Stive. Cozi

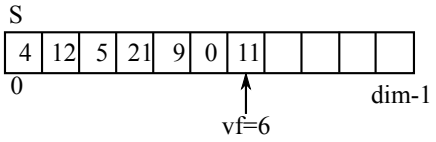
Universitatea "Transilvania" din Brașov

7 martie 2022

Stiva

Definiție: Stiva este o structură liniară de tip **LIFO** - *Last In First Out*, adică ultimul element introdus va fi primul care se extrage pentru prelucrare. Accesul la elementele stivei se realizează doar prin vârful stivei, unde se află ultimul element introdus.

Stiva - reprezentarea în memorie



Reprezentarea secvențială cu un tablou unidimensional - *array*.
Utilizăm o structura stiva care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- o variabilă *vf* ce reprezentă vârful stivei = poziția în vector pe care se află ultimul element aparținând stivei
- Dacă $vf = -1$ atunci stiva este vidă și nu pot extrage elemente
- Dacă $vf = dim - 1$ atunci stiva este plină și nu pot adăuga elemente noi. Este nevoie de realocare de memorie.

Stiva - adăugarea unui element

Algoritm: PUSH

Intrare: Stiva S în care adaug și elementul *val* care se adaugă

daca $S.vf = S.dim - 1$ **atunci**

    realocare de memorie

sfarsit_daca

$S.vf \leftarrow S.vf + 1$

$S.data[S.vf] \leftarrow val$

Stiva - eliminarea unui element

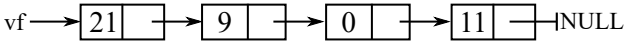
Algoritm: POP

Intrare: Stiva S din care elimin elementul din vârf
daca $S.vf$ = *-1* **atunci**
| scrie("Stiva este vida")
sfarsit_daca
altfel
| $S.vf \leftarrow S.vf - 1$
sfarsit_daca

Stiva - reprezentarea în memorie

Reprezentarea înlănțuită cu o listă înlănțuită - *array*.

1. Pentru fiecare element utilizăm o structură NOD cu câmpurile:
   - info - conține informația de interes
   - urm - pointer care conține adresa elementului precedent din stivă
2. Pentru stivă utilizăm o structura STIVA cu câmpul VARF - pointer care indică adresa elementului din vârful stivei

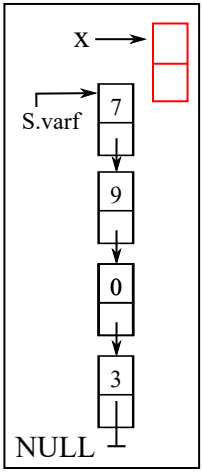


Observație: când stiva este vidă VARF este NULL

Structura Stiva în C++

```
#include<iostream>

struct stiva {
    struct nod {
        int info;
        nod* urm;
    };
    nod* varf;
    //functii pentru stiva
};
```

Stiva - adăugarea unui element



Algoritm: Adăugarea unui element pe stivă

Functie *PUSH(S, val)*

Intrare: Stiva S în care adaug și *val* care se pune în varful stivei

alocă memorie pentru nodul x

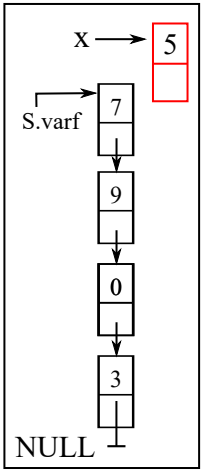
$x.info \leftarrow val$

$x.urm \leftarrow S.varf$

$S.varf \leftarrow x$

end

Stiva - adăugarea unui element



Algoritm: Adăugarea unui element pe stivă

Functie *PUSH(S, val)*

Intrare: Stiva S în care adaug și *val* care se pune în varful stivei

alocă memorie pentru nodul x

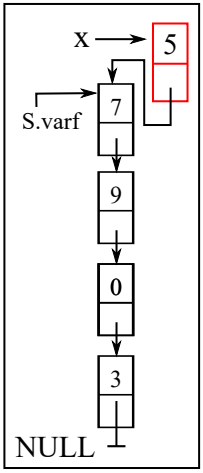
$x.info \leftarrow val$

$x.urm \leftarrow S.varf$

$S.varf \leftarrow x$

end

Stiva - adăugarea unui element



Algoritm: Adăugarea unui element pe stivă

Functie *PUSH(S, val)*

Intrare: Stiva S în care adaug și *val* care se pune în varful stivei

alocă memorie pentru nodul x

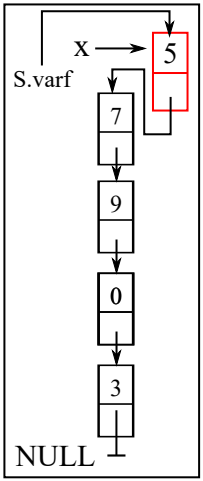
$x.info \leftarrow val$

$x.urm \leftarrow S.varf$

$S.varf \leftarrow x$

end

Stiva - adăugarea unui element



Algoritm: Adăugarea unui element pe stivă

Functie *PUSH(S, val)*

Intrare: Stiva S în care adaug și *val* care se pune în varful stivei

alocă memorie pentru nodul x

$x.info \leftarrow val$

$x.urm \leftarrow S.varf$

$S.varf \leftarrow x$

end

Stiva - extragerea elementului din vârful stivei

Algoritm: Extragerea elementului din vârful stivei

Functie *POP(S)*
  Intrare: Stiva S din care se sterge elementul din vârf
  daca $varf \neq NULL$ **atunci**
    $y \leftarrow varf$
    $varf \leftarrow varf.urm$
    elibereaza memoria pentru y
  sfarsit_daca
end

Structura Stiva în C++

```
#include<iostream>

struct stiva {
    struct nod {
        int info;
        nod* urm;
    };
    nod* varf = nullptr;
    void push(int val)
    {
        nod* nou = new nod;
        nou->info = val;
        nou->urm = varf;
        varf = nou;
    }
};
```

```
int main()
{
    stiva S;
    int nrElem, elem;
    std::cin >> nrElem;
    for (int index = 0; index < nrElem; index++)
    {
        std::cin >> elem;
        S.push(elem);
    }
    return 0;
}
```

Stiva

Complexitate: atât PUSH cât și POP au complexitate constantă!

Stiva - Exemplu

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (((() nu este o parantezare corectă,)()(nu este o parantezare corectă, (((()))) este o parantezare corectă.

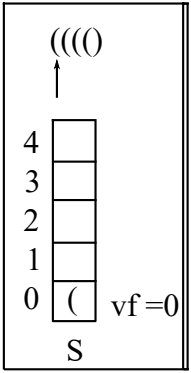
Stiva - Exemplu

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (((() nu este o parantezare corectă,)()(nu este o parantezare corectă, ((()())) este o parantezare corectă.
Rezolvare: Se utilizează o stiva S

- când se găsește în șir o paranteză deschisă, se pune pe stivă.
- când se găsește o paranteză închisă, se scoate paranteza deschisă din vf stivei.
- dacă la găsirea unei paranteze închise stiva este goală $\Rightarrow$ eroare.
- dacă după parcurgerea șirului de paranteze stiva nu s-a golit $\Rightarrow$ eroare.

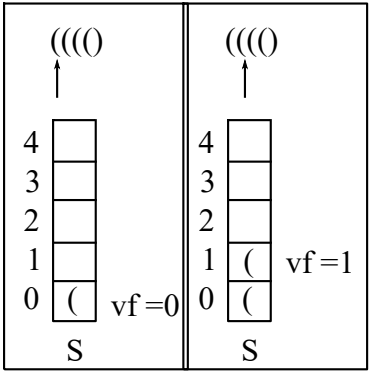
Stiva - Problema parantezării - Exemplul 1

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: ((((



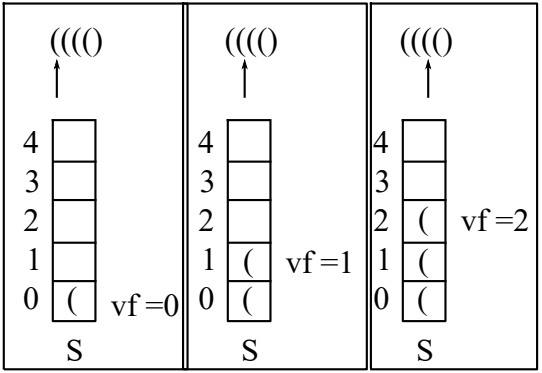
Stiva - Problema parantezării - Exemplul 1

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: ((((



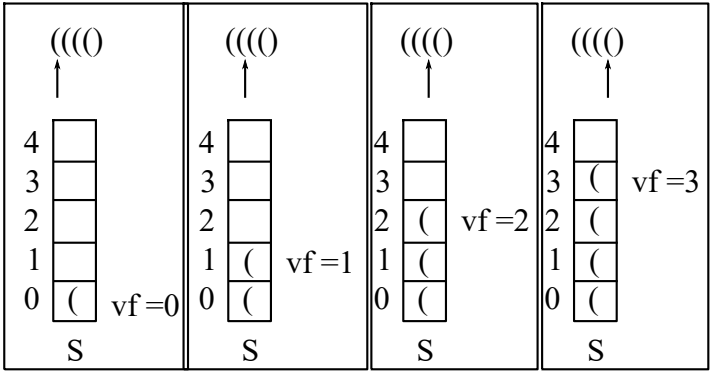
Stiva - Problema parantezării - Exemplul 1

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: ((((



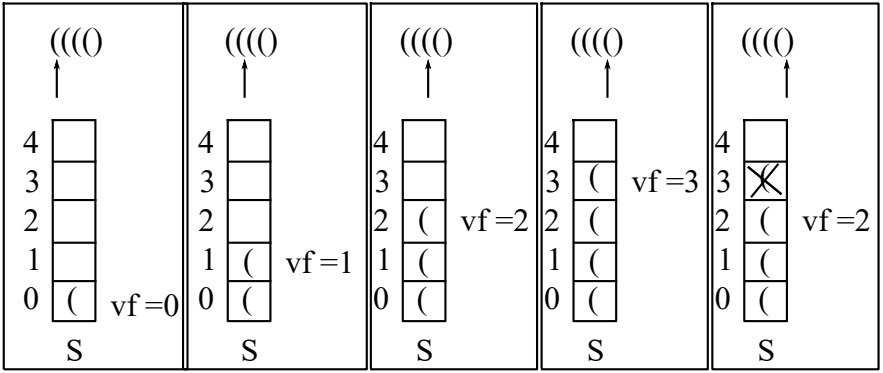
Stiva - Problema parantezării - Exemplul 1

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: ((((



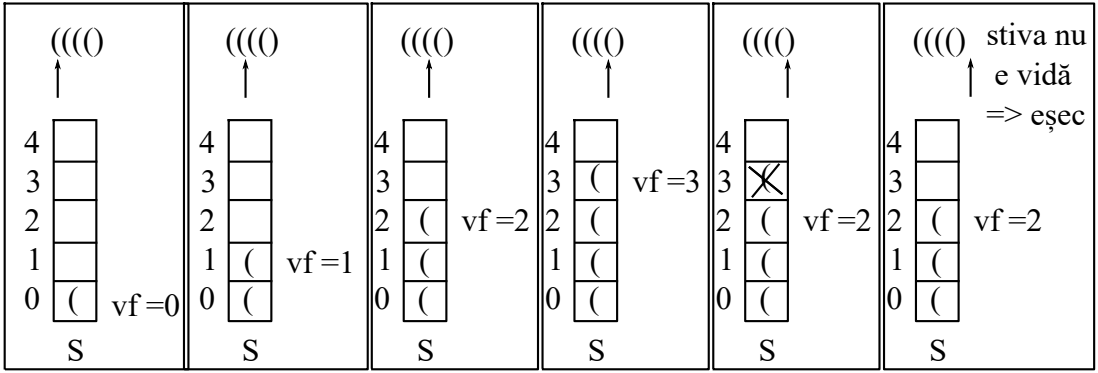
Stiva - Problema parantezării - Exemplul 1

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: ((((



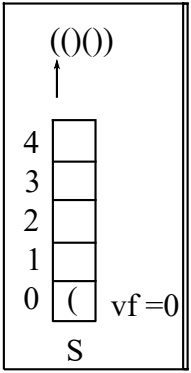
Stiva - Problema parantezării - Exemplul 1

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: ((((



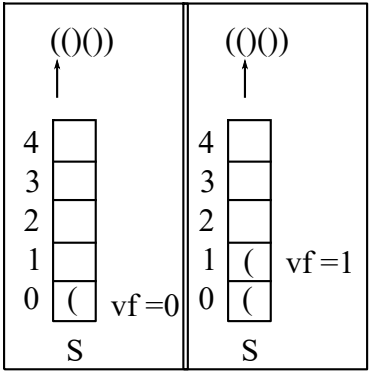
Stiva - Problema parantezării - Exemplul 2

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (()())



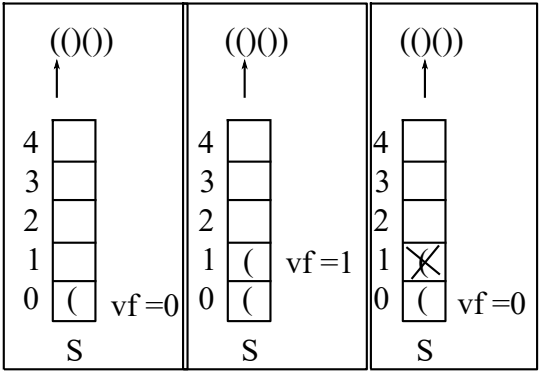
Stiva - Problema parantezării - Exemplul 2

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (()())



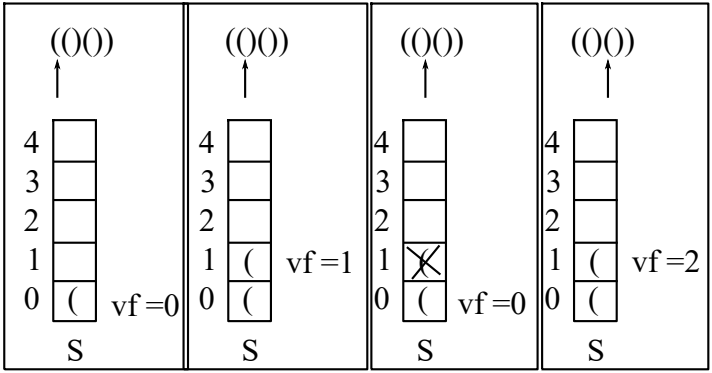
Stiva - Problema parantezării - Exemplul 2

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (()())



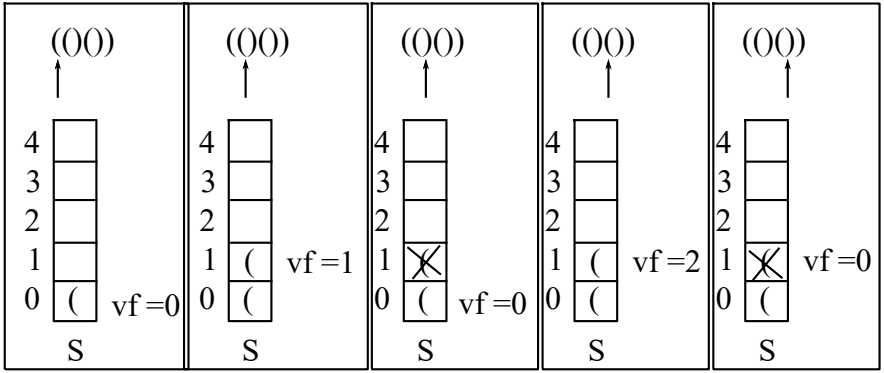
Stiva - Problema parantezării - Exemplul 2

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (()())



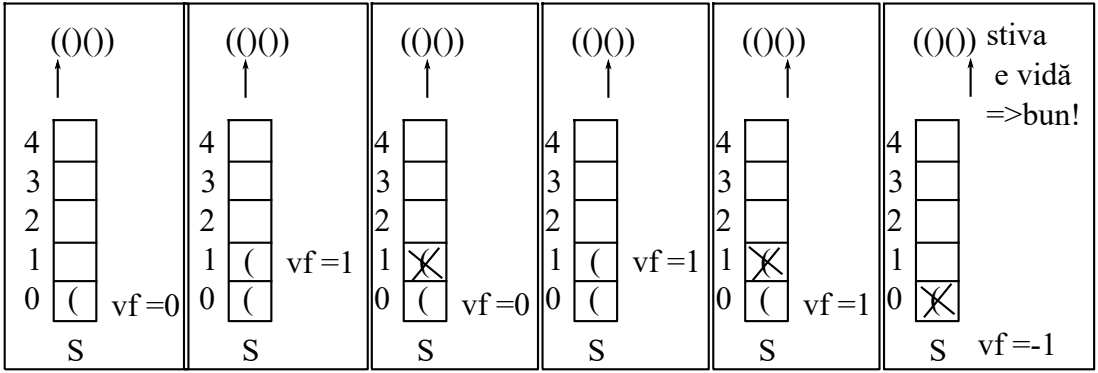
Stiva - Problema parantezării - Exemplul 2

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (()())



Stiva - Problema parantezării - Exemplul 2

Problemă: se dă un șir de paranteze deschise și închise. Să se verifice dacă este o parantezare corectă. De exemplu: (()())

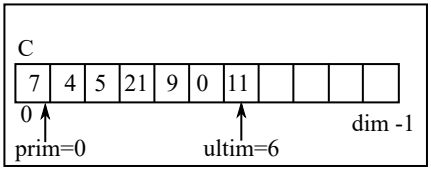


Coada

Definiție: Coada este o structură liniară de tip **FIFO** - *First In First Out*, adică primul element introdus va fi primul care se extrage pentru prelucrare.

Coada modelează procese care presupun formarea de cozi, de exemplu deservirea clienților la un ghișeu. De asemenea se utilizează cozi pentru operații precum parcurgerea în lățime a unui graf sau a unui arbore.

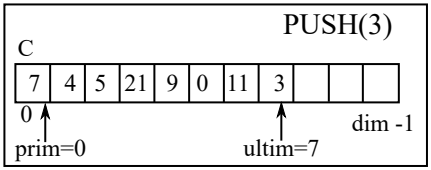
Coada - reprezentarea în memorie



Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

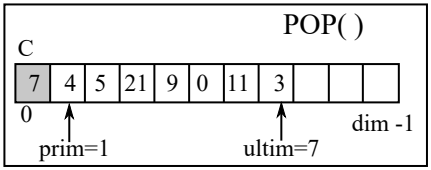
Coada - reprezentarea în memorie



Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

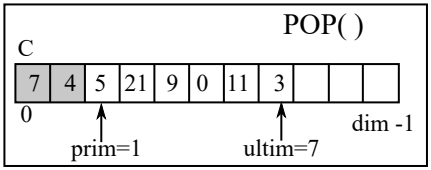
Coada - reprezentarea în memorie



Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

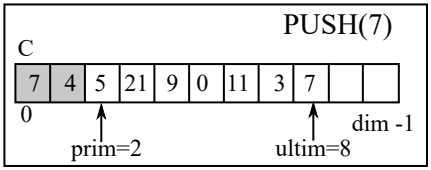
Coada - reprezentarea în memorie



Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

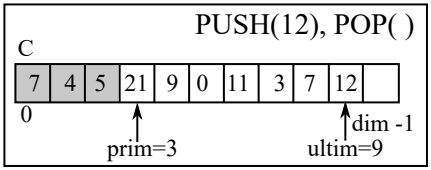
Coada - reprezentarea în memorie



Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

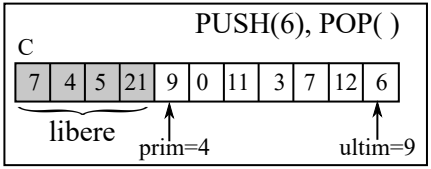
Coada - reprezentarea în memorie



Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

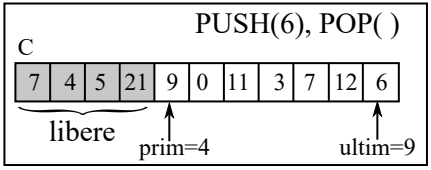
Coada - reprezentarea în memorie



Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

Coada - reprezentarea în memorie



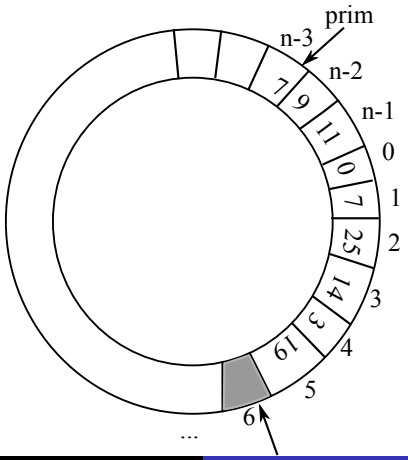
PROBLEMA: coada e considerată plină, deși avem 4 poziții neocupate!

Reprezentarea secvențială - cu un tablou unidimensional - *array*.
Utilizăm o structura *coada* care dispune de următoarele câmpuri

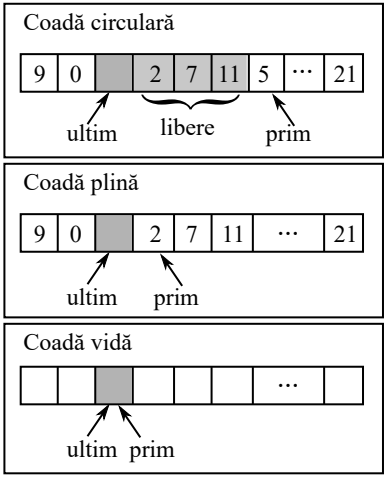
- un vector *data* de dimensiune *dim* = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* și *ultim* care reprezentă poziția în vector pe care se află primul, respectiv ultimul element aparținând cozii.

Coada

Observație: la fiecare extragere și adăugare, coada migrează înspre dreapta! Astfel se poate ajunge mesaj de coadă plină, deși mai sunt locuri neocupate la început. Acest lucru se evită utilizând cozi circulare!



Coada circulară



Observație: - variabila *ultim* indică prima poziție liberă după ultimul element din coadă.

Condiții:

- coada plină: $(ultim + 1)$ mod $dim = prim$
- coada vidă: $ultim = prim$

deque

deque - *double-ended queue*

- permite adăugare și extragere la ambele capete în complexitate $O(1)$

deque

deque - *double-ended queue*

- permite adăugare și extragere la ambele capete în complexitate $O(1)$
- permite acces la orice element prin poziție în complexitate $O(1)$

deque

deque - *double-ended queue*

- permite adăugare și extragere la ambele capete în complexitate $O(1)$
- permite acces la orice element prin poziție în complexitate $O(1)$
- necesită pentru inserție sau ștergerea unui element din interior cel mult $N/2$ operații

deque

deque - *double-ended queue*

- permite adăugare și extragere la ambele capete în complexitate $O(1)$
- permite acces la orice element prin poziție în complexitate $O(1)$
- necesită pentru inserție sau ștergerea unui element din interior cel mult $N/2$ operații
- memoria necesară este alocată / dealocată in funcție de necesitate

deque

deque - *double-ended queue*

- permite adăugare și extragere la ambele capete în complexitate $O(1)$
- permite acces la orice element prin poziție în complexitate $O(1)$
- necesită pentru inserție sau ștergerea unui element din interior cel mult $N/2$ operații
- memoria necesară este alocată / dealocată in funcție de necesitate
- prin adăugare / ștergere la oricare dintre capete nu sunt invalidate referințele către elemente neafectate de aceste operații

deque

deque - implementări

- *array* circular - asemănător cu coada circulară prezentată anterior - necesită realocare, atunci când memoria alocată se umple.

deque

deque - implementări

- *array* circular - asemănător cu coada circulară prezentată anterior - necesită realocare, atunci când memoria alocată se umple.
- *array* în care se începe introducerea elementelor din centru

deque

deque - implementări

- *array* circular - asemănător cu coada circulară prezentată anterior - necesită realocare, atunci când memoria alocată se umple.
- *array* în care se începe introducerea elementelor din centru
  - se realocă atunci când se ajunge la unul din capete.

deque

deque - implementări

- *array* circular - asemănător cu coada circulară prezentată anterior - necesită realocare, atunci când memoria alocată se umple.
- *array* în care se începe introducerea elementelor din centru
  - se realocă atunci când se ajunge la unul din capete.
  - necesită realocări mai frecvente decât varianta circulară

deque

deque - implementări

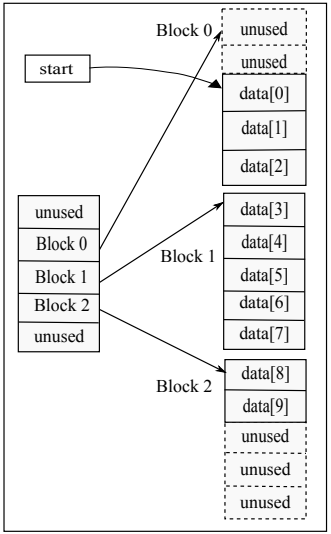
- *array* circular - asemănător cu coada circulară prezentată anterior - necesită realocare, atunci când memoria alocată se umple.
- *array* în care se începe introducerea elementelor din centru
  - se realocă atunci când se ajunge la unul din capete.
  - necesită realocări mai frecvente decât varianta circulară
  - poate apărea mai mult spațiu nefolosit, mai ales dacă adăugările se fac preponderent la unul dintre capete

deque

deque - implementări

- *array* circular - asemănător cu coada circulară prezentată anterior - necesită realocare, atunci când memoria alocată se umple.
- *array* în care se începe introducerea elementelor din centru
  - se realocă atunci când se ajunge la unul din capete.
  - necesită realocări mai frecvente decât varianta circulară
  - poate apărea mai mult spațiu nefolosit, mai ales dacă adăugările se fac preponderent la unul dintre capete
- *array* de *array* -uri - metoda folosită în implementarea STL

STL - *deque*



- blocuri de dimensiune constantă
- **push_back()** adaugă un element la ultimul bloc sau se alocă un nou bloc în care se inserează; similar pentru **push_front()**
- funcția **insert** deplasează elementele de la punctul de inserție înspre cel mai apropiat capăt.
- container utilizat pentru implementarea structurilor **queue** - coadă - și **stack** - stivă.

STL - *deque*

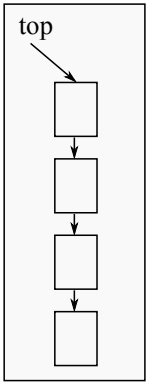
```
#include<iostream>
#include<deque>
#include<vector>

void RightPermut(std::deque<int>& Number, int k)
{
    for (int i = 0; i < k; i++)
    {
        Number.push_back(Number.front());
        Number.pop_front();
    }
}
void LeftPermut(std::deque<int>& Number, int k)
{
    for (int i = 0; i < k; i++)
    {
        Number.push_front(Number.back());
        Number.pop_back();
    }
}
```

Adaptori de containere

- tipuri de date din STL care adptează containere pentru o interfață specifică
- ex: coadă, stivă, coadă de priorități

Adaptori de containere - stack

```
template <class T, class Container = deque<T> > class stack;
```



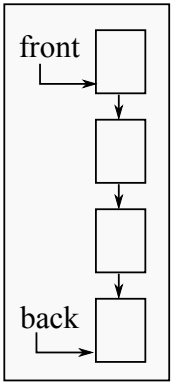
- **empty** - testează dacă stiva este vidă
- **size** - determină câte elemente sunt în stivă
- **top** - returnează elementul din vârf
- **push** - pune un element în stivă
- **pop** - elimină elementul din vârful stivei.

Stack - exemplu

```
#include<iostream>
#include<stack>

bool checkParanthesis(std::string paranthesis)
{
    std::stack<char> parant;
    for (char c : paranthesis)
    {
        if (c == '(')
            parant.push(c);
        else
            if (c == ')')
            {
                if (parant.empty())
                    return false;
                parant.pop();
            }
    }
    if (!parant.empty())
        return false;
    return true;
}
```

Adaptori de containere - coada

```
template <class T, class Container = deque<T> > class queue;
```



- **empty** - testează dacă coada este vidă
- **size** - determină câte elemente sunt în coadă
- **front** și **back**
- **push** - pune un element în coadă
- **pop** - elimină elementul din coadă.