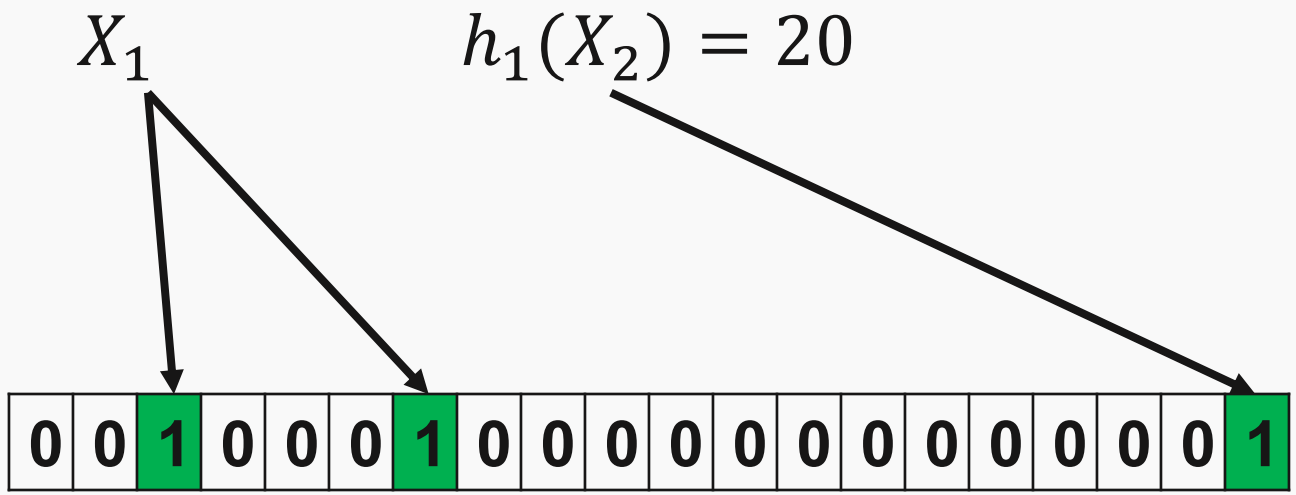
Bloom filter – inițializare



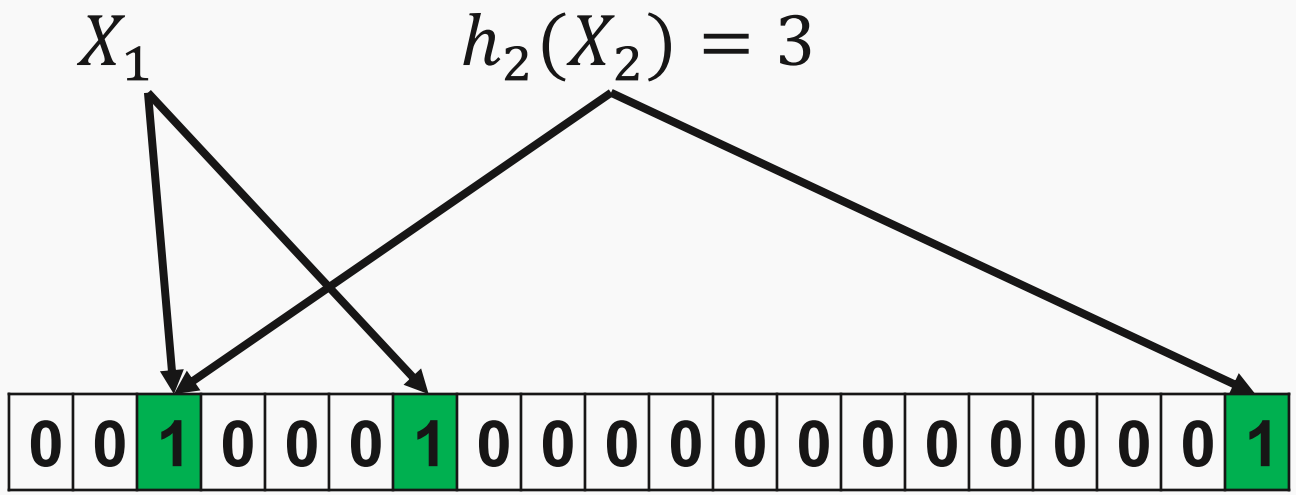
Bloom filter – inițializare



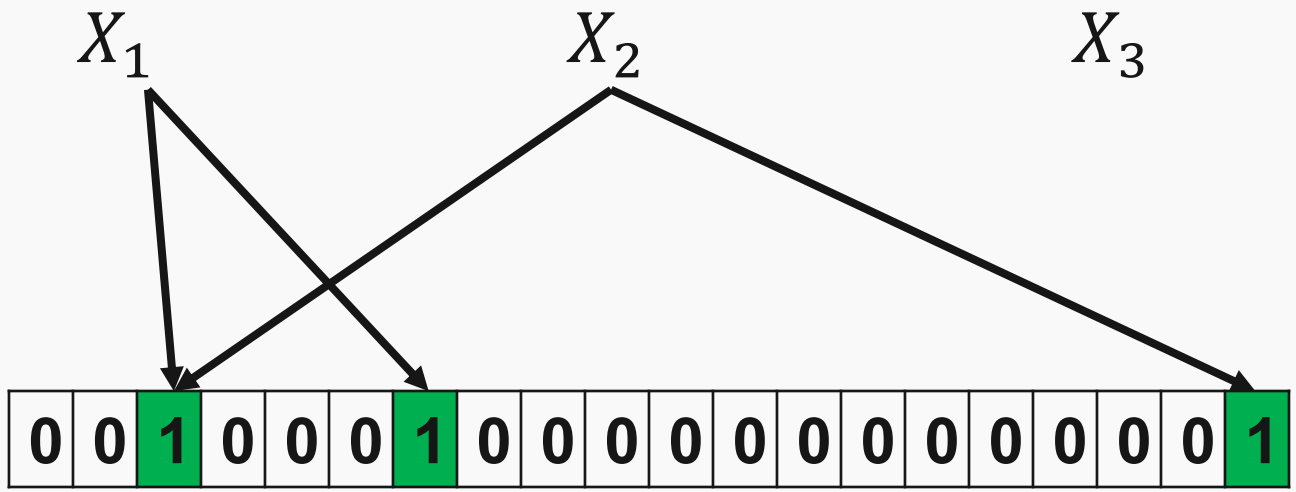
Bloom filter – inițializare



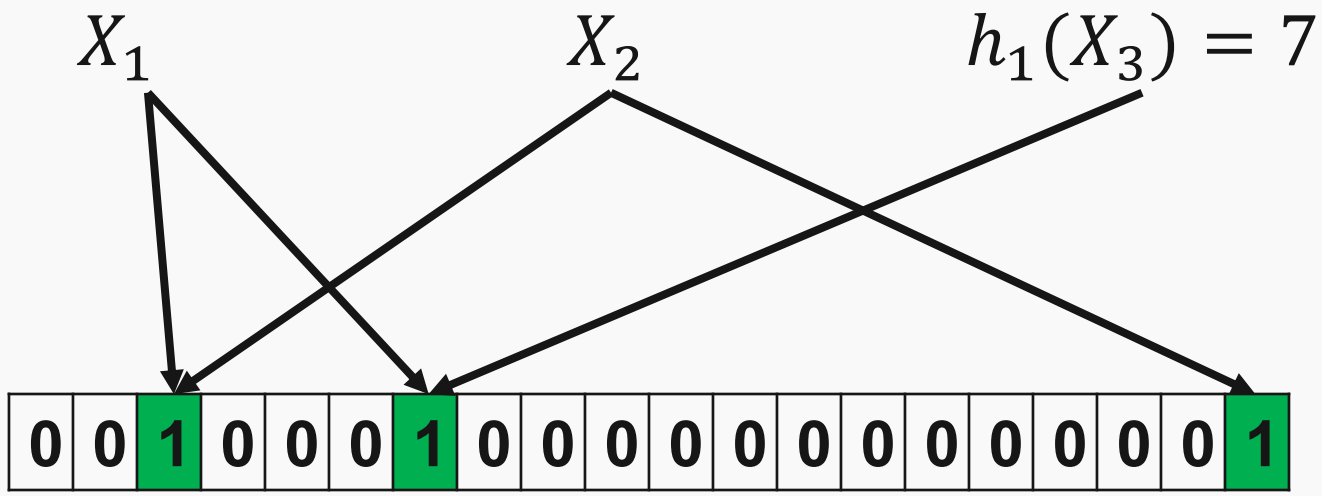
Bloom filter – inițializare



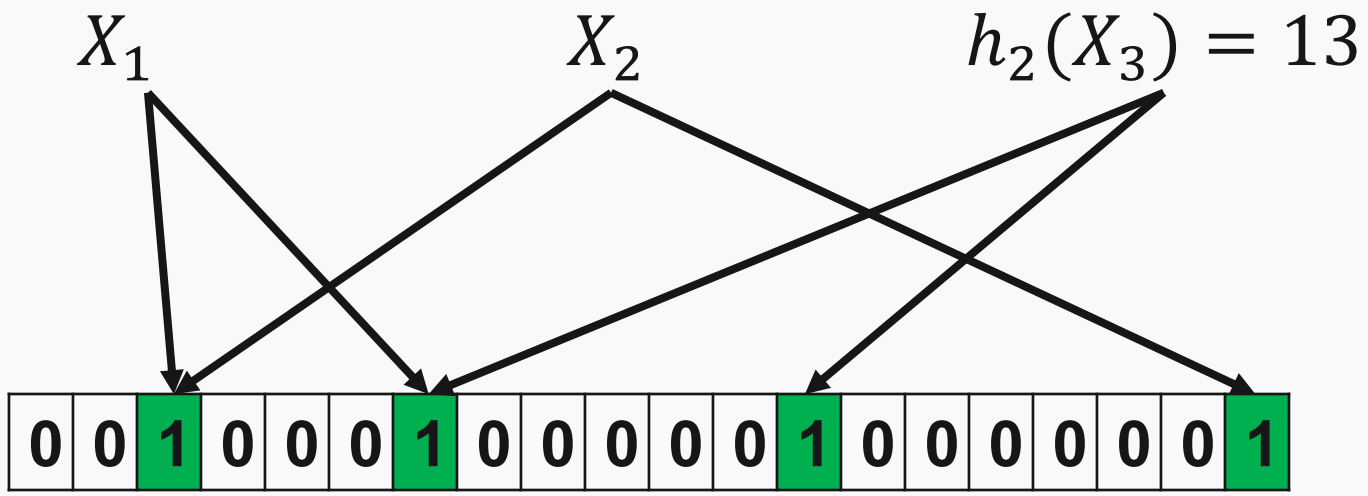
Bloom filter – inițializare



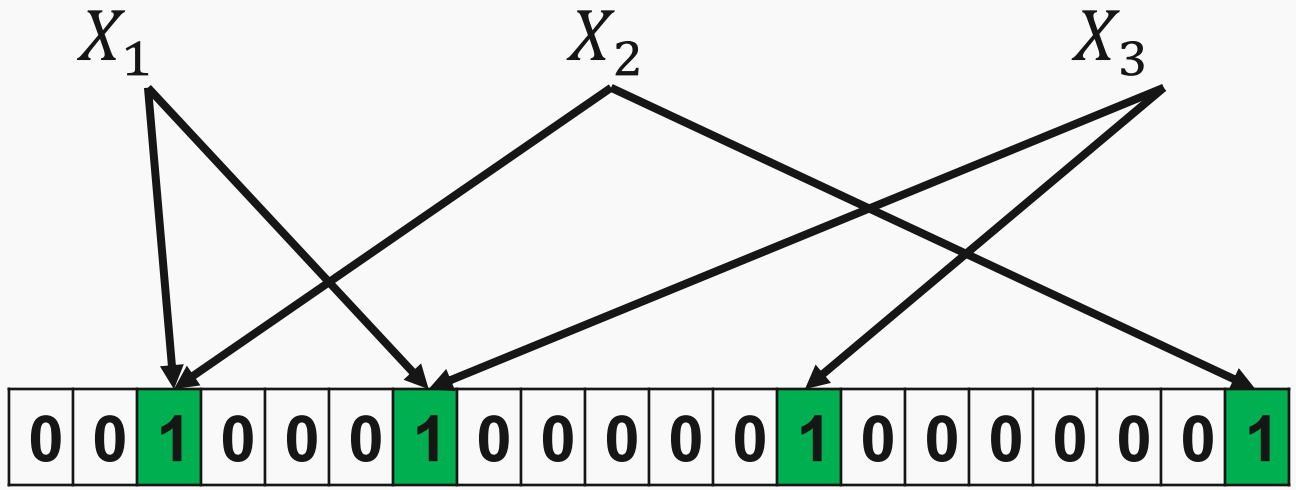
Bloom filter – inițializare



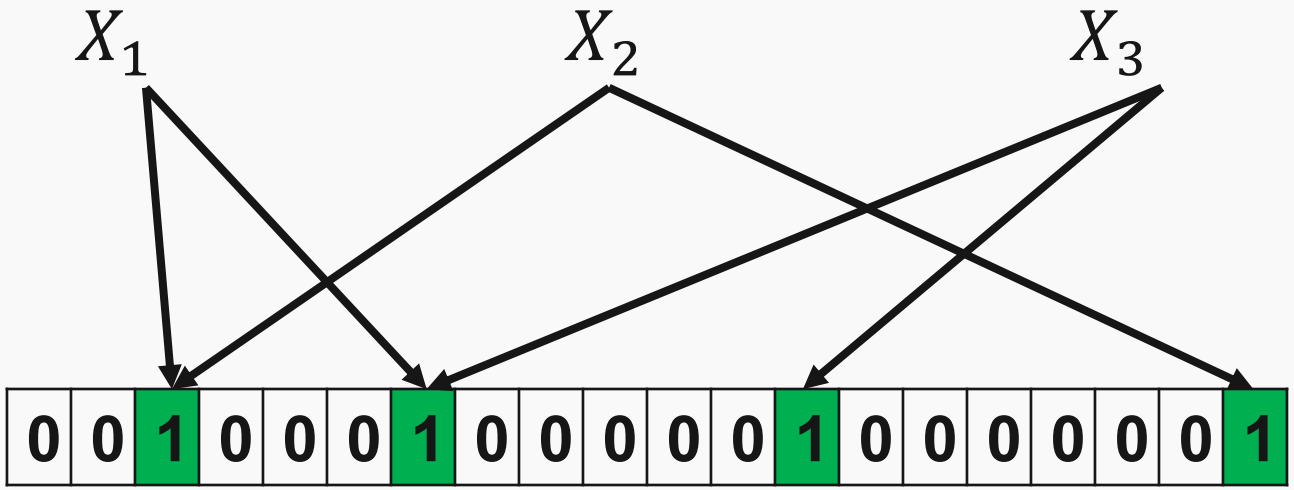
Bloom filter – inițializare



Bloom filter – inițializare

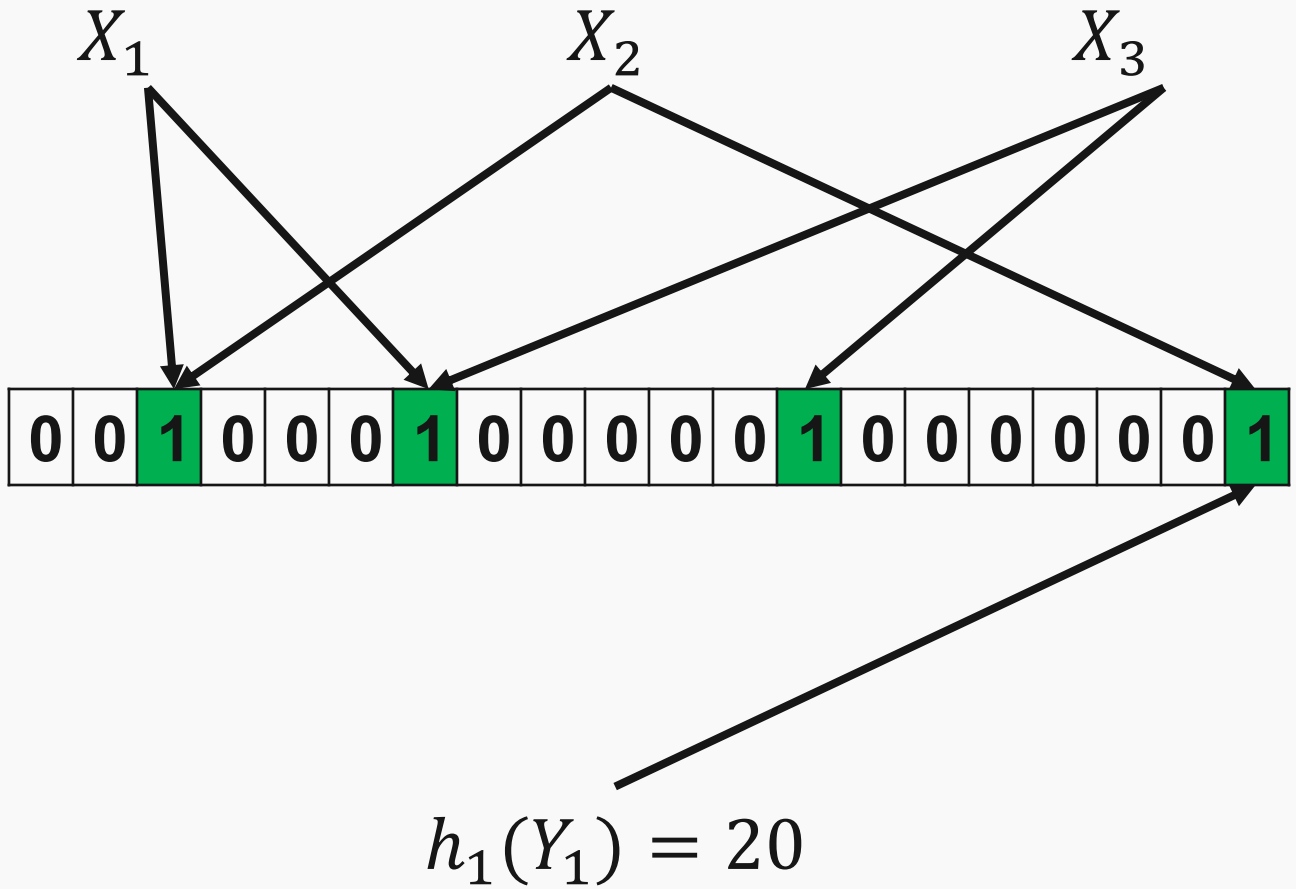


Bloom filter – folosire

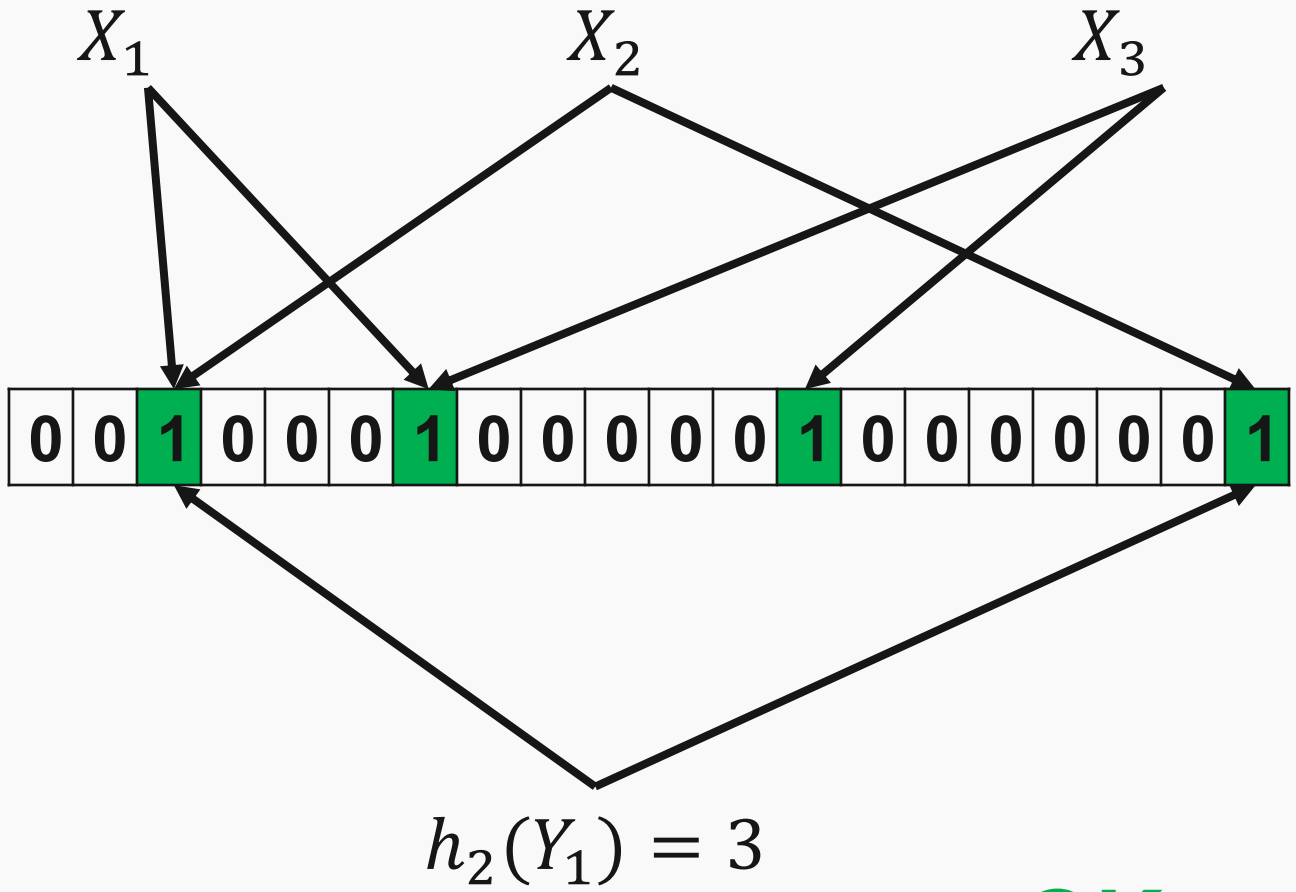


$$Y_1 = X_2$$

Bloom filter – folosire

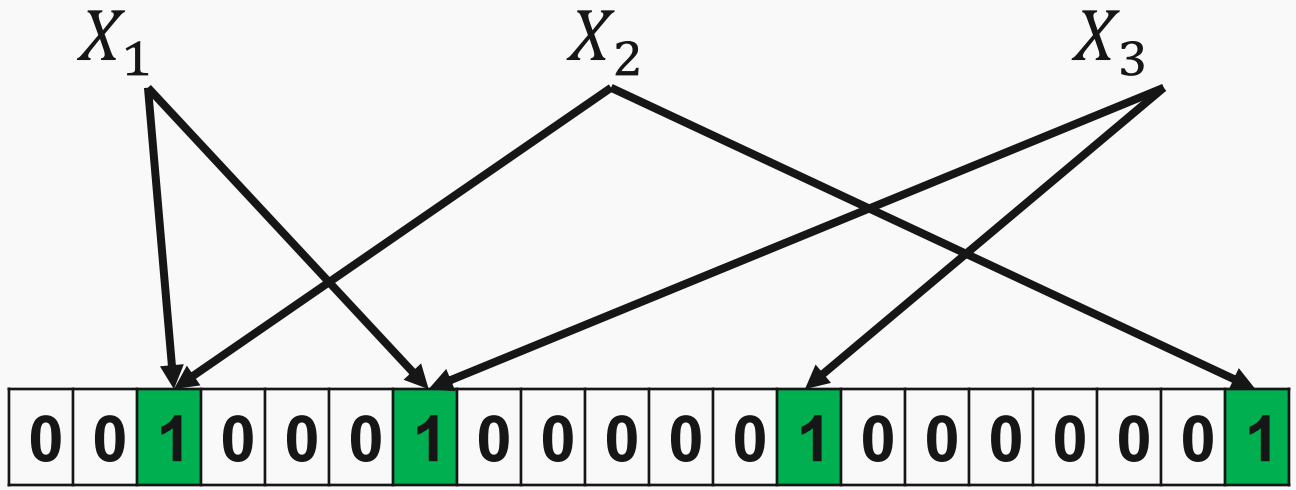


Bloom filter – folosire



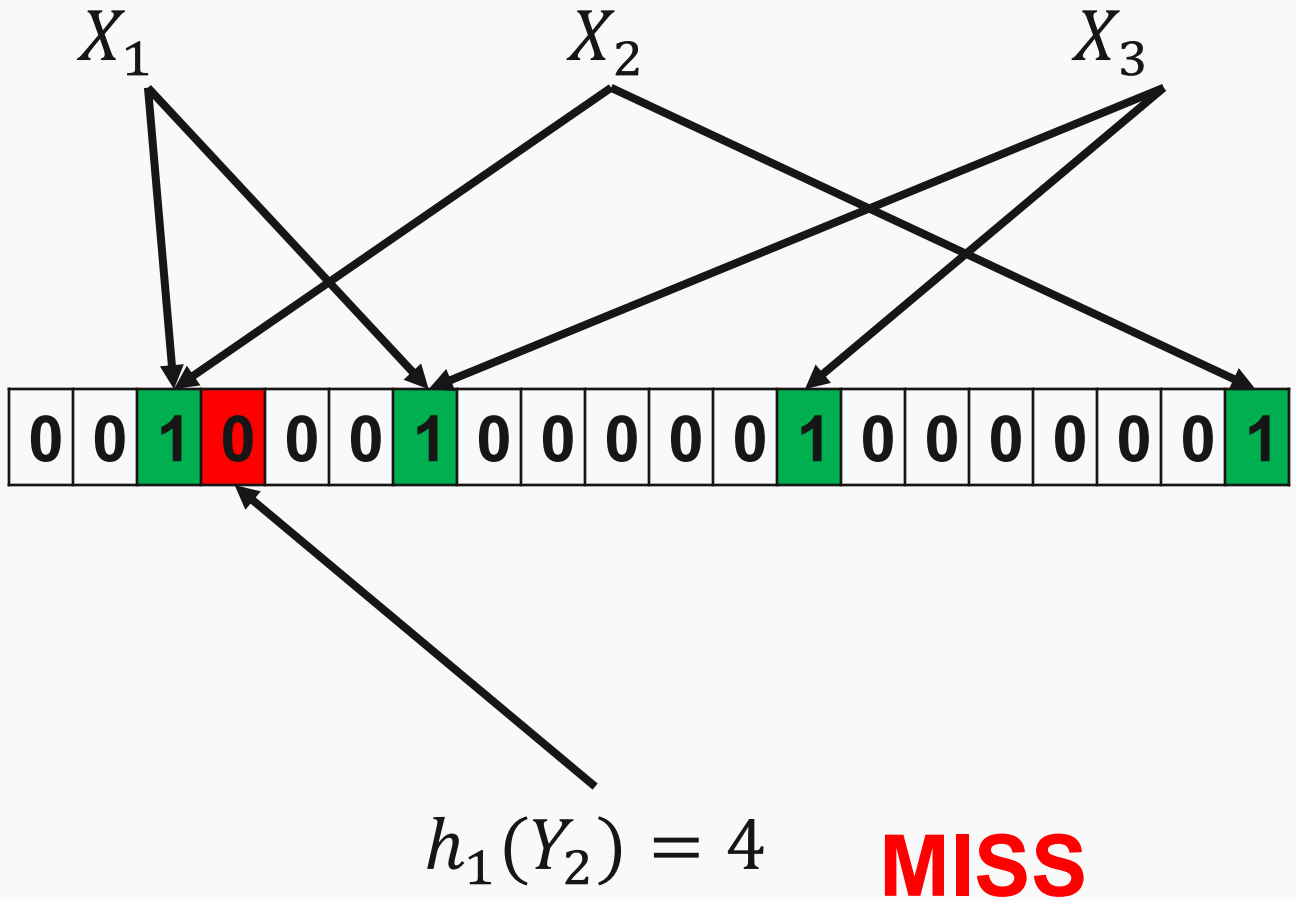
OK

Bloom filter – folosire

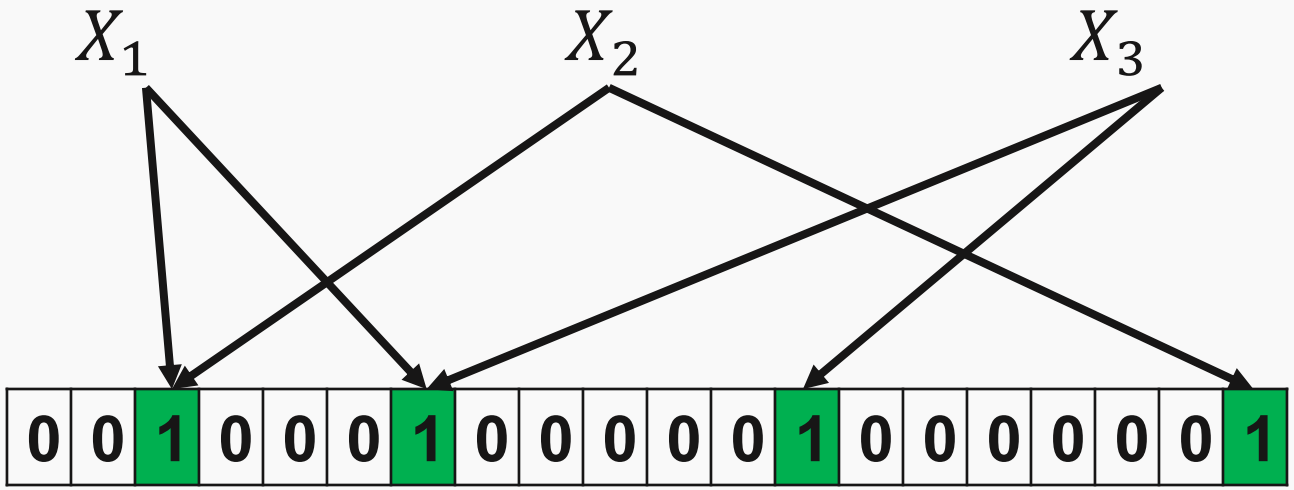


Y_2

Bloom filter – folosire

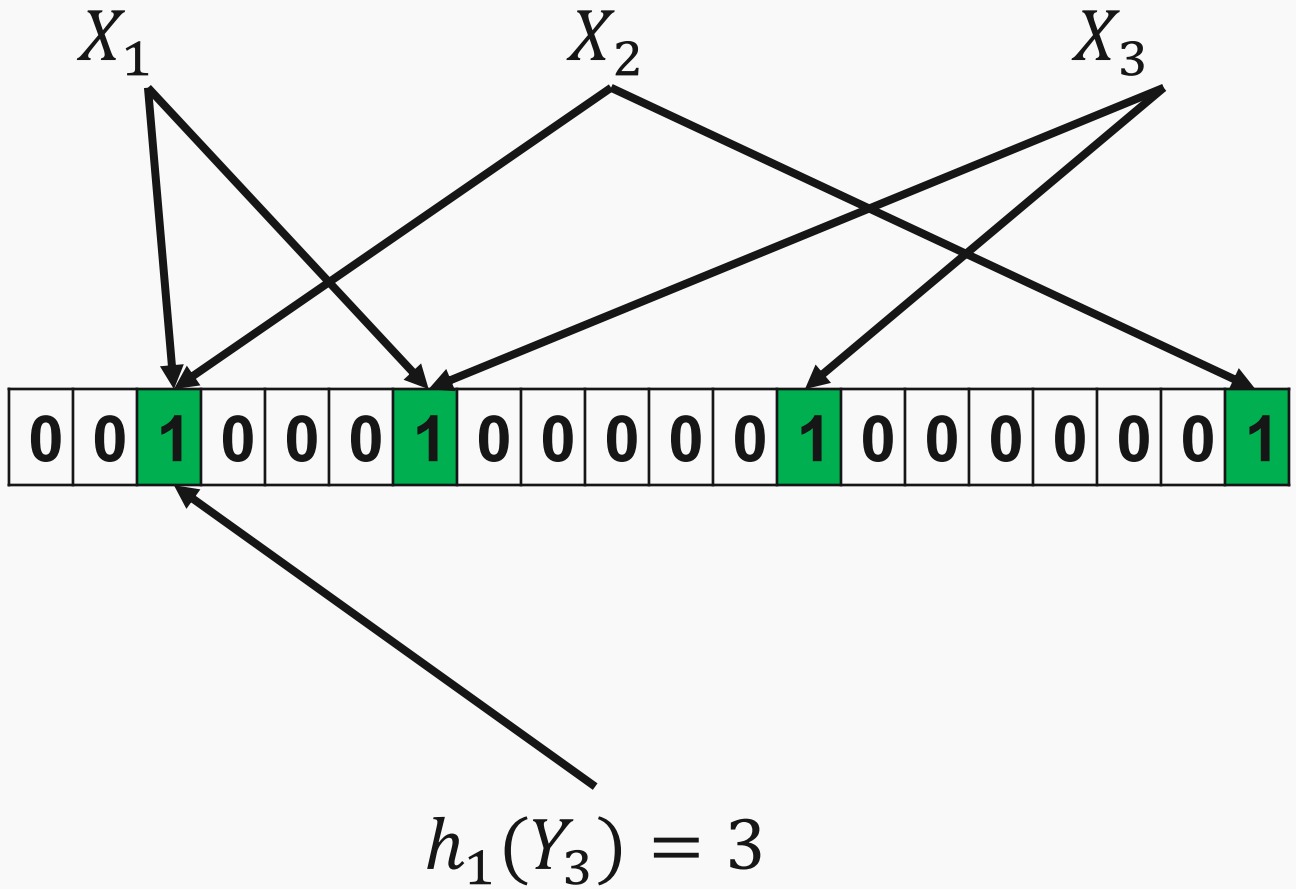


Bloom filter – folosire

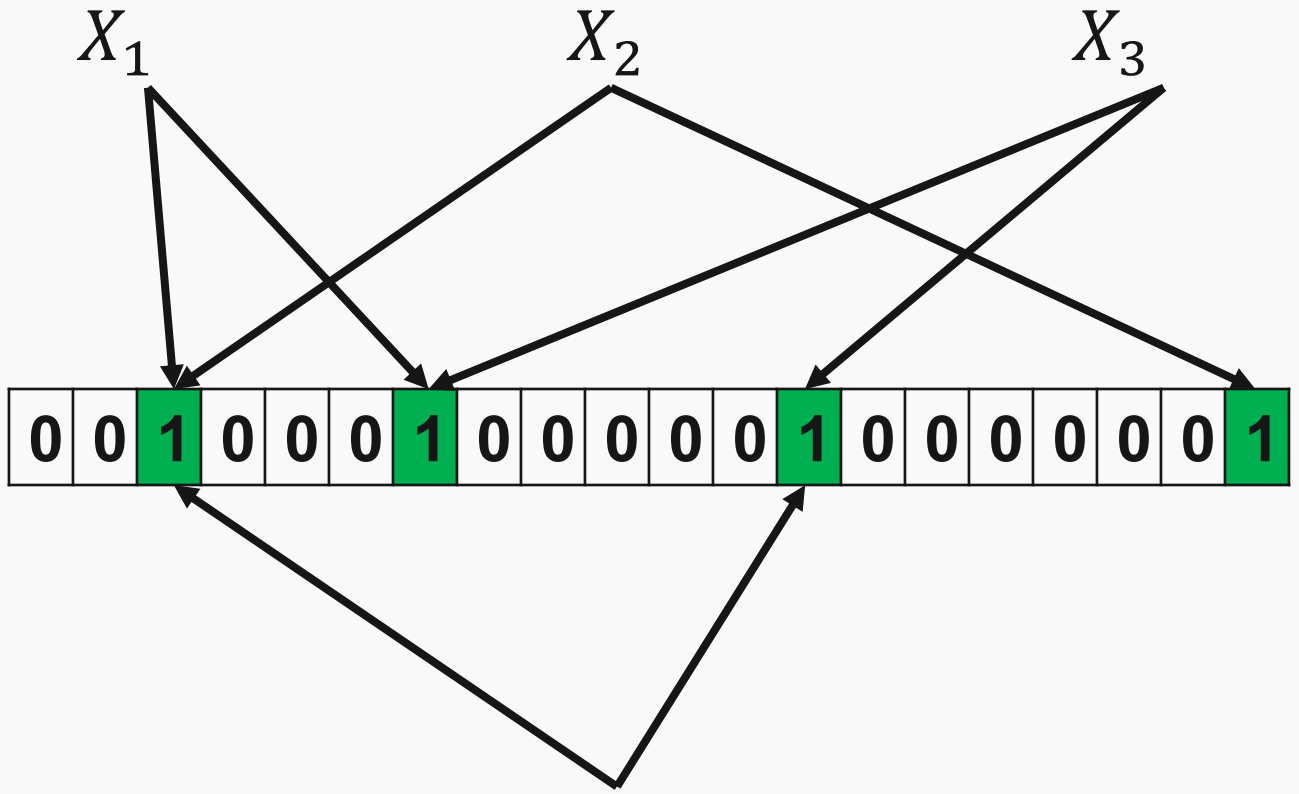


$$Y_3! = X_1 \qquad Y_3! = X_2 \qquad Y_3! = X_3$$

Bloom filter – folosire



Bloom filter – folosire



FALSE OK

Bloom filter – Performanță

- Probabilitate de false positive = $\left(\frac{(\#1)}{N}\right)^k$
- Procent de biți egali cu **0**:

$$(1-\frac{1}{N})^{d*k} = (1-\frac{1}{N})^{\frac{1}{N}N*d*k} = e^{-\frac{d*k}{N}}$$

- Probabilitate de false positive:

$$(1-e^{-\frac{d*k}{N}})^k$$

d numărul de valori din filtru
k numărul de funcții de hash
N numărul de biți

Bloom filter

- Avantaje
  - Verificare extrem de rapidă (chiar mai rapid decât la hash map)
    - La hash map am avea O(1) dar constanta poate fi mare, unele cazuri O(N)
  - Memorie extrem de puțină (foarte bun pentru sisteme distribuite)
- Dezavantaj
  - Nu poate fi folosit mereu deoarece este aproximativ (false positives)
- Folosire:
  - Baze de date distribuite (se filtrează cererile)
  - Distribuirea liste imense (gen URL-uri nesigure)
  - Web Crawling – listă cu URL-uri deja crawluite
  - Sistem recomandări – bloom filter pentru fiecare utilizator verifică dacă a primit deja o recomandare