BIBLIOGRAFIE (partea de Structuri de date):

1. I. Ignat, C.L. Ignat, *Structuri de date şi algoritmi*, Editura Albastră, Cluj-Napoca, 2007.
2. O.Bâscă, M.Jaică (Miroiu), *Structuri de date (note de curs)*, Editura Universităţii din Piteşti, 2000.
3. M.D Zaharia, *Structuri de date şi algoritmi – algoritmi, exemple în limbajele C şi C++*, Editura Albastră, Cluj-Napoca, 2002.
4. I. Tomescu, *Data Structures*, Editura Universităţii din Bucureşti, 1997.
5. Iorga, V., Chiriţă, P., Stratan, C., Opincaru, C., *Programare în C/C++. Culegere de probleme*. Editura Niculescu, Bucureşti, 2003.
6. T.C. Ionescu, I. Zsako, *Structuri arborescente cu aplicaţiile lor*, Editura Tehnică, Bucureşti, 1990.
7. D.Burdescu, *Structuri de date arborescente,* vol. 1 şi 2, Editura Mirton, Timişoara, 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 şi analiza algoritmilor*, Editura Ştiintifică şi enciclopedică, 1986.
10. R. Andonie, I. Gabarcea, *Algoritmi fundamentali. O perspectiva C++*, Editura Libris, 1995.
11. Octavian Pătrăşcoiu, Gheorghe Marian, Nicolae Mitroi, *Elemente de grafuri, combinatorică, metode, algoritmi şi programe*, Editura All, Bucureşti, 1994.
12. Viorel Păun, *Algoritmică şi programarea calculatoarelor. Limbajul C++.* Editura Universităţii din Piteşti, 2003.
13. B. Pătruţ, *Aplicaţii în C şi 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 înşiruire de elemente de acelaşi tip în care se distinge un *prim element* şi un *ultim element*, nu neapărat distincte, fiecare element în afară de primul având un element care îl precede şi fiecare element în afară de ultimul având unul care îl succede. O listă care nu conţine nici un element se numeşte *listă vidă*.

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

- accesul la un anumit element pentru prelucrarea lui (examinare, modificare etc.);
- inserarea unui nou element într-o poziţie 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;
- despărţirea 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 proprietăţi.

Frecvenţa cu care sunt efectuate diferitele operaţii asupra listelor determină modul de reprezentare a lor pentru a obţine cea mai bună eficienţă din punct de vedere al spaţiului ocupat şi al timpului de calcul. Metodele de reprezentare se împart în două categorii: *reprezentarea secvenţială* (statică) şi *reprezentarea înlănţuită* (dinamică).

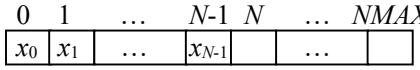
Implementarea secvenţială a listelor liniare

Unul dintre cele mai simple moduri de reprezentare în memoria calculatorului a listelor liniare este modul secvenţial în care elementele listei sunt memorate unul după altul în ordinea în care apar ele în listă, în locaţii consecutive (într-un vector). Notând cu $\&a$ locaţia unde se memorează elementul a din listă, se deduce relaţia $\&x_{i+1}=\&x_i+c$, unde c este numărul de locaţii 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 contă în accesul la oricare element printr-o operaţie simplă şi rapidă, cu durata de execuţie constantă, indiferent de pozitia elementului în listă.

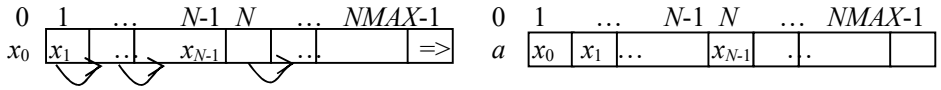
În continuare se descriu operaţiile elementare efectuate asupra unei liste liniare reprezentată secvenţial. Vom memorarea elemente listei într-un vector global x în poziţii consecutive, ordinea fizică a lor fiind aceeaşi cu ordinea lor logică. Pentru a şti câte elemente există efectiv în listă se defineşte 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
```



Iniţializarea 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 spaţiul rezervat) dacă numărul efectiv al elementelor listei este chiar NMAX (N==NMAX).

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

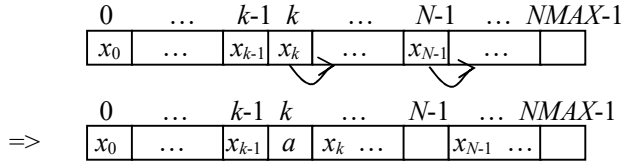


Adică: $x_N \leftarrow x_{N-1}$
$\dots$
$x_2 \leftarrow x_1$
$x_1 \leftarrow x_0$

$\Leftrightarrow$ $x_i \leftarrow x_{i-1}$, $i = N, 1, -1$ 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 poziţie k, se face prin deplasarea elementelor de la ultimul până la al k-lea cu o poziţie spre dreapta şi inserarea noului element pe locul rămas liber. În particular, pentru $k = 0$ se obţine cazul precedent.



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

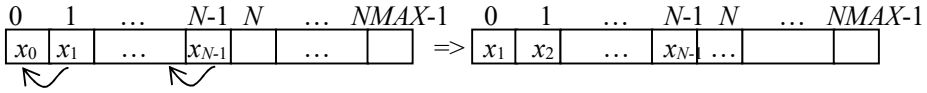
$\Leftrightarrow$ $x_i \leftarrow x_{i-1}$, $i = N, k+1, -1$ sau $x_{i+1} \leftarrow x_i$, $i = N-1, k, -1$.

```
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 sfârşitul 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 poziţie spre stânga.

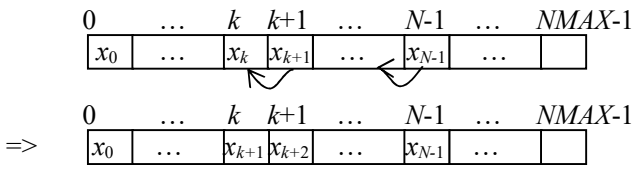


Adică: $x_0 \leftarrow x_1$ … $x_{N-2} \leftarrow x_{N-1}$

$\Leftrightarrow$ $x_i \leftarrow x_{i+1}$, $i = 0, N-2$ 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 reţine 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 poziţia k se face prin deplasarea elementelor de la poziţia $k+1$ până la ultima cu o poziţie 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 reţine valoarea elementului de şters
        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 şi anume elementul aflat la unul din capetele listei, putându-se introduce elemente noi la acelaşi capăt. Stivele mai sunt cunoscute şi sub numele de LIFO (*last in – first out*), *pilă* sau *stack*. Elementul care se află la capătul unde se pot face operaţii de scoatere şi introducere de elemente se numeşte *vârful stivei*. Elementul din celălalt capăt al stivei se numeşte *baza stivei*. La scoaterea unui element, după ce s-a citit valoarea corespunzătoare vârfului stivei, acest element este eliminat şi următorul element devine vârful stivei. De fiecare dată se elimină din stivă elementul cel mai "nou" dintre elementele stivei. Dacă stiva nu conţine nici un element, spunem că ea este *vidă*. În cazul când toate locaţiile din spaţiul rezervat stivei sunt ocupate şi vrem să mai punem un element apare fenomenul numit *depăşire superioară* sau *overflow* şi operaţia nu se poate face. Dacă stiva este vidă şi vrem să scoatem un element apare

fenomenul numit *depăşire inferioară* sau *underflow* şi nici în acest caz nu se poate efectua operaţia. În celelalte cazuri se pot efectua operaţii de punere sau scoatere de elemente.

Implementarea în C/C++ operaţiile elementare (iniţializare, inserare element şi eliminare element) pentru o stivă reprezentată secvenţial se poate face similar cu cea de la liste liniare, inserarea şi extragerea făcându-se la sfârşit. Aici, stiva este dată de vectorul elementelor şi de poziţia vârfului stivei care aici (în C++) indică primului loc liber din vector (şi nu indică poziţia elementului din vârful stivei ca în Pascal), unde se va face inserarea unui element şi 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}	…		

Operaţia de iniţializare presupune doar iniţializarea poziţiei vârfului cu valoarea 0 (prima poziţie 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 poziţia *V* se pune noul element şi apoi se trece vârfului pe poziţia 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 poziţionarea vârfului stivei pe elementul anterior şi 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).

Enunţul problemei: **Să se calculeze nerecursiv valoarea funcţiei Manna-Pnueli pentru o valoare întreagă dată a argumentului *x*.**

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

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

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 funcţiei nerecursiv, se poate folosi o stivă, în care iniţial 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 unităţi 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 puţin cu o unitate.

8	10 8	12 10 8	11 8	13 11 8	12 8	11	13 11	12	=> $f(8) = 12 - 1 = 11$.

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)

Enunţul problemei: **(Forma poloneză) Se dă o expresie aritmetică ce foloseşte 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: "+" şi "-" cu prioritatea 1, "*" şi "/" cu prioritatea 2, iar "(" şi ")" cu prioritatea 0 şi se foloseşte o stivă *S*. Expresia aritmetică iniţială se citeşte în variabila *exp*, forma poloneză este construită în variabila *fp.* Pentru buna funcţionare a algoritmului, expresia citită se include între paranteze. Se citeşte fiecare caracter al expresiei şi se procedează astfel:

- operatorul "(" se trece în stivă;
- la întâlnirea operatorului ")" se trec din stivă în *fp* toţi operatorii, până la intâlnirea operatorului "(", operator care se şterge 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* toţi 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 exmplu, pentru expresia (*a**(*b*+*c*)-*e*/(*a*+*d*)+*h*) se procedează astfel:

- se citeşte "(" şi se pune în stivă (deci stiva conţine: (şi *fp* este vidă);
- se citeşte "*a*" şi se pune în *fp* (deci stiva conţine: (şi *fp* conţine: *a*);
- se citeşte "*" şi se pune în stivă (deci stiva conţine: (* şi *fp* conţine: *a*);
- se citeşte "(" şi se pune în stivă (deci stiva conţine: (* (şi *fp* conţine: *a*);
- se citeşte "*b*" şi se trece în *fp* (deci stiva conţine: (* (şi *fp* conţine: *ab*);
- se citeşte "+" ce are prioritatea 1, mai mare decât prioritatea vârfului stivei "(" cu prioritatea 0 şi deci se pune în stivă (deci stiva conţine: (* (+ şi *fp* conţine: *ab*);
- se citeşte "*c*" şi se pune în *fp* (deci stiva conţine: (* (+ şi *fp* conţine: *abc*);
- se citeşte ")" şi deci se trec din stivă în *fp* toţi operatorii până la "(", adică "+" şi se şterge "(" din stivă (deci stiva conţine: (* şi *fp* conţine: *abc*+);
- se citeşte "-" care are prioritatea 1, mai mică decât prioritatea vârfului stivei cu prioritatea 2 şi se trec toţi operantorii cu prioritate mai mare sau egală cu 1 din stivă în *fp* cu excepţia parantezelor, apoi se trece în stivă "-" (deci stiva conţine: (- şi *fp* conţine: *abc*+*);
- se citeşte "*e*" şi se pune în *fp* (deci stiva conţine: (- şi *fp* conţine: *abc*+**e*);
- se citeşte "/" care are prioritate 2, mai mare decât vârful stivei, deci se pune în stivă (deci stiva conţine: (- / şi *fp* conţine: *abc*+**e*);
- se citeşte "(" şi se trece în stivă (deci stiva conţine: (- / (şi *fp* conţine: *abc*+**e*);
- se citeşte "*a*" şi se pune în *fp* (deci stiva conţine: (- / (şi *fp* conţine: *abc*+**ea*);
- se citeşte "+", care are prioritate 1, mai mare decât a vârfului stivei şi se pune în stivă (deci stiva conţine: (- / (+ şi *fp* conţine: *abc*+**ea*);
- se citeşte "*d*" şi se pune în stivă (deci stiva conţine: (- / (+ şi *fp* conţine: *abc*+**ead*);
- se citeşte ")" şi se trec din stivă în *fp* toţi operatorii (fără paranteze) până la întâlnirea operatorului "(", care se şterge din stivă (deci stiva conţine: (- / şi *fp* conţine: *abc*+**ead*+);
- se citeşte "+" care are prioritate 1, mai mică decât a vârfului stivei, deci se scot din stivă operatorii cu prioritate mai mică sau egală (adică / şi -) şi se pun în *fp*, apoi se trece operatorul citit în stivă (deci stiva conţine: (+ şi *fp* conţine: *abc*+**ead*+/-);

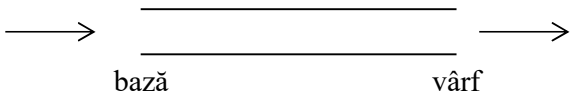
- se citeşte "*h*" şi se trece în *fp* (deci stiva conţine: (+ şi *fp* conţine: *abc+*ead+/-h*);
- se citeşte ")" se trece din stivă în *fp* operatorul "+" şi se şterge ")" (deci stiva este vidă *fp* conţine: *abc+*ead+/-h+*).

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 şi anume elementul aflat la unul din capetele listei putându-se introduce elemente noi la celălalt capăt. Cozile mai sunt cunoscute şi sub numele de FIFO (*first in – first out*) sau *queue*.



Elementul care se află la capătul unde se pot face operaţii de scoatere de elemente se numeşte *vârful cozii*. Elementul din celălalt capăt al cozii, unde se pot introduce elemente, se numeşte *baza cozii*. La scoaterea unui element, după ce s-a citit valoarea corespunzătoare vârfului, acest element este eliminat şi 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 conţine nici un element, spunem că ea este *vidă*. În cazul când toate locaţiile din spaţiul rezervat cozii sunt ocupate şi vrem să mai punem un element apare fenomenul numit *depăşire superioară* sau *overflow* şi operaţia nu se poate face. Dacă coada este vidă şi vrem să scoatem un element apare fenomenul numit *depăşire inferioară* sau *underflow* şi nici în acest caz nu se poate efectua operaţia. În celelalte cazuri se pot efectua operaţii de punere sau scoatere de elemente.

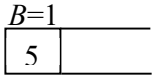
Cozi reprezentate secvenţial

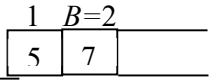
Pentru *reprezentarea secvenţială a unei cozi C* să presupunem că au fost rezervate *NMAX* locaţii de memorie ce pot fi referite prin x_i pentru cea de-a *i*-a locaţie rezervată. Vom prezenta trei tipuri de reprezentări secvenţiale.

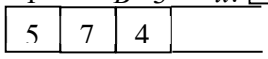
În prima reprezentare, cu **elemente la începutul spaţiului afectat**, va avea vârful cozii pe primul loc şi va fi suficient să utilizăm o variabilă suplimentară *B* care să indice baza cozii numită *pointerul cozii*. Aşadar elementele cozii sunt de la poziţia 1 până la poziţia indicata de *B*. Este cazul particular al listelor, cu inserare la început şi ştergere la sfârşit.

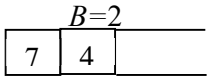
Exemplu:

- iniţializare: $B \leftarrow 0$ => x: 

- inserare 5: $B \leftarrow B+1$ şi $x_B = 5$ => x: 

- inserare 7: $B \leftarrow B+1$ şi $x_B = 7$ => 

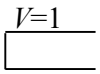
- inserare 4: $B \leftarrow B+1$ şi $x_B = 4$ => x: 

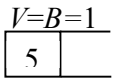
- ştergere: se şterge x_1=5 şi pentru aceasta restul elementelor se deplasează cu un pas spre stânga, apoi $B \leftarrow B+1$ => x: 

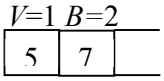
Deoarece la fiecare scoatere de element sunt deplasate elementele rămase în coadă cu o locaţie către începutul vectorului, sistemul poate să devină ineficient pentru cazul în care apar un număr mare de elemente în coadă.

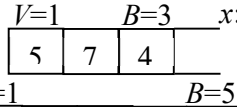
A doua metodă, cu **translatări cu paşi diferiţi** ale elementelor, va utiliza două variabile suplimentare: *V* care să indice vârful cozii şi *B* care să indice baza cozii numite *pointerii cozii*.

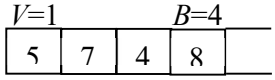
Exemplu:

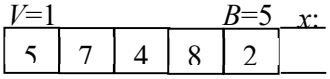
- iniţializare: $B \leftarrow 0$ => x: 

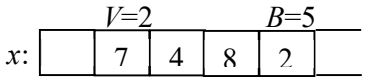
- inserare 5: $B \leftarrow B+1$ şi $x_B = 5$ => x: 

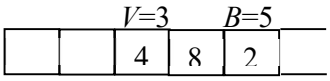
- inserare 7: $B \leftarrow B+1$ şi $x_B = 7$ => 

- inserare 4: $B \leftarrow B+1$ şi $x_B = 4$ => x: 

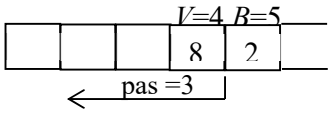
- inserare 8: $B \leftarrow B+1$ şi $x_B = 8$ => 

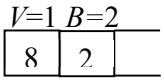
- inserare 2: $B \leftarrow B+1$ şi $x_B = 2$ => x: 

- ştergere: se şterge $x_V=5$ şi apoi $V \leftarrow V+1$ => 

- ştergere: se şterge $x_V=7$ şi apoi $V \leftarrow V+1$ => x: 

- ştergere: se şterge $x_V=4$ şi apoi $V \leftarrow V+1$, apoi se face translatarea cu pasul presupus 3:

=> x: 

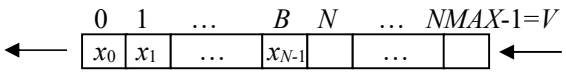
=> x: 

Această metodă este ceva mai eficientă decât metoda precedentă, deplasările elementelor făcându-se, în general, cu paşi mai mari. Eficienţa 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 şi *B* care să indice baza cozii numite *pointerii cozii*. Pentru a deosebi cazurile excepţie, vom presupune că în spaţiul alocat cel puţin un element nu este ocupat în orice moment.

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

```
#define NMAX 50
int x[NMAX],V,B; //elementele cozii se presupun intregi
```



Iniţializarea listei ca listă vidă presupune:

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

Inserarea unui nou element se face la baza cozii astfel:

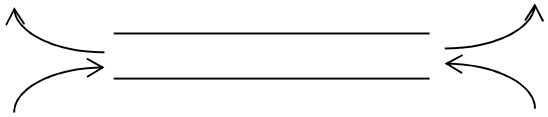
```
void InserareElem (int a)
{ if (V == (B+1)%NMAX) cout << "Coada este plina";
  else  {B = (B+1)%NMAX; x[B] = a; }
}
```

Ştergerea 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 operaţiile de intrare şi ieşire se pot face la oricare dintre capetele listei. Cozile complete mai sunt cunoscute şi sub numele de *decoadă* sau *deque*.



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

EXEMPLE DE PROGRAME CU LISTE IMPLEMENTATE STATIC (VECTORI)

R1.

Enunţul problemei: **Să se determine un algoritm prin care să se adauge elementele unui vector la sfârşitul altui vector.** De exemplu, pentru $x = (1\ 2\ 3\ 4)$ şi $y=(5\ 7\ 9)$, după adăugarea elementelor vectorului y la sfârşitul vectorului x, se obţine $x = (1\ 2\ 3\ 4\ 5\ 7\ 9)$.

Metoda de rezolvare: Considerăm primul vector $(x_0,\ldots, x_{m-1})$ şi al doilea vector $(y_0,\ldots, y_{n-1})$ – acestea sunt datele de intrare. Adăugând elementele vectorului y la sfârşitul vectorului x se obţine:

Pentru aceasta: $x_m \leftarrow y_0$ … $x_{m+n-1} \leftarrow y_{n-1}$

$\Leftrightarrow\ x_{m+i} \leftarrow y_i$, $i = 0,2,\ldots,n-1$.

Descrierea algoritmului în C++:

```
#include<iostream>
#define NMAX 50
using namespace std;
int n,m;
float x[NMAX],y[NMAX];
```

```
void CitireVector(float x[NMAX], int &n){
//se citeste dimensiunea unui vector si elementele sale
 cout<<"Dati dim vect: "; cin>>n;
 cout<<"Dati elementele: ";
 for (int i=0;i<n;i++) cin>>x[i];
}
void AfisareVector(float x[NMAX], int n) {
//afisarea elementelor unui vector
 for (int i=0;i<n;i++) cout<<x[i]<<" ";
}
void AdaugareSfarsit() {
 for (int i=0;i<n;i++) x[m+i] = y[i];
}
int main() {
 cout<<"Primul vector."<<endl;
 CitireVector(x,m);
 cout<<"Al doilea vector."<<endl;
 CitireVector(y,n);
 AdaugareSfarsit();
 cout<<"Dupa adaugarea elem celui de-al doilea vector la
sfarsitul primului vector:"<<endl;
 AfisareVector(x,n+m);
 return 0;
}
```

Rulare:

```
Primul vector.
Dati dim vect: 2
Dati elementele: 5 5
Al doilea vector.
Dati dim vect: 3
Dati elementele: 7 8 9
Dupa adaugarea elem celui de-al doilea vector la sfarsitul
primului vector:
5 5 7 8 9
```

R2.

Enunţul problemei: **Să se determine un algoritm pentru adaugarea elementelor unui vector la începutul altui vector.** De exemplu, pentru x = (1 2 3 4) şi y = (5 7 9), după adăugarea elementelor vectorului y la începutul vectorului x, se obţine x = (5 7 9 1 2 3 4).

Metoda de rezolvare: Considerăm primul vector $x_0,\ldots, x_{m-1}$ şi al doilea vector $y_0,\ldots, y_{n-1}$, acestea fiind datele de intrare. Adăugând elementele vectorului y la începutul vectorului x se obţine:

$$x: \overbrace{x_0 \ \ldots \ x_{m-1}}^{0 \ \ldots \ m-1} \longrightarrow x: \overbrace{y_0 \ \ldots \ y_{n-1} \ x_0 \ldots x_{m-1}}^{1 \ \ldots \ n-1 \quad n \ \ldots \ n+m-1}$$

Dar pentru inserarea celor n elemente din y la început, trebuie să "facem loc" pentru n componente, adică să deplasăm elementele din x spre dreapta (de la ultimul din grup la primul), nu cu câte o poziţie, ci cu câte n poziţii spre dreapta, adică

$$\begin{vmatrix} x_{n+m-1} \leftarrow x_{m-1} \\ \ldots \\ x_n \leftarrow x_0 \end{vmatrix}$$

$$\Leftrightarrow \ x_{n+i} \leftarrow x_i \ , \ i = m\text{-}1, \ m\text{-}2\ldots,0.$$

Apoi inserăm la început elementele vectorului y:

$$\begin{vmatrix} x_0 \leftarrow y_0 \\ \ldots \\ x_{n-1} \leftarrow y_{n-1} \end{vmatrix}$$

$$\Leftrightarrow \ x_i \leftarrow y_i \ , \ i = 1,2,\ldots, n.$$

Descrierea algoritmului în C++:

```
#include<iostream>
#define NMAX 50
using namespace std;
int n,m;
float x[NMAX],y[NMAX];

void CitireVector(float x[NMAX], int &n) {
//se citeste dimensiunea unui vector si elementele sale
 cout<<"Dati dim vect: "; cin>>n;
 cout<<"Dati elementele: ";
 for (int i=0;i<n;i++) cin>>x[i];
}
void AfisareVector(float x[NMAX], int n) {
//afisarea elementelor unui vector
 for (int i=0;i<n;i++) cout<<x[i]<<" ";
}
void AdaugareInceput() {
//deplasarea elem din x cu n pozitii spre dreapta, primul
//elem.ce se "deplaseaza" fiind cel mai din dreapta vectorului
 int i;
 for (i=m-1;i>=0;i--) x[n+i]=x[i];
                 //elem de pe poz.i se copiaza pe pozitia n+i
 //inseram elem din y
 for (i=0;i<n;i++) x[i] = y[i];
}
int main() {
 cout<<"Primul vector."<<endl;  CitireVector(x,m);
 cout<<"Al doilea vector."<<endl;  CitireVector(y,n);
 AdaugareInceput();
 cout<<"Dupa adaugarea elem celui de-al doilea vector la
inceputul primului vector:"<<endl;
 AfisareVector(x,n+m);
 return 0;
}
```

Rulare:

```
Primul vector.
Dati dim vect: 2
Dati elementele: 5 5
Al doilea vector.
Dati dim vect: 3
Dati elementele: 7 8 9
Dupa adaugarea elem celui de-al doilea vector la sfarsitul
primului vector:
7 8 9 5 5
```

R3.

Enunţul problemei: **Să se determine un algoritm pentru deplasarea elementelor unui vector cu o poziţie spre dreapta, ultimul element devenind primul (~permutare circulară).** De exemplu, pentru x = (1 2 3 4), după deplasarea elementelor vectorului x cu o poziţie spre dreapta se obţine x = (4 1 2 3).

Metoda de rezolvare: Ideea constă în a reţine ultimul element într-o variabilă suplimentară, apoi penultimul element se "deplasează" spre dreapta cu o poziţie, ş.a.m.d., primul element se "deplasează" spre dreapta cu o poziţie şi la final, pe poziţia "rămasă liberă" se pune valoarea reţinută în variabila suplimentară.

Descrierea algoritmului în C++:

```
#include<iostream>
#define NMAX 50
using namespace std;
int n;
float x[NMAX];

void CitireVector(float x[NMAX], int &n) {
//se citeste dimensiunea unui vector si elementele sale
 cout<<"Dati dim vect: "; cin>>n;
 cout<<"Dati elementele: ";
 for (int i=0;i<n;i++) cin>>x[i];
}
void AfisareVector(float x[NMAX], int n) {
//afisarea elementelor unui vector
 for (int i=0;i<n;i++) cout<<x[i]<<" ";
}
void DeplasareSpreDreapta() {
 float a = x[n-1];
 for (int i=n-2;i>=0;i--) x[i+1]=x[i];
                  //elem. de pe poz.i se copiaza pe pozitia i+1
 x[0] = a;
}
int main()
{
 CitireVector(x,n);
 DeplasareSpreDreapta();
 cout<<"Dupa deplasarea elementelor spre dreapta: "<<endl;
 AfisareVector(x,n);
}
```

R4.

Enunţul problemei: **Să se determine un algoritm pentru eliminarea elementelor impare ale unui vector.** De exemplu, pentru x = (1 2 3 4), după eliminarea elementelor impare se obţine x = (2 4), iar pentru vectorul x = (1 3 3 5), după eliminarea elementelor impare se obţine vectorul vid.

Metoda de rezolvare: Se parcurge vectorul $x_0,\ldots, x_{n-1}$ şi dacă elementul curent (de pe poziţia i) are valoare impară atunci acesta se elimină din vector (prin deplasarea elementelor de pe pozitia i+1 până la n-1 cu o poziţie spre stânga, descrescând apoi numărul efectiv de elemente din vector, $n \leftarrow n$-1).

Descrierea algoritmului în C++:

```
#include<iostream>
#define NMAX 50
using namespace std;
int n;
int x[NMAX];

void CitireVector(int x[NMAX], int &n){
//se citeste dimensiunea unui vector si elementele sale
 cout<<"Dati dim vect: "; cin>>n;
 cout<<"Dati elementele: ";
 for (int i=0;i<n;i++) cin>>x[i];
}
void AfisareVector(int x[NMAX], int n) {
 if (n) //sau if(n!=0)
 {
```

```
  cout<<"Elementele vectorului: ";
  for (int i=0;i<n;i++) cout<<x[i]<<" ";
 }
 else cout<<"Vectorul este vid";
}
void EliminarePoz(int i0, int x[NMAX], int &n) {
 //elimin elem de pe poz i din vect x de dim n (dim se modif)
 //prin depl. elementelor i+1...n-1 cu o pozitie spre stanga
 for (int i=i0+1;i<n;i++) x[i-1]=x[i];
 n--; //numarul efectiv de elemente scade cu o unitate
}
void EliminareValImpare(int x[NMAX], int &n) {
 for (int i=0;i<n;i++)
  if (x[i]%2==1) //sau if (x[i]%2)
    {
     EliminarePoz(i,x,n);
     i--; //ramanem pe loc caci a venit un nou elem pe poz i
     }
}
int main() {
 CitireVector(x,n);
 EliminareValImpare(x,n);
 cout<<"Dupa eliminarea valorilor impare: "<<endl;
 AfisareVector(x,n);
 return 0;
}
```

Rulare:

```
Dati dim vect: 4
Dati elementele: 1 2 3 4
Dupa eliminarea valorilor impare:
Elementele vectorului: 2 4
sau
Dati dim vect: 4
Dati elementele: 1 3 3 5
Dupa eliminarea valorilor impare:
Vectorul este vid
```

O altă variantă a funcției EliminareValImpare este

```
void EliminareValImpare(int x[NMAX], int &n)
{
 int i=0;
 while (i<n)
  if (x[i]%2) //sau if (x[i]%2==1)
     EliminarePoz(i,x,n); //si ramanem pe loc
  else
     i++; //mergem la urmatorul element
}
```

R5.

Enunţul problemei: **Să se determine un algoritm pentru eliminarea elementelor de pe poziţii pare ale unui vector.** De exemplu, pentru $x = (5\ 7\ 2\ 4)$, după eliminarea elementelor impare se obţine $x = (5\ 2)$.

Metoda de rezolvare: Algoritmul pare similar, însă după eliminarea unui element poziţiile elementelor de după el se schimbă faţă de poziţiile iniţiale:

- iniţial:

	1	2	3	4	5	6
x:	5	~~2~~	8	6	7	4

- după eliminarea elementului de pe poziţia a 2-a:

	1	2	3	4	5
x:	5	8	~~6~~	7	4

- la pasul următor trebuie eliminată valoarea 6 care iniţial era pe poziţia 4, acum însă este pe poz 3 şi se obţine:

	1	2	3	4	5
x:	5	8	7	~~4~~	

- la pasul următor trebuie eliminată valoarea 4 care iniţial era pe poziţia 6, acum însă este pe poz 4.

Aşadar, am eliminat în ordine poziţia 2, apoi 3, 4, 5…

Descrierea algoritmului în C++:

```
#include<iostream>
#define NMAX 50
using namespace std;
int n;
int x[NMAX];
void CitireVector(int x[NMAX], int &n){
//se citeste dimensiunea unui vector si elementele sale
 cout<<"Dati dim vect: "; cin>>n;
 cout<<"Dati elementele: ";
 for (int i=0;i<n;i++) cin>>x[i];
}
void AfisareVector(int x[NMAX], int n) {
 if (n!=0) //sau if(n)
 {cout<<"Elementele vectorului: ";
  for (int i=0;i<n;i++) cout<<x[i]<<" ";
 }
 else cout<<"Vectorul este vid";
}
void EliminarePoz(int i0, int x[NMAX], int &n) {
 //elimin elem de pe poz i din vect x de dim n (dim se modif)
 //prin depl elementelor i+1...n-1 cu o pozitie spre stanga
 for (int i=i0+1;i<n;i++) x[i-1]=x[i];
 n--; //numarul efectiv de elemente scade cu o unitate
}
void EliminarePozPare(int x[NMAX], int &n) {
 for (int i=1;i<n;i++) EliminarePoz(i,x,n);
}
int main() {
 CitireVector(x,n);
 EliminarePozPare(x,n);
 cout<<"Dupa eliminarea pozitiilor pare: "<<endl;
 AfisareVector(x,n);
 return 0;
}
```

Rulare:

```
Dati dim vect: 4
Dati elementele: 1 2 3 4
Dupa eliminarea pozitiilor pare:
Elementele vectorului: 1 3
```

```
sau
      Dati dim vect: 5
      Dati elementele: 1 2 3 4 5
      Dupa eliminarea pozitiilor pare:
      Vectorul este 1 3 5
```

R6.

Enunţul problemei: **Să se determine un algoritm pentru eliminarea duplicatelor dintr-un vector.** De exemplu, pt x = (5 7 5 9 5 5 7), după eliminarea duplicatelor se obţine x = (5 7 9). Detalii: x = (**5** 7 5 9 5 5 7) -> x = (5 **7** 9 7) -> x = (5 7 **9**)

Metoda de rezolvare: Se parcurge vectorul şi se verifică dacă valoarea curentă, să zicem x_i, se mai găseşte şi mai departe (pe poziţiile i+1,…,n-1) – în caz afirmativ se elimină respectiva poziţie.

Descrierea algoritmului în C++:

```
#include<iostream>
#define NMAX 50
using namespace std;
int n,x[NMAX];
void CitireVector(int x[NMAX], int &n){
//se citeste dimensiunea unui vector si elementele sale
 cout<<"Dati dim vect: "; cin>>n;
 cout<<"Dati elementele: ";
 for (int i=0;i<n;i++) cin>>x[i];
}
void AfisareVector(int x[NMAX], int n) {
 cout<<"Elementele vectorului: ";
  for (int i=0;i<n;i++) cout<<x[i]<<" ";
}
void EliminarePoz(int i0, int x[NMAX], int &n) {
//elimin.elem de pe poz i din vect x de dim n (dim se modifica)
 //prin deplasarea elementelor i+1...n-1 cu o pozitie spre st.
 for (int i=i0+1;i<n;i++) x[i-1]=x[i];
 n--; //numarul efectiv de elemente scade cu o unitate
}
void EliminareDuplicate(int x[NMAX], int &n) {
 for (int i=0;i<n-1;i++) //pt fiecare element fara ultimul
  for (int j=i+1;j<n;j++) //ma uit dupa el daca gasim val egala
   if (x[j]==x[i])
    {
     EliminarePoz(j,x,n);
     j--;
    }
}
int main() {
 CitireVector(x,n);
 EliminareDuplicate(x,n);
 cout<<"Dupa eliminarea duplicatelor: "<<endl;
 AfisareVector(x,n);
 return 0;
}
```