Structuri de date si algoritmi

Curs, IS – An II

100101001010
0101011000101001010 0
010101100010100101001010010 100
0010101100010100101001010010100 10 10
01010110000101001010010100101001 11
10001010010100101001010010
0100101001010010100 1010
101100
011000
01100
011000
101100
00101
110001010010011
010101100010100101001010 01

Cozi

Tipul abstract de dată – Coadă (En. Queue)

Coadă la un magazin Apple din New York (2011)

Drum cu sens unic și o singură bandă

Coada – tip abstract de date (TAD) de tip FIFO (sau LILO)

TAD Coadă

O coadă este o listă specială organizată pe principiul FIFO (First In – First Out, primul sosit – primul servit) sau LILO (Last In – Last Out):

- inserarea se realizează numai la sfârşitul listei;
- este şters numai elemental de la începutul listei.

Operaţii pe structuri de tipul coadă

- Put: un atom (nod) este inserat la sfârşitul cozii (echivalent *Insert*, *Enqueue*);
- Get: primul atom (nod) al cozii este îndepărtat (echivalent *Delete*, *Dequeue*);
- Front: returnează informația primului atom (nod) al cozii fără să-l șteargă (echivalent *Top*, *Peak*);
- Init: inițializează coada (coadă vidă);
- IsEmpy: întoarce 1 (true) dacă coada este goală și 0 (false) dacă coada conține atomi.

TAD Coadă

Exemple de utilizare a cozilor:

- Playlist
- Cozi mesaje intr-o retea de comunicatii
- Comenzile de tipărire în coada de aşteptare a imprimantei
- Comenzile clienţilor pe un portal Web

Există situații speciale în care principiul FIFO este încălcat (de pildă cererile de întreruperi către sistemul de operare, sau modelarea cazului în care o persoană cu nevoi speciale ”sare” peste coadă) -> *Coada de priorități* pe care o vom dicuta peste câteva săptamâni.

Implementare (similar listelor sau stivelor)

- Dinamic
- Static

Coada – Implementare dinamică

Deoarece operațiile de inserare (put), ștergere (get) și consultare sunt permise doar la unul din cele două capete ale cozii, în cazul implementării dinamice este nevoie de doi pointeri către primul și ultimul atom (nod) din structură:

- ✓ head (primul nod inserat) - astfel încât să *obţinem /(get, front)* primul element din coadă;
- ✓ tail (ultimul nod inserat) - astfel încât să putem *insera/put* un nou element în coadă.

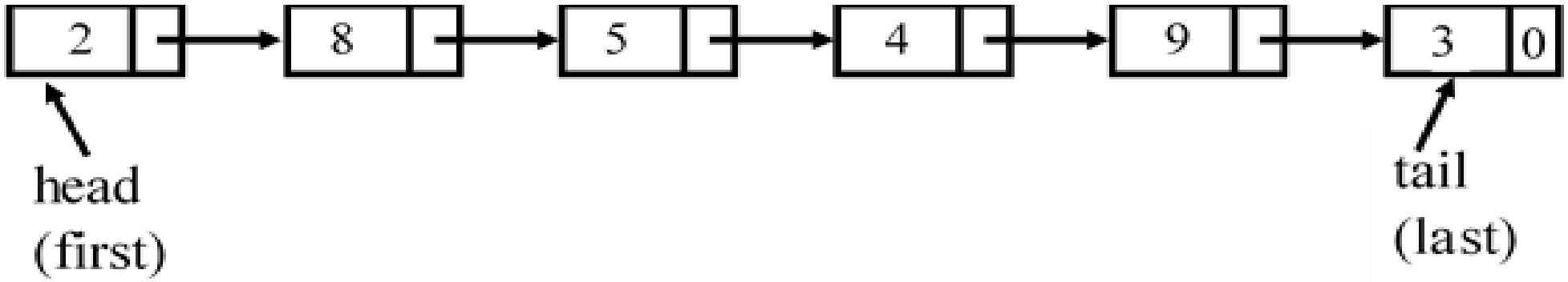


Fig. C1

Coada – Implementare dinamică

Definiție

```
typedef  int atom;

struct Element
{
   Atom info;
   Element *succ;
};

struct Queue
{
   Element *head, *tail;//pointeri la primul
                        //și ultimul element
} ;
```

Funcții

```
Queue InitQ(void);
int IsEmptyQ(Queue q);
void Put(Queue &q, Atom x);
Atom Get(Queue &q);
Atom Front(Queue q);
```

Coada – Implementare dinamică

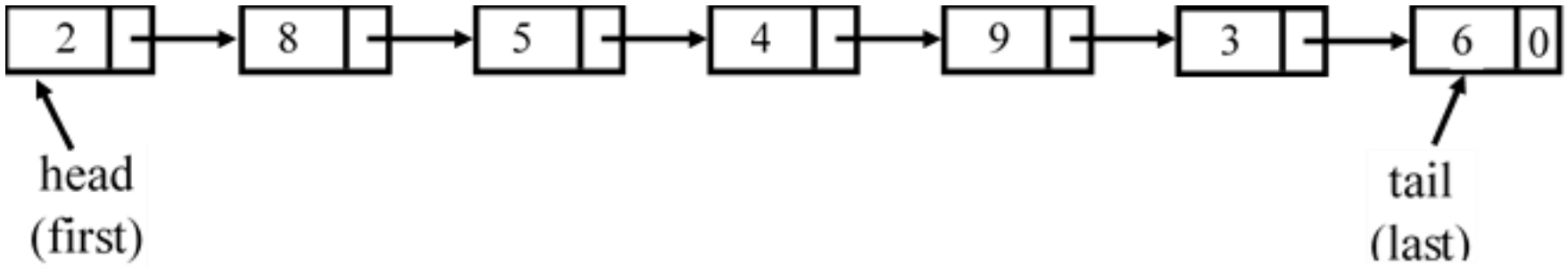
```
Queue InitQ(void)
{
    Queue q;
    q.head = q.tail = 0;//ambii pointeri nuli
   return q;
}

int IsEmptyQ(Queue q)
{
   return (q.head == 0 && q.tail==0);//testez dacă ambii pointeri sunt nuli
}
```

Coada – Implementare dinamică

Put – inserarea unui nod în coada din figura C1

```
Queue q;
p<-get_sp(); // se alocă memorie pentru un nou Element
info(p)<-val; // 6 in cazul dat
succ(p)<-0; // va fi ultimul element
succ(q.tail)<-p
q.tail <- p
```

De tratat separat situatia cand coada este goala inainte de adaugare!

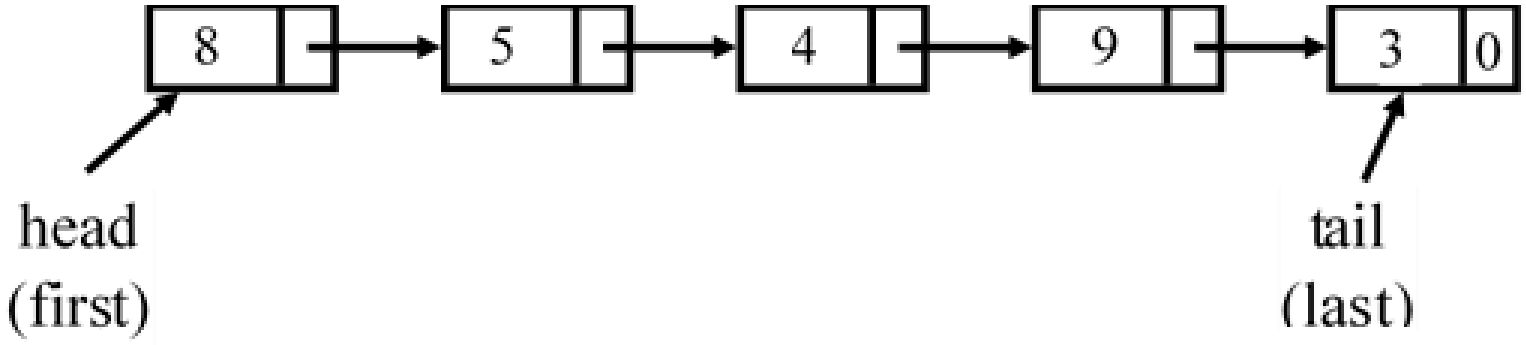


Ultimul nod a fost cel cu info 3 și s-a inserat nodul de info 6.

Coada – Implementare dinamică

Get – extragerea (ștergerea) unui nod din coada din figura C1

```
Queue q;
p<-q.head; // pointer spre primul nod ce urmează a fi șters
aux<-info(p); //valoarea ce va fi returnata
q.head<-succ(p); // noul head
free_sp(p); // dealocare memorie
return aux;
```



S-a extras elementul de info 2 și elementul de info 8 a devenit head.

Atentie la situatia cand coada are un singur element – trebuie actualizat si q.tail!

Coada – Implementare dinamică

- Put și Get modifică coada, Front nu (similar cu Push, Pop și Top de la stivă);
- Toate operațiile sunt de complexitate O(1) (similar stivei);
- Nu are sens să implementăm o funcție de "parcurgere" a cozii, nu este o operație specifică (similar stivei);
- Discuție privind prototipurile

```
Atom Get(Queue &q); Atom Front(Queue q);
```

- Cum tratăm cazul când coada este vidă și totuși se apelează una dintre aceste funcții?

```
Ex:  int Get(Queue &q,Atom &v);
```

Funcția întoarce un cod de eroare (1 dacă coada este vidă) iar v trebuie inițializată înainte de apel și valoarea se întoarce prin referință.

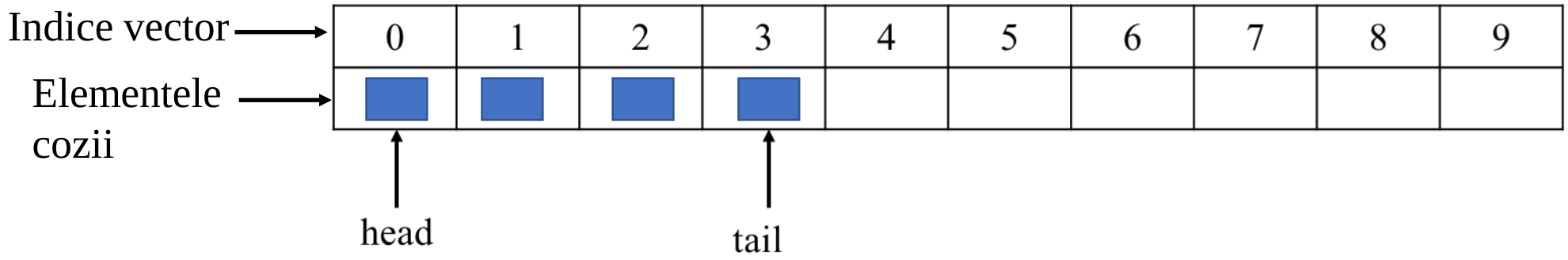
Alte soluții?

Coada – Implementare statică

O coadă poate fi implementată static pe un spațiu de memorare de tip tablou (vector).

Pentru a păstra timpul de execuţie constant O(1) atât pentru operaţia *Put* cât şi pentru *Get*, trebuie să nu mutăm elementele cozii în vector.

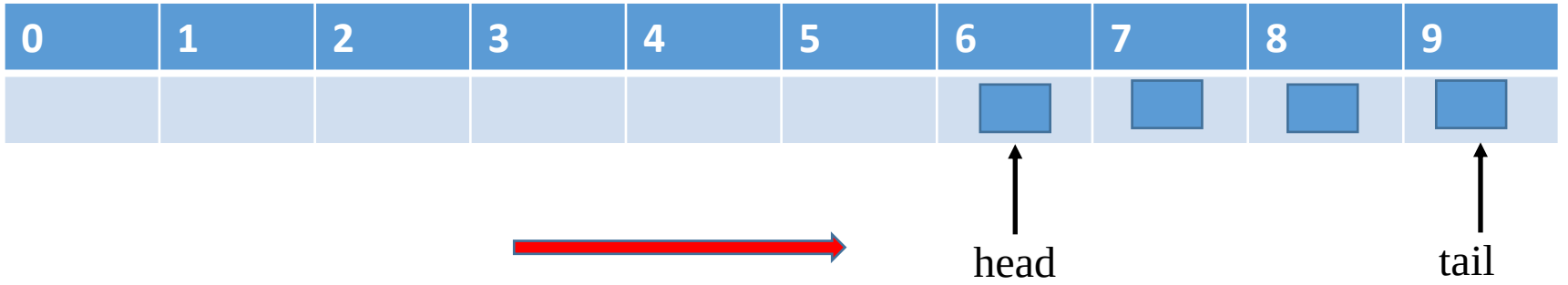
Se utilizează doi indici pe vector: unul pe începutul cozii (head) şi unul pe sfârşitul ei (tail).



Inserând și extrăgând elemente din coadă se poate ajunge la stuația următoare:

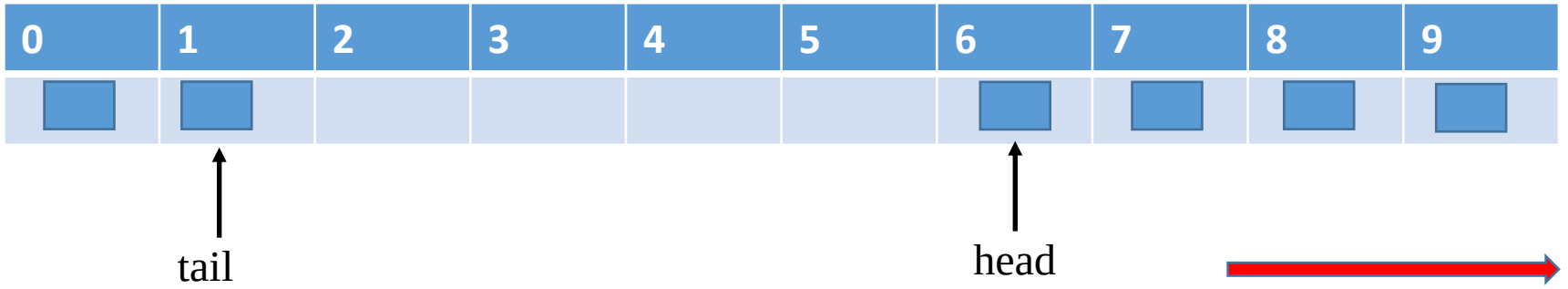
Coada – Implementare statică

Indicii se deplasează spre dreapta:



Deși mai sunt locuri libere în vector, indicele tail este pe poziția maximă și nu se mai pot insera elemente în coadă.

Soluția: să se gestioneze spatiul circular astfel încât tail să poată trece pe poziția 0.



Coada – Implementare statică

Totuși, pentru comoditatea implementării condițiilor de coadă plină, coadă goală și a trecerii celor doi indici de pe poziția maximă (9 în cazul nostru) pe prima poziție (0 în cazul unui vector), tail va indica prima poziție liberă în care se poate insera un element și NU poziția pe care s-a inserat ultimul element.

Astfel, o poziție rămâne neocupată când coada va fi plină!

Condiția de coada vidă: head ≡ tail

Condiția de coadă plină: head= următoarea valoare a lui tail după eventuală inserare

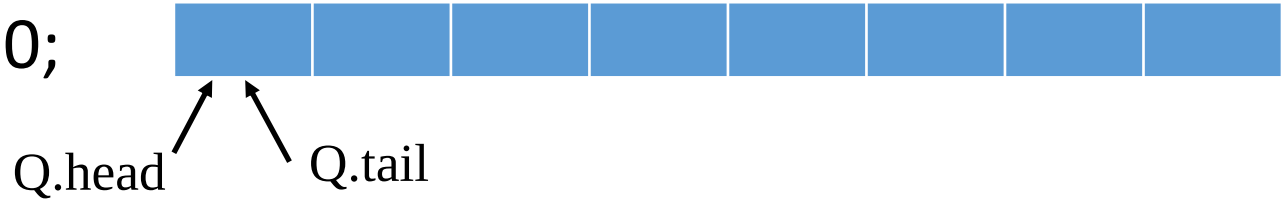
```
typedef  int Atom;
#define dimmax 10 //dimensiune vector
struct Queue
{
    int head, tail;
    Atom vect[dimmax];
};
```

```
int NextPos(int index)
{
    if (index<dimmax - 1)//altă soluție??
        return index + 1;
    else return 0;

}//index va fi Q.head sau Q.tail dupa caz
```

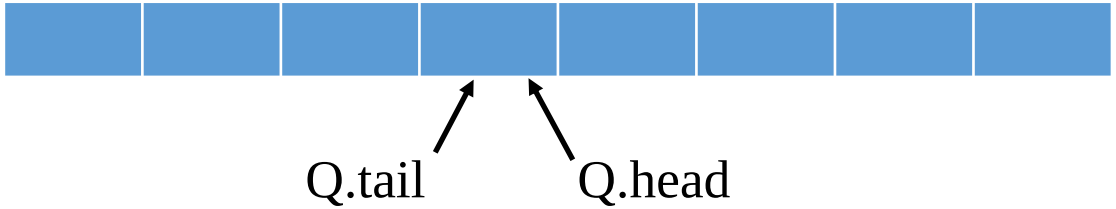
Coada – Implementare statică

```
Queue Q;
```

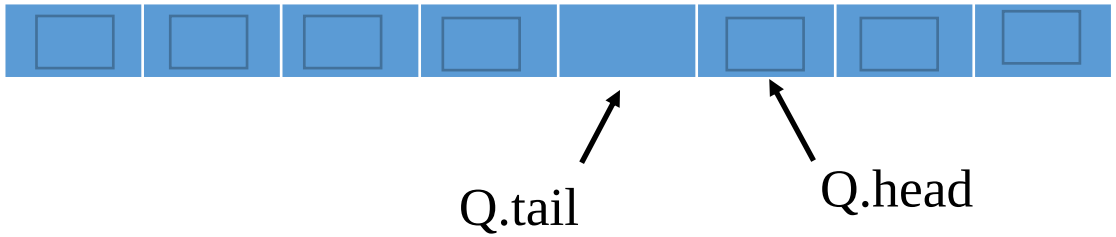
Inițializarea: Q.head = Q.tail = 0;



Coadă goală: se returnează rezultatul (Q.head == Q.tail)



Coadă plină: NextPos(Q.tail)==Q.head



Coada – Implementare statică

Prototipurile pentru coada alocată static

```
int IsEmpty(const queue &Q);
void InitQ(queue &Q);
void Put(queue &Q, Atom a);
Atom Get(queue &Q);
Atom Front(const queue &Q);
```

Link pentru simulare structuri de date:
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Stive și cozi – Implementări în Java și C++

La curs și laborator, au fost prezentate proprietățile și implementările standard pentru stive și cozi.

Ulterior, veți întâlni implementări ușor diferite în alte medii de programare.

Operații stivă

JAVA

1. **Object push(*Object element*)** : Pune un element în vârful stivei.
2. **Object pop()** : Extrage și returnează elemental din vârful stivei. Daca se apelează când stiva este goală aruncă exceptie de tipul ‘EmptyStackException’.
3. **Object peek()** : Returnează elemental din vârful stivei, dar nu-l șterge.
4. **boolean empty()** : returnează true dacă stiva este goală și false în caz contrar.
5. **int search(*Object element*)** : Determină dacă un element există în stivă și dacă da, returnează poziția acelui element. Dacă elementul nu se află în stivă intoarce -1.

Operatii stivă, C++

1. **empty** : returnează true dacă stiva este goală și false în caz contrar
2. **size**: returnează dimensiunea stivei
3. **top**: Accesează valoarea din vârful stivei fără să șteargă
4. **push**: Insert element
5. **pop**: Sterge elemental din varful stivei, fără sa-l returneze
6. **swap:** Swap contents

```
int main()
{
    // Empty stack
    stack<int> mystack;
    mystack.push(0);
    mystack.push(1);
    mystack.push(2);

    // Printing content of stack
    while (!mystack.empty()) {
        cout << ' ' << mystack.top(); //afisează ce este în vârful stivei
        mystack.pop(); //șterge ce este în vârful stivei
    }
}
```

```
//calculeaza suma elementelor unei
stive
sum=0
while (!mystack.empty()) {
        sum = sum + mystack.top();
        mystack.pop();
  }
```

Cozi - Java

- **add()-** is used to add elements at the tail of queue.
- **peek()-** is used to view the head of queue without removing it. It returns Null if the queue is empty.
- **element() –** it is similar to peek(). It throws *NoSuchElementException* when the queue is empty.
- **remove()-** removes and returns the head of the queue. It throws *NoSuchElementException* when the queue is empty.
- **poll()-** removes and returns the head of the queue. It returns null if the queue is empty.
- **size()-** return the no. of elements in the queue.

Cozi – C++

- **empty:** Test whether container is empty
- **size:** Return size
- **front:** Access next element
- **back:** Access last element
- **push:** Insert element
- **pop:** Remove next element
- **swap:** Swap contents