

BIBLIOGRAFIE (partea de Structuri de date):

1. I. Ignat, C.L. Ignat, *Structuri de date și algoritmi*, Editura Albastră, Cluj-Napoca, 2007.
2. O.Bâscă, M.Jaică (Miroiu), *Structuri de date (note de curs)*, Editura Universității din Pitești, 2000.
3. M.D Zaharia, *Structuri de date și algoritmi – algoritmi, exemple în limbajele C și C++*, Editura Albastră, Cluj-Napoca, 2002.
4. I. Tomescu, *Data Structures*, Editura Universității din București, 1997.
5. Iorga, V., Chiriță, P., Stratan, C., Opincaru, C., *Programare în C/C++*. *Culegere de probleme*. Editura Niculescu, București, 2003.
6. T.C. Ionescu, I. Zsako, *Structuri arborescente cu aplicațiile lor*, Editura Tehnică, București, 1990.
7. D.Burdescu, *Structuri de date arborescente*, vol. 1 și 2, Editura Mirton, Timișoara, 1993.
8. T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introducere în algoritmi*, Editura Libris Agora, 2001 (traducere).
9. Leon Livovshi, Horia Georgescu, *Sinteza și analiza algoritmilor*, Editura Științifică și enciclopedică, 1986.
10. R. Andonie, I. Gabarcea, *Algoritmi fundamentali. O perspectiva C++*, Editura Libris, 1995.
11. Octavian Pătrășcoiu, Gheorghe Marian, Nicolae Mitroi, *Elemente de grafuri, combinatorică, metode, algoritmi și programe*, Editura All, București, 1994.
12. Viorel Păun, *Algoritmica și programarea calculatoarelor. Limbajul C++*. Editura Universității din Pitești, 2003.
13. B. Pătruț, *Aplicații în C și C++*, Editura Teora, 2000.

STRUCTURI DE DATE

LISTE LINIARE

O clasă particulară de structuri de date o constituie listele liniare. O **listă liniară** este o înșiruire de **elemente de același tip** în care se distinge un **prim element** și un **ultim element**, nu neapărat distincte, fiecare element în afară de primul având un element care îl **precede** și fiecare element în afară de ultimul având unul care îl **succede**. O listă care nu conține nici un element se numește **listă vidă**.

Asupra listelor se pot efectua mai multe tipuri de operații dintre care cele mai frecvente sunt:

- accesul la un anumit element pentru prelucrarea lui (examinare, modificare etc.);
- inserarea unui nou element într-o poziție dată (înainte de primul element, după ultimul element, înainte sau după un element dat);
- eliminarea unui element din listă;
- combinarea a două sau mai multe liste în una singură;
- despărțirea unei liste în două sau mai multe liste;
- copierea unei liste;
- determinarea numărului de elemente ale unei liste;
- sortarea elementelor după anumite criterii;
- căutarea unor elemente cu anumite proprietăți.

Frecvența cu care sunt efectuate diferitele operații asupra listelor determină modul de reprezentare a lor pentru a obține cea mai bună eficiență din punct de vedere al spațiului ocupat și al timpului de calcul. Metodele de reprezentare se împart în două categorii: **reprezentarea secvențială** (statică) și **reprezentarea înlănțuită** (dinamică).

Implementarea secvențială a listelor liniare

Unul dintre cele mai simple moduri de reprezentare în memoria calculatorului a listelor liniare este modul secvențial în care elementele listei sunt memorate unul după altul în ordinea în care apar ele în listă, în locații consecutive (într-un vector). Notând cu $&a$ locația unde se memorează elementul a din listă, se deduce relația $&x_{i+1} = &x_i + c$, unde c este numărul de locații ocupate de un element. Deci $&x_i = l_0 + c \cdot (i-1)$, unde l_0 este adresa de început a primului element al listei, numită *adresa de început a listei*.

Avantajul metodei constă în accesul la oricare element printr-o operație simplă și rapidă, cu durata de execuție constantă, indiferent de poziția elementului în listă.

În continuare se descriu **operațiile elementare** efectuate asupra unei liste liniare reprezentată secvențial. Vom memora elementele listei într-un vector global x în poziții consecutive, ordinea fizică a lor fiind aceeași cu ordinea lor logică. Pentru a ști câte elemente există efectiv în listă se definește o variabilă globală N . În C, *structura listei* se poate defini astfel:

```
#define NMAX 50
int N;           // numărul efectiv de elemente din listă (maxim NMAX)
int x[NMAX];     // elementele listei sunt presupuse întregi
```

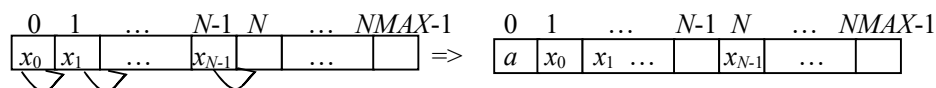
0	1	...	N-1	N	...	NMAX-1
x_0	x_1	...	x_{N-1}		...	

Inițializarea listei ca o listă vidă se face punând 0 numărul elementelor listei ($N=0$).

Putem vedea dacă **lista este vidă** dacă numărul efectiv al elementelor listei este 0 ($N==0$).

Putem vedea dacă **lista este plină** (s-a ocupat tot spațiul rezervat) dacă numărul efectiv al elementelor listei este chiar NMAX ($N==NMAX$).

Implementarea operației de inserare a unui nou element a la început (pe poziția 0) se face astfel: toate elementele listei, de la **dreapta la stânga**, adică de la ultimul element la primul, se **deplasează cu o poziție spre dreapta**, iar pe prima poziție, "rămasă acum liberă", se inserează noul element.



Adică: $x_N \leftarrow x_{N-1}$

...

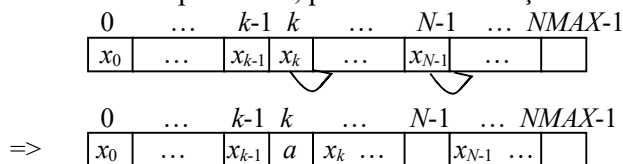
$x_2 \leftarrow x_1$

$x_1 \leftarrow x_0$

$\Leftrightarrow x_i \leftarrow x_{i-1}, i = N, 1, -1$ sau $x_{i+1} \leftarrow x_i, i = N-1, 0, -1$.

```
void InserareInceput(int x[NMAX], int &N, int a){
    if (N==NMAX)
        cout << "Nu mai este spatiu pentru un nou element"; //overflow
    else {
        for (int i = N++; i > 0; i--) x[i] = x[i-1];
        x[0] = a;
    }
}
```

Inserarea unui element a în interiorul listei, pe o poziție k , se face prin deplasarea elementelor de la ultimul până la al k -lea cu o poziție spre dreapta și inserarea noului element pe locul rămas liber. În particular, pentru $k = 0$ se obține cazul precedent.



Adică: $x_N \leftarrow x_{N-1}$
 \vdots
 $x_{k+1} \leftarrow x_k$

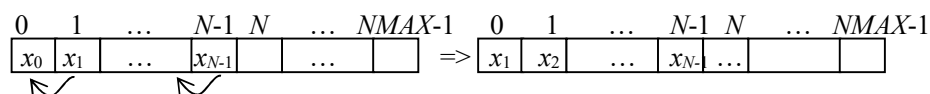
$\Leftrightarrow x_i \leftarrow x_{i-1}, i = N, k+1, -1$ sau $x_{i+1} \leftarrow x_i, i = N-1, k, -1$.

```
void InserareInterior(int x[NMAX], int &N, int a, int k) {
    if (N==NMAX)
        cout << "Nu mai este spatiu pentru inserare"; //overflow
    else
        if ((k>=0) && (k<=N-1)) {
            for (int i = N++; i > k; i--) x[i] = x[i-1];
            x[k] = a;
        }
}
```

Inserarea unui nou element a la sfârșitul listei, adică după ultimul element din listă, nu mai necesită deplasări de elemente.

```
void InserareSfarsit(int x[NMAX], int &N, int a) {
    if (N==NMAX) cout << "Nu mai este spatiu pentru inserare";
    else x[N++] = a; //sau x[N]=a; N++;
}
```

Eliminarea primului element se face prin deplasarea elementelor de la cel de-al doilea până la ultimul cu o poziție spre stânga.

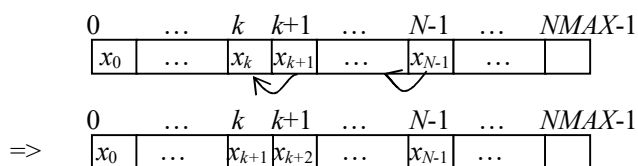


Adică: $x_0 \leftarrow x_1$
 \vdots
 $x_{N-2} \leftarrow x_{N-1}$

$\Leftrightarrow x_i \leftarrow x_{i+1}, i = 0, N-2$ sau $x_{i-1} \leftarrow x_i, i = 1, N-1$.

```
int StergereInceput(int x[NMAX], int &N, int &a)
{ if (N==0) {
    cout << "Nu sunt elemente de sters"; //underflow
    return 0; //operatie incheiata cu esec
  }
  a = x[0]; // se reține valoarea primului element
  for (int i = 0; i < N-1; i++) x[i] = x[i+1];
  N--; // numărul efectiv de elemente din listă scade cu o unitate
  return 1; //operatie incheiata cu succes
}
```

Eliminarea elementului de pe poziția k se face prin deplasarea elementelor de la poziția $k+1$ până la ultima cu o poziție spre stânga.



Adică: $x_k \leftarrow x_{k+1}$

...

$x_{N-2} \leftarrow x_{N-1}$

$\Leftrightarrow x_i \leftarrow x_{i+1}, i = k, N-2$ sau $x_{i-1} \leftarrow x_i, i = k+1, N-1$.

```
int StergereInterior(int x[NMAX], int &N, int k, int &a) {
    if (N==0) {
        cout << "Nu sunt elemente de sters"; //underflow
        return 0; //operatie incheiata cu esec
    }
    if ((k<0) || (k>=N)) {
        cout<<"Nu exista elementul de rang " << k;
        return 0; //operatie incheiata cu esec
    }
    else {
        a = x[k]; // se reține valoarea elementului de șters
        for(int i = k; i < N-1; i++) x[i] = x[i+1];
        N--;
        return 1; //operatie incheiata cu succes
    }
}
```

Eliminarea ultimul element constă în decrementarea numărului de elemente al listei, pentru ca acest element să nu mai fie luat în considerare.

```
int StergereSfarsit(int x[NMAX], int &N, int &a) {
    if (N==0) {
        cout << "Nu sunt elemente de sters"; //underflow
        return 0; //operatie incheiata cu esec
    }
    a = x[--N]; //sau {a = x[N-1]; N--;}
    return 1; //operatie incheiata cu succes
}
```

STIVE

O *stivă* (de exemplu stivă de farfurii) este o listă liniară în care, la un moment dat, este accesibil numai un element și anume elementul aflat la unul din capetele listei, putându-se introduce elemente noi la același capăt. Stivele mai sunt cunoscute și sub numele de LIFO (*last in – first out*), *pilă* sau *stack*. Elementul care se află la capătul unde se pot face operații de scoatere și introducere de elemente se numește **vârful stivei**. Elementul din celălalt capăt al stivei se numește **baza stivei**. La scoaterea unui element, după ce s-a citit valoarea corespunzătoare vârfului stivei, acest element este eliminat și următorul element devine vârful stivei. De fiecare dată se elimină din stivă elementul cel mai "nou" dintre elementele stivei. Dacă stiva nu conține nici un element, spunem că ea este *vidă*. În cazul când toate locațiile din spațiul rezervat stivei sunt ocupate și vrem să mai punem un element apare fenomenul numit *depășire superioară* sau *overflow* și operația nu se poate face. Dacă stiva este vidă și vrem să scoatem un element apare fenomenul numit *depășire inferioară* sau *underflow* și nici în acest caz nu se poate

efectua operația. În celelalte cazuri se pot efectua operații de punere sau scoatere de elemente.

Implementarea în C/C++ operațiile elementare (inițializare, inserare element și eliminare element) pentru o stivă reprezentată secvențial se poate face similar cu cea de la liste liniare, inserarea și extragerea făcându-se la sfârșit. Aici, stiva este dată de vectorul elementelor și de poziția vârfului stivei care aici (în C++) indică primul loc liber din vector (și nu indică poziția elementului din vârful stivei ca în Pascal), unde se va face inserarea unui element și fiind, de fapt, egal cu numărul de elemente din listă.

```
#define NMAX 50 //dimensiunea maxima a vectorului
```

```
int V; //varfului stivei = nr. elemente
```

```
int x[NMAX]; //vectorul elementelor
```

```
0 1 ... V-1 V ... NMAX-1
```

x_0	x_1	...	x_{V-1}	x_V	...	
-------	-------	-----	-----------	-------	-----	--

Operația de inițializare presupune doar inițializarea poziției vârfului cu valoarea 0 (prima poziție liberă din vector, care în C++ este 0).

```
void Initializare() //nu am mai declarat stiva ca parametru
```

```
{ V = 0; }
```

```
V=0 ... NMAX-1
```

--	--	--	--	--	--	--

Introducerea unui element a în stivă se face astfel: pe poziția V se pune noul element și apoi se trece vârfului pe poziția următoare.

```
void Inserare(int a) //nu am mai declarat stiva ca parametru
```

```
{ if (V < NMAX) //stiva nu este plina
```

```
    x[V++] = a;
```

```
    else cout << "Nu mai exista spatiu pentru inserare";
```

```
}
```

Eliminarea unui element a din stivă constă în poziționarea vârfului stivei pe elementul anterior și preluarea valorii lui.

```
void Stergere(int &a) {
```

```
    if (V) //stiva nu este vida
```

```
        a = x[--V];
```

```
    else cout << "Nu mai sunt elemente in stiva";
```

```
}
```

R1 (suplimentar).

Enunțul problemei: Să se calculeze nerecursiv valoarea funcției Manna-Pnueli pentru o valoare întreagă dată a argumentului x .

Metoda de rezolvare: Funcția Manna-Pnueli este definită în felul următor:

$$f: \mathbf{Z} \rightarrow \mathbf{Z}, f(x) = \begin{cases} x-1, & x \geq 12 \\ f(f(x+2)), & x < 12 \end{cases}$$

De exemplu, pentru $x = 8$: $f(8) = f(f(10)) = f(f(f(12))) = f(f(11)) = f(f(f(13))) = f(f(12)) = f(11) = f(f(13)) = f(12) = 11$.

Pentru calculul valorii funcției nerecursiv, se poate folosi o stivă, în care inițial se pune valoarea lui x , în exemplul de mai sus 8. Dacă vârful stivei este mai mic decât 12, se adaugă în stivă elementul cu 2 unități mai mare decât acesta; altfel, dacă stiva are mai mult de un element se scot două elemente din stivă; se pune în stivă elementul mai mic cu o unitate decât vechiul vârf al stivei. Procesul se termină când stiva devine vidă, rezultatul calculat fiind ultimul element extras mai puțin cu o unitate.

		12		13				
	10	10	11	11	12		13	
8	8	8	8	8	8	11	11	12

 $\Rightarrow f(8) = 12 - 1 = 11.$

Un program C++ ce rezolvă problema poate fi următorul:

```
# include <iostream>
# include <conio.h> //pt.getch
# define NMAX 50
using namespace std;
int V,x[NMAX];
void Initializare() {V=0;};
void Inserare(int a)
{ if(V < NMAX) //stiva nu este plina
    x[V++] = a;
  else cout << "Nu mai exista spatiu pentru inserare";
}
void Stergere(int &a)
{ if (V) //stiva nu este vida
    a = x[--V];
  else cout << "Nu mai sunt elemente in stiva";
}

void Listare()
{ if (V==0) cout << "Stiva este vida";
  else
  { int i;
    cout << "Elementele stivei: ";
    for (i=0;i<V;i++) cout << x[i]<<" ";
  }
  cout << endl;
  getch();
}

int main()
{ int x;
  cout << "Dati argumentul functiei Manna-Pnueli:";
  cout << " x = "; cin >> x; cout << endl;
  int xx = x, y; // retinem valoarea initiala a lui x
  Initializare(); Inserare(x); Listare();
  do
  { if(x < 12)
    { cout << "      " << x << " < 12, deci ";
      x += 2; cout << "inserez pe " << x << endl;
      Inserare(x);
      Listare();
    }
    else
    { cout << "      " << x << " >= 12, deci ";
      cout << "scot varful stivei" << endl;
      Stergere(x);
      if(V == 0)
      { cout << "Deci f(" << xx << ") = " << x - 1 << endl;
        return 0;
      }
      else
      { cout << "scot inca un element din varf ";
        Stergere(y);
        cout << "si inserez " << --x << endl;
        Inserare(x);
        Listare();
      }
    }
  } while(1);
  return 0;
}
```

R2. (suplimentar)

Enunțul problemei: (Forma poloneză) Se dă o expresie aritmetică ce folosește operatorii “+”, “-”, “*”, “/”, fiecare operand fiind o literă sau o cifră. Se cere să se convertească această expresie într-o expresie în forma poloneză (postfixată) folosind o stivă.

De exemplu, forma poloneză postfixată a expresiei $a+b$ este $ab+$, a expresiei $a*(b+c)$ este $abc+*$, iar forma poloneză a expresiei $a*(b+c)-e/(a+d)+h$ este $abc+*ead+/-h+$.

Metoda de rezolvare: Pentru realizarea conversiei, operatorii se evaluează cu o anumită prioritate: “+” și “-” cu prioritatea 1, “*” și “/” cu prioritatea 2, iar “(” și “)” cu prioritatea 0 și se folosește o stivă S . Expresia aritmetică inițială se citește în variabila exp , forma poloneză este construită în variabila fp . Pentru buna funcționare a algoritmului, expresia citită se include între paranteze. Se citește fiecare caracter al expresiei și se procedează astfel:

- operatorul “(” se trece în stivă;
- la întâlnirea operatorului “)” se trec din stivă în fp toți operatorii, până la întâlnirea operatorului “(”, operator care se șterge din stivă;
- la întâlnirea operatorilor “+”, “-”, “*”, “/” se compară prioritatea sa cu prioritatea operatorului aflat în vârful stivei: dacă este mai mică sau egală se copiază din stivă în fp toți operatorii cu prioritate mai mare sau egală decât acesta, apoi acest operator se trece în stivă; în caz contrar, acest operator se trece direct în stivă;
- la întâlnirea unui operand, acesta se trece în fp .

De exemplu, pentru expresia $(a*(b+c)-e/(a+d)+h)$ se procedează astfel:

- se citește “(” și se pune în stivă (deci stiva conține: (și fp este vidă);
- se citește “a” și se pune în fp (deci stiva conține: (și fp conține: a);
- se citește “*” și se pune în stivă (deci stiva conține: (* și fp conține: a);
- se citește “(” și se pune în stivă (deci stiva conține: (* (și fp conține: a);
- se citește “b” și se trece în fp (deci stiva conține: (* (și fp conține: ab);
- se citește “+” ce are prioritatea 1, mai mare decât prioritatea vârfului stivei “(” cu prioritatea 0 și deci se pune în stivă (deci stiva conține: (* (+ și fp conține: ab);
- se citește “c” și se pune în fp (deci stiva conține: (* (+ și fp conține: abc);
- se citește “)” și deci se trec din stivă în fp toți operatorii până la “(”, adică “+” și se șterge “(” din stivă (deci stiva conține: (* și fp conține: abc+);
- se citește “-” care are prioritatea 1, mai mică decât prioritatea vârfului stivei cu prioritatea 2 și se trec toți operatorii cu prioritate mai mare sau egală cu 1 din stivă în fp cu excepția parantezelor, apoi se trece în stivă “-” (deci stiva conține: (- și fp conține: abc+*);
- se citește “e” și se pune în fp (deci stiva conține: (- și fp conține: abc+*e);
- se citește “/” care are prioritate 2, mai mare decât vârful stivei, deci se pune în stivă (deci stiva conține: (- / și fp conține: abc+*e);
- se citește “(” și se trece în stivă (deci stiva conține: (- / (și fp conține: abc+*e);
- se citește “a” și se pune în fp (deci stiva conține: (- / (și fp conține: abc+*ea);
- se citește “+”, care are prioritate 1, mai mare decât a vârfului stivei și se pune în stivă (deci stiva conține: (- / (+ și fp conține: abc+*ea);
- se citește “d” și se pune în stivă (deci stiva conține: (- / (+ și fp conține: abc+*ead);
- se citește “)” și se trec din stivă în fp toți operatorii (fără paranteze) până la întâlnirea operatorului “(”, care se șterge din stivă (deci stiva conține: (- / și fp conține: abc+*ead+);
- se citește “+” care are prioritate 1, mai mică decât a vârfului stivei, deci se scot din stivă operatorii cu prioritate mai mică sau egală (adică / și -) și se pun în fp , apoi se trece operatorul citit în stivă (deci stiva conține: (+ și fp conține: abc+*ead+/-);

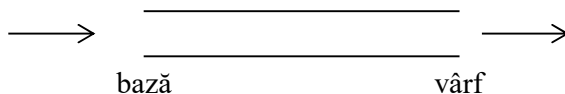
- se citește “h” și se trece în *fp* (deci stiva conține: (+ și *fp* conține: *abc+*ead+/-h*);
- se citește “)” se trece din stivă în *fp* operatorul “+” și se șterge “)” (deci stiva este vidă *fp* conține: *abc+*ead+/-h*).

Un program C++ pentru trecerea în forma poloneză poate avea următoarea formă:

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
using namespace std;
char s0[20], s[20], fp[20], stiva[20];
int varf, n, i;
int Pri(char c)
{switch (c)
{case '*':
case '/': return 2;
case '+':
case '-': return 1;
case '(':
case ')': return 0;
}
return -1;
}
int main()
{cout<<"Introduceti expresia aritmetica: ";cin>>s0;
//se verifica daca sirul este introdus corespunzator
for (i=0;i<strlen(s0);i++)
{if ( (!isdigit(s0[i])) && (!isalpha(s0[i])))//nu-i litera/cifra
if (strchr("()+-*/",s0[i])==NULL) //nu-i car ()+-*/
{ cout<<"Eroare"; exit(1); }
}
//se completeaza expresia cu ( la inceput si ) la sfarsit
strcpy(s,"("); strcat(s,s0); strcat(s,")");
//se determina forma poloneza a expresiei date
for (i=0;i<strlen(s);i++)
{switch (s[i])
{case '(': stiva[++varf]='(';break;
case ')': while (stiva[varf]!='(')
fp[++n]=stiva[varf--];
varf--; break;
case '+':
case '-':
case '*':
case '/':
if (Pri(stiva[varf])>=Pri(s[i]))
while (Pri(stiva[varf])>=Pri(s[i]))
{
fp[++n]=stiva[varf];
varf--;
}
stiva[++varf]=s[i];break;
default: fp[++n]=s[i];
}
}
}
cout<<endl<<"Forma poloneza postfixata este: ";
for (i=1;i<=n;i++) cout<<fp[i];
return 0;
}
```


COZI

O **coadă** este o listă liniară în care este accesibil la un moment dat numai un element și anume elementul aflat la unul din capetele listei putându-se introduce elemente noi la celălalt capăt. Cozile mai sunt cunoscute și sub numele de FIFO (*first in – first out*) sau *queue*.



Elementul care se află la capătul unde se pot face **operații de scoatere** de elemente se numește **vârful cozii**. Elementul din celălalt capăt al cozii, **unde se pot introduce elemente**, se numește **baza cozii**. La scoaterea unui element, după ce s-a citit valoarea corespunzătoare vârfului, acest element este eliminat și următorul element devine vârful cozii. De fiecare dată se elimină din coadă elementul cel mai "vechi" dintre elementele cozii. La adăugarea unui element, noul element devine baza cozii. Dacă coada nu conține nici un element, spunem că ea este *vidă*. În cazul când toate locațiile din spațiul rezervat cozii sunt ocupate și vrem să mai punem un element apare fenomenul numit *depășire superioară* sau *overflow* și operația nu se poate face. Dacă coada este vidă și vrem să scoatem un element apare fenomenul numit *depășire inferioară* sau *underflow* și nici în acest caz nu se poate efectua operația. În celelalte cazuri se pot efectua operații de punere sau scoatere de elemente.

Cozi reprezentate secvențial

Pentru *reprezentarea secvențială a unei cozi C* să presupunem că au fost rezervate $NMAX$ locații de memorie ce pot fi referite prin x_i pentru cea de-a i -a locație rezervată. Vom prezenta trei tipuri de reprezentări secvențiale.

În prima reprezentare, cu **elemente la începutul spațiului afectat**, va avea vârful cozii pe primul loc și va fi suficient să utilizăm o variabilă suplimentară B care să indice baza cozii numită *pointerul cozii*. Așadar elementele cozii sunt de la poziția 1 până la poziția indicată de B . Este cazul particular al listelor, cu **inserare la început** și **ștergere la sfârșit**.

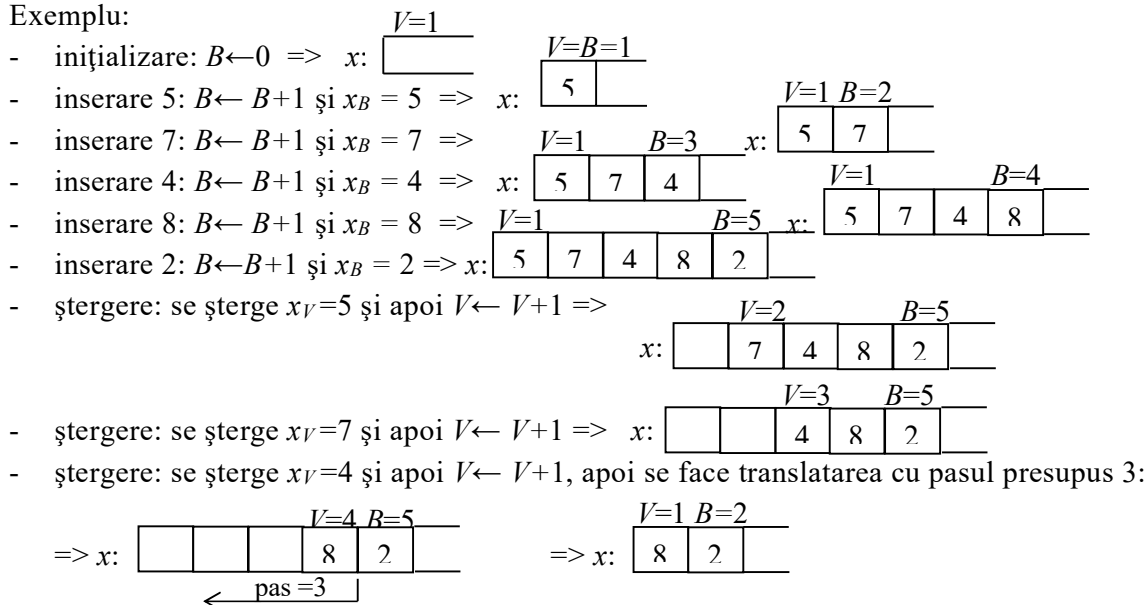
Exemplu:

- inițializare: $B \leftarrow 0 \Rightarrow x: \begin{array}{|c|} \hline 1 \\ \hline \end{array}$
- inserare 5: $B \leftarrow B+1$ și $x_B = 5 \Rightarrow x: \begin{array}{|c|c|} \hline B=1 & \\ \hline 5 & \end{array}$
- inserare 7: $B \leftarrow B+1$ și $x_B = 7 \Rightarrow x: \begin{array}{|c|c|c|} \hline 1 & B=2 & \\ \hline 5 & 7 & \end{array}$
- inserare 4: $B \leftarrow B+1$ și $x_B = 4 \Rightarrow x: \begin{array}{|c|c|c|c|} \hline 1 & B=3 & & \\ \hline 5 & 7 & 4 & \end{array}$
- ștergere: se șterge $x_1=5$ și pentru aceasta restul elementelor se deplasează cu un pas spre stânga, apoi $B \leftarrow B+1 \Rightarrow x: \begin{array}{|c|c|c|} \hline B=2 & & \\ \hline 7 & 4 & \end{array}$

Deoarece **la fiecare scoatere de element sunt deplasate elementele rămase în coadă** cu o locație către începutul vectorului, sistemul poate să devină ineficient pentru cazul în care apar un număr mare de elemente în coadă.

A doua metodă, cu **translatări cu pași diferiți** ale elementelor, va utiliza două variabile suplimentare: V care să indice vârful cozii și B care să indice baza cozii numite *pointerii cozii*.

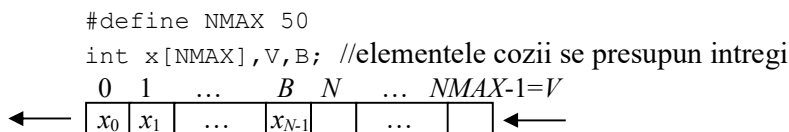
Exemplu:



Această metodă este ceva mai eficientă decât metoda precedentă, deplasările elementelor făcându-se, în general, cu pași mai mari. Eficiența scade cu cât numărul de elemente din coadă se apropie de $NMAX$.

În cea de-a treia metodă, numită **coadă circulară**, elementele nu se mai deplasează. Se presupune că după x_{NMAX} urmează, din punct de vedere logic, x_1 . Vom utiliza, ca în cazul precedent, două variabile suplimentare: V care să indice vârful cozii și B care să indice baza cozii numite *pointerii cozii*. Pentru a deosebi cazurile excepție, vom presupune că în spațiul alocat cel puțin un element nu este ocupat în orice moment.

Pentru implementarea C/C++ operațiile elementare ale unei cozi reprezentată secvențial, tip coadă circulară, în afară de vectorul elementelor se folosesc două variabile suplimentare: V , vârful cozii (de unde se extrag), care reprezintă **poziția anterioară primului element ce se poate extrage** și B , baza cozii (unde se introduc elementele), care reprezintă **poziția ultimului element introdus în coadă**. Deci structura listei apare astfel:



Inițializarea listei ca listă vidă presupune:

```
void Initializare() {V = B = NMAX-1;}
```

Inserarea unui nou element se face la baza cozii astfel:

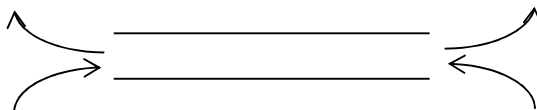
```
void InserareElem (int a)
{ if (V == (B+1)%NMAX) cout << "Coada este plina";
  else {B = (B+1)%NMAX; x[B] = a; }
}
```

Ștergerea unui element din vârful stivei se face astfel:

```
void StergereElem (int &a)
{ if (V == B) cout<<"Nu sunt elemente de sters";
  else { V = (V+1)%NMAX; a = x[V]; }
}
```

COZI COMPLETE

O **coadă completă** este o listă liniară în care operațiile de intrare și ieșire se pot face la oricare dintre capetele listei. Cozile complete mai sunt cunoscute și sub numele de *decoadă* sau *deque*.



Elementul care se află la unul dintre capete se numește *capătul stâng al cozii complete* și cel din celălalt capăt se numește *capătul drept*. La scoaterea unui element, după ce s-a citit valoarea corespunzătoare elementului din unul dintre capete, acest element este eliminat și următorul element devine noul capăt. La adăugarea unui element la un capăt, noul element devine capăt. Dacă coada completă nu conține nici un element, spunem că ea este *vidă*. În cazul când toate locațiile din spațiul rezervat cozii complete sunt ocupate și vrem să mai punem un element apare fenomenul numit *depășire superioară* sau *overflow* și operația nu se poate face. Dacă coada este vidă și vrem să scoatem un element apare fenomenul numit *depășire inferioară* sau *underflow* și nici în acest caz nu se poate efectua operația. În celelalte cazuri se pot efectua operații de punere sau scoatere de elemente. Dacă nu sunt permise operații de intrare sau de ieșire la unul din capetele cozii complete spunem că este o coadă completă *restricționată la intrare* și, respectiv, *restricționată la ieșire* la acel capăt. Stiva poate fi privită ca o coadă completă restricționată la intrare și la ieșire la unul dintre capete (baza stivei), iar coada ca o coadă completă restricționată la intrare la un capăt (vârful cozii) și restricționată la ieșire la celălalt capăt (baza cozii).