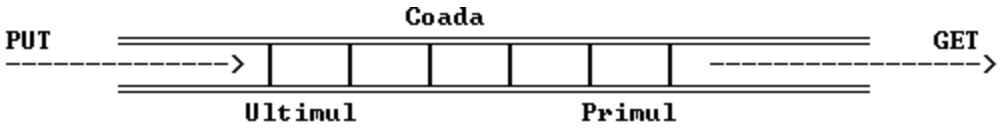
Cozi

1.1. Introducere

O ***coadă*** este o listă în care operațiile de acces sunt permise la inserarea la un capăt și ștergerea de la celălat capăt.



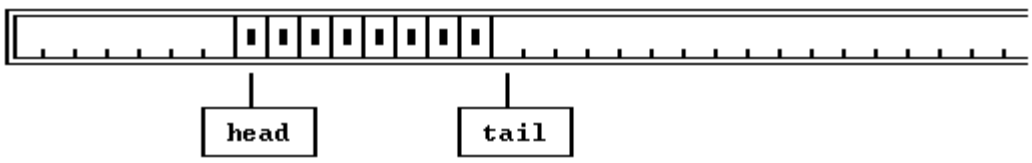
Pricipalele operații de acces sunt:

PUT(Coada, Atom) --> Coada
Adaugă un element la coadă.
GET(Coada) --> Coada, Atom
Extrage un Atom din coadă. Returnează atomul scos.

2.2. Implementari ale conceptului de coadă

[A] Coada ordonată liniară

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



Sunt necesari doi indicatori:
- ***head*** - indică poziția primului element care urmează să fie scos.
- ***tail*** - indică poziția unde va fi inserat următorul element adăugat la coadă.

Condiția "coada vida" este echivalentă cu: *head* ≡ *tail.*
Indicatorii vor fi inițializați astfel încât să indice ambii poziția primului element din vector.

Operatia **PUT** înseamnă:
- *V[tail]* primește Atomul adăugat
- incrementează *tail*

Operația **GET** înseamnă:
- întoarce *V[head]*
- incrementează *head*

Se observă că adaugări și stergeri repetate în coadă deplasează conținutul cozii la dreapta, față de începutul vectorului și astfel, primele poziții din coadă devin libere dar totuși inserarea nu este posibilă deoarece intexul *tail* este pe ultima poziție a tabloului.. Pentru a evita acest lucru ar trebui ca operația GET să deplaseze la stânga conținutul cozii cu o pozitie. Primul element care urmează să fie scos va fi întotdeauna în prima poziție, indicatorul *head* pierzându-și utilitatea. Dezavantajul acestei soluții constă în faptul că operatia GET necesita o parcurgere a conținutului cozii.

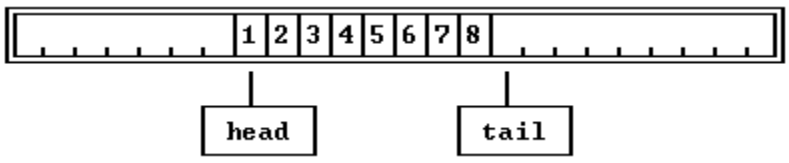
[B] Coada ordonată circulară

Pentru a obține o coadă ordonată circulară vom porni de la o coadă liniară ordonată simplă (cu doi indicatori) și vom face în așa fel încât la incrementarea indicatorilor *head* și *tail*, când aceștia ating ultima poziție din vector să se continue cu prima poziție din vector. Funcția următoare poate realizeaza această cerința:

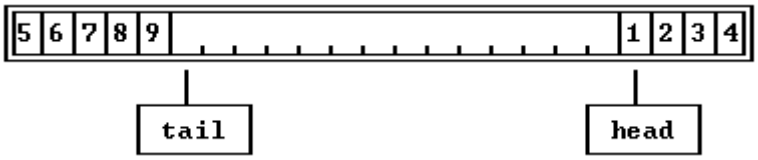
```
int nextPoz(int index)
{
        if (index<DIMVECTOR-1) return index+1;
        else return 0;
}
```

unde DIMVECTOR este dimensiunea vectorului în care se memorează elementele cozii.

Conținutul cozii va arăta astfel:

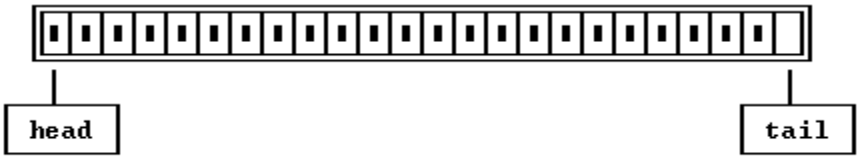


sau:

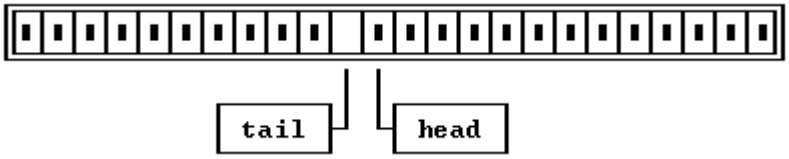


■ Condiția "coada vida" rămâne echivalentă cu: *head* ≡ *tail*

■ Coada va fi plină dacă: *head=1* și *tail=DIMVECTOR*



sau dacă: *tail+1* ≡ *head*

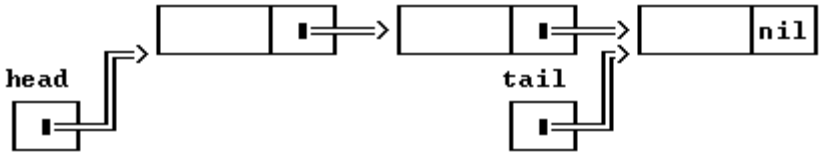


Ambele situații sunt conținute în condiția:

nextPoz(tail) = head // condiția "coada plina"

[C] Coada înlănțuită liniară

O coadă poate fi implementată printr-o listă liniară simplu înlănțuită la care operațiile de acces sunt permise doar la capete.



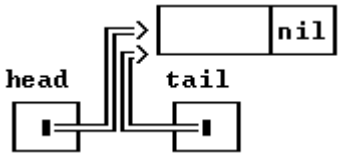
Este nevoie de doi indicatori (pointeri):
head - indică primul element din coadă (primul care va fi scos);
tail - indică ultimul element din coadă (ultimul introdus).

O coadă vidp va avea ***head=tail=nil***

In mod obișnuit, adăugarea unui element în coadă modifică numai *tail* iar ștergerea unui element numai *head*. Intr-un mod special trebuie să fie tratate cazurile:

Inserare într-o coadă vidă:

Initial: head=tail=nil
Final: Coadă cu un element:



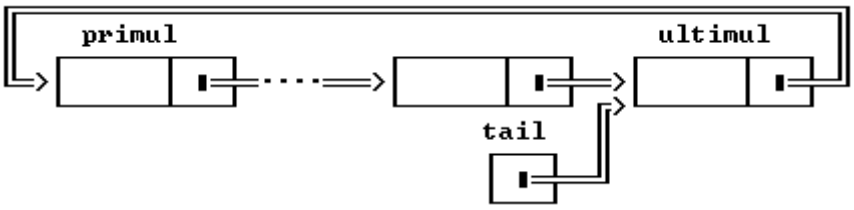
Ștergere dintr-o coadă cu un element:

Initial: Coadă cu un element
Final: head=tail=nil

In aceste cazuri se modifică atât *head* cât și *tail*.

[D] Coada înlănțuită circulară

Daca reprezentăm coada printr-o structură înlănțuită circulară va fi nevoie de un singur pointer prin intermediul căruia se pot face ambele operații de adăugare și ștergere din coadă:



TEMA 1
Realizați un proiect pentru implementarea operațiilor aferente unei cozi circulare alocată static. Testați implementarea realizată. Conținut header:

```
#define DIMMAX 8
typedef int Atom;
struct Queue {
      int head,tail;
      Atom vect[DIMMAX];
      };
void initQueue(Queue& Q);
void put(Queue& Q, Atom a);
Atom get(Queue& Q);
Atom front(Queue& Q);      // coada este pasata prin referinta din motive
int isEmpty(Queue& Q);     // de eficienta
```

Atenție: nu utilizați o funcție pentru ”afișarea” cozii – această operație nu este specifică structurii de tip coadă (nu are sens ”afișarea/parcurgerea” cozii)

TEMA 2

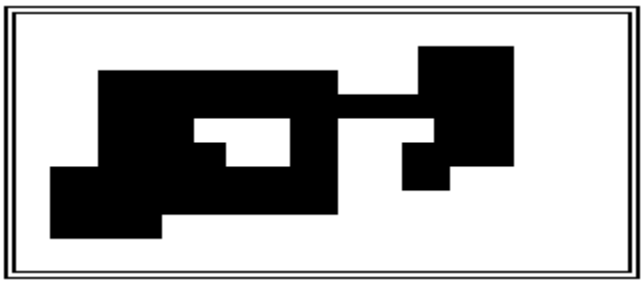
Realizați un proiect pentru implementarea operațiilor aferente unei cozi înlănțuite (una din variante). Testați implementarea realizată. Conținut header:

```
struct Element{
      Atom data;
      Element* succ;
      };
struct Queue {
      Element* tail;//coada inlantuiata circular
      };

void initQueue(Queue& Q);
void put(Queue& Q, Atom a);
Atom get(Queue& Q);
Atom front(Queue& Q);      // coada este pasata prin referinta din motive
int isEmpty(Queue& Q);     // de eficienta
```

TEMA 3

Definim "domeniu" ca fiind o zonă din ecranul grafic care conține pixeli învecinați direct sau indirect, pe verticală sau orizontală și care au aceiași culoare. De exemplu:



Mai sus sunt definite trei domenii:

-cel colorat cu negru;
-cel colorat cu alb, aflat în exterior;
-cel colorat cu alb, în interiorul zonei ngre (insula).

Să se scrie un program care schimbă culoarea unui domeniu, pornind de la un pixel oarecare din acel domeniu. Completați aplicația C++ furnizată împreună cu laboratorul.

Schița metodei de rezolvare propuse

Descrierea metodei:

Se folosește o coadă în care se plasează coordonate ale unor pixeli.

```
struct Pozitie {
        int x,y;
        };

typedef Pozitie Atom;       // elementele cozii sunt de tip Pozitie
```

Se colorează și se pune în coadă mai întâi poziția pixelului inițial. Vom prelucra pe rând elementele scoase din coadă. Pentru fiecare element scos din coadă se pun în coadă și se coloreaza toți vecinii săi care aparțin domeniului și nu au fost atinși (nu au fost colorați). Astfel vor trece prin coadă toate pozițiile incluse în domeniu.

Pseudocod:

```
coloreaza_domeniu(pozitie_initiala)
{
 memoreaza culoarea domeniului (culoarea pozitiei initiale)
 put(C, pozitie_initiala);
 coloreaza pozitia_initiala;
 WHILE not isEmpty(C) DO
        ┌──────
        │p := get(C);    // scoate în p urmatoarea pozitie din coada
        │FOR pi:=toate pozitiile invecinate lui p DO
        │   IF pi nu iese în afara ecranului si
        │      pi are culoarea domeniului THEN
        │        ┌──────
        │        │put(C, pi);
        │        │coloreaza pi;
        │        └──────
        └──────
}
```

Observație:

Să încercăm să punem în evidență, pentru cazul nostru, "vocația" cozii: adaptarea între un producător și consumator nesincronizați. În coadă folosită în această problemă consumatorul este ritmic (la fiecare iterație se scoate din coadă și se prelucrează o poziție). Producătorul, datorită plasarii în coadă a pozițiilor învecinate, este aritmic (tratarea unei poziții pune în coadă mai mulți - cel mult 4 – sau niciun vecin).

NOTARE

Tema 1: 5p
Tema 2: 3p
Tema 3: 2p