5. Stive.

O stivă este o listă la care elementele pot fi inserate şi şterse la un singur capăt (*vârful stivei*).
Elementele sunt şterse în ordine inversă punerii lor în stivă, motiv pentru care o stivă urmează principiul LIFO (**L**ast **I**n **F**irst **O**ut) - ultimul introdus este primul şters. De aceea o stivă este folosită uneori pentru inversarea ordinii unui şir de valori.
Stiva se foloseşte pentru a implementa apelul de funcție şi recursivitatea.

Specificarea TAD Stivă.

Domenii (sorturi): **`Stiva, Elem, int`**

Semnături:
- crează o stivă vidă **`new:           → Stiva`**
- întoarce 1 / 0, după cum stiva este sau nu vidă **`empty :   Stiva → int`**
- returnează numărul elementelor din stivă **`size :    Stiva → int`**
- întoarce vîrful stivei, fără a-l şterge din stivă
- dacă stiva este vidă, se produce eroare **`top :     Stiva → Elem`**
- şterge elementul din vârful stivei şi returnează valoarea acestuia
- dacă stiva este vidă se produce eroare **`pop :     Stiva → Stiva`**
- inserează un element în vârful stivei **`push: Stiva × Elem → Stiva`**

Axiome:
```
empty (new()) = 1
empty (push(S,e)) = 0
top(push(S,e)) = e
top(Stiva_Vida) = eroare
pop(push(S,e)) = S
pop(Stiva_Vida) = eroare
```

Precondiții:
```
pop(S)  ⇒ !empty(S)
top(S)  ⇒ !empty(S)
```

Interfața TAD Stivă.

Fişierul de interfață are forma:
```
// stiva.h - interfata stiva
#ifndef _STIVA_
#define _STIVA_
struct stiva;
typedef struct stiva *Stiva;

//constructor
Stiva S_New(int c); //pentru stiva alocata cu tablou
Stiva S_New();      //pentru stiva alocata cu lista inlantuita

//destructor
void  S_Delete(Stiva *pS);

//lungime stiva
int   S_Size(Stiva S);
```

```
//test stiva vida
int   S_Empty(Stiva S);

//punere in stiva
//    preconditie: stiva neplina
void  Push(Stiva S, void *x);

//scoatere din stiva
//    preconditie: stiva nevida
void  *Pop(Stiva S);

//inspectare element din varf
//    preconditie: stiva nevida
void  *Top(Stiva S);
#endif
```

Aplicații cu stive.

1. Să se calculeze rezistența echivalentă, obținută prin legarea în serie şi în paralel a mai multor rezistențe.
Configurația este descrisă postfixat sub forma: R_1R_2 **`operație`**.

Soluție:

```
#include "stiva.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
   Stiva S;
   S = S_New(20);
   double *pR1, *pR2, *pR;
   char descr[20];
   int i;
   gets(descr);
   for(i=0; i<strlen(descr); i++)
       if(descr[i]=='R'){
          pR = (double*)malloc(sizeof(double));
           scanf("%lf", pR);
           Push(S, pR);
       }
        else
        {  pR1 = Pop(S);
           pR2 = Pop(S);
           pR = (double*)malloc(sizeof(double));
           if(descr[i]=='S')
              *pR = *pR1 + *pR2;
           else
             *pR=*pR1**pR2/(*pR1+*pR2);
           Push(S, pR);
           free(pR1);
           free(pR2);
        }
   pR = Top(S);
```

```
   printf("rezistenta echivalenta = %6.2lf\n", *pR);
   pR = Pop(S);
   free(pR);
   if(!S_Empty(S))
      printf("descriere incorecta\n");
   return 0;
}
```

2. Un palindrom este un şir de caractere, având aceeaşi semnificaţie dacă este citit înainte sau înapoi. Spaţiile şi semnele de punctuaţie sunt ignorate.
Un mod de a testa dacă un şir este palindrom constă în inversarea caracterelor şi compararea cu şirul dat - cele două secvenţe vor trebui să fie identice.
a- scrieţi o funcţie **`int EstePalindrom(char* S)`**, care foloseste o stivă pentru a determina dacă şirul este palindrom.
b- Scrieţi un program care citeste un şir , îi scoate spaţiile şi semnele de punctuaţie, converteşte toate caracterele în litere mici, apelează **`EstePalindrom()`** şi raportează rezultatul.

Soluţie:

```
#include "stiva.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

int EstePalindrom(char* s){
   Stiva S1, S2;
   char *pc, *pc1, *pc2;
   S1 = S_New(100);
   S2 = S_New(100);
   int i, d;
   for(i=0; i<strlen(s); i++){
      pc = (char*)malloc(sizeof(char));
      *pc = s[i];
      Push(S1, pc);
   }
   while(!S_Empty(S1))
      Push(S2, Pop(S1));
   for(i=0; i<strlen(s); i++){
      pc = (char*)malloc(sizeof(char));
      *pc = s[i];
      Push(S1, pc);
   }
   for(d=0; !S_Empty(S1) && !S_Empty(S2); ){
      pc1 = Pop(S1);
      pc2 = Pop(S2);
      if(*pc1!= *pc2)
         d++;
      free(pc1);
      free(pc2);
   }
   return d==0 && S_Empty(S1) && S_Empty(S2);
}
```

```
int main(){
   char *p1 = (char*)malloc(80);
   char *p2 = (char*)malloc(80);
   char *p3 = p2;
   gets(p1);
   //conversie in majuscule retinand numai litere
   while(*p1)  {
      if(isalpha(*p1))
         *p2++ = toupper(*p1);
      p1++;
   }
   if(EstePalindrom(p3))
      printf("este palindrom\n");
   else
      printf("nu este palindrom\n");
}
```

3. O expresie incomplet parantezată conține ca termeni constante reale, separate prin operatori şi paranteze. Obțineți expresia postfixată şi evaluați-o.

Trecerea de la forma infixată a expresiei la cea postfixată presupune operațiile următoare:
a)- un termen se pune în şirul postfixat
b)- o '(' se pune în stiva de operatori
c)- un operator cu prioritate mai mare decât a operatorului din vârful stivei se pune în stivă
d)- un operator cu prioritate mai mică sau egală cu a operatorilor din vârful stivei, îi descarcă pe aceştia în şirul postfixat. Când prioritatea operatorului ajunge mai mare decât a celui din vârf, se procedează ca la c), adică se pune în stivă.
e)- o ')' scoate un operator din stivă şi o '('.

Pentru a forța descărcarea stivei vom scrie întreaga expresie între 2 paranteze.

De exemplu expresia: **A * (B + C) / D - (E + F)**
se scrie mai întâi cu o pereche suplimentară de paranteze exterioare:

```
( A * ( B + C ) / D - ( E + F ) )

          +               +
      (   (           (   (               stiva de
    * *   *   * /   - -   -  -            operatori
(   ( (   (   ( (   ( (   (  ( (

  A     B   C + * D /   E   F+ -          sirul postfixat
```

Soluție: Pentru trecerea la forma postfixată vom utiliza o stivă de operatori **So**. Aceasta este o stivă de şiruri de caractere, pentru a uşura comunicația cu expresiile infixată şi postfixată. Expresia infixată este dată ca un şir de caractere. Expresia postfixată este tot un şir de caractere, dar preferăm să o păstrăm, pentru a uşura evaluarea, într-un tablou de şiruri, în care un element este un termen sau un operator. Gestionarea alocării de memorie este realizată prin 3 funcții:

- `alchr(c)` – alocă memorie pentru un şir format dintr-un caracter (un operator)
- `alstr(p,q)` – alocă memorie pentru un şir delimitst de pointerii p şi q
- `alflt(x)` – alocă memorie pentru un real

Toate aceste funcții întorc un pointer la zona alocată.

Evaluarea expresiei postfixate impune folosirea unei stive de termeni **St**. Aceasta este o stivă de reali.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "Stiva.h"

char  *alchr(char c);
char  *alstr(char *p, char *q);
float *alflt(float x);
int  pri(char op);
int main(){
   Stiva So;           /*stiva operatorilor*/
   Stiva St;           /*stiva termenilor*/
   char  ifx[100];     /*expresia infixata*/
   char  *pfx[20];     /*sirul postfixat*/
   char  *p, *q, *r;
   char c;
   int i;
   float x;
   int l = 0;          /*lungimea sirului postfixat*/
   /*formarea sirului postfixat */
   So = S_New();
   Push(So, "(");
   gets(ifx);
   strcat(ifx, ")");
   for(p=ifx; p; )
      switch(*p){
      case '(':
         r=alchr('('); Push(So, r); p++; break;
      case ')':
         while((c=*(char*)Top(So))!='('){
            r = alchr(c);
            Pop(So);
            pfx[l++] = r;
         }
         Pop(So); p++; break;
      case '+':case '-': case '*': case '/':
         while(pri(*p) >= pri(*(char*)Top(So))){
            r = alchr(*(char*)Top(So));
            Pop(So);
            pfx[l++] = r;
         }
         p++; break;
      default:
         if(isdigit(*p)){
            q = p;
            while(isdigit(*q)) q++;
            if(*q=='.') q++;
```

```
               while(isdigit(*q)) q++;
               r = alstr(p, q);
               pfx[l++] = r;
               p = q; break;
            }
      }
      /*evaluarea sirului postfixat*/
      for(i=0; i<l; i++)
         if(isdigit(pfx[i][0])){
             x = atof(pfx[i]);
             Push(St, &x);
         }
         else
             {  p = (char*)Pop(St);
                q = (char*)Pop(St);
                switch(pfx[i][0]){
                   case '+': x = atof(p) + atof(q); break;
                   case '-': x = atof(p) - atof(q); break;
                   case '*': x = atof(p) * atof(q); break;
                   case '/': x = atof(p) / atof(q); break;
                }
                Push(St, alflt(x));
             }
      x = *(float*)Top(St);
      printf("%7.2f\n", x);
      return 0;
}

char *alchr(char c){
   char *r = (char*)malloc(2);
   r[0] =c;
   r[1] = 0;
   return r;
}
char *alstr(char *p, char *q){
   char *r = (char*)malloc(q-p+1);
   strncpy(r, p, q-p);
   r[q-p] = 0;
   return r;
}
int  pri(char op){
   if(op=='+' || op=='-') return 1;
   if(op=='*' || op=='/') return 2;
   return 0;
}
float *alflt(float x){
   float *f;
   f = (float*)malloc(sizeof(float));
   *f = x;
   return f;
}
```

Implementarea stivelor.

Sunt posibile două abordări:

- *implementatorul lasă în seama utilizatorului gestiunea memoriei*

Aceasta impune utilizatorului, ca înaintea fiecărei operații de punere în stivă:

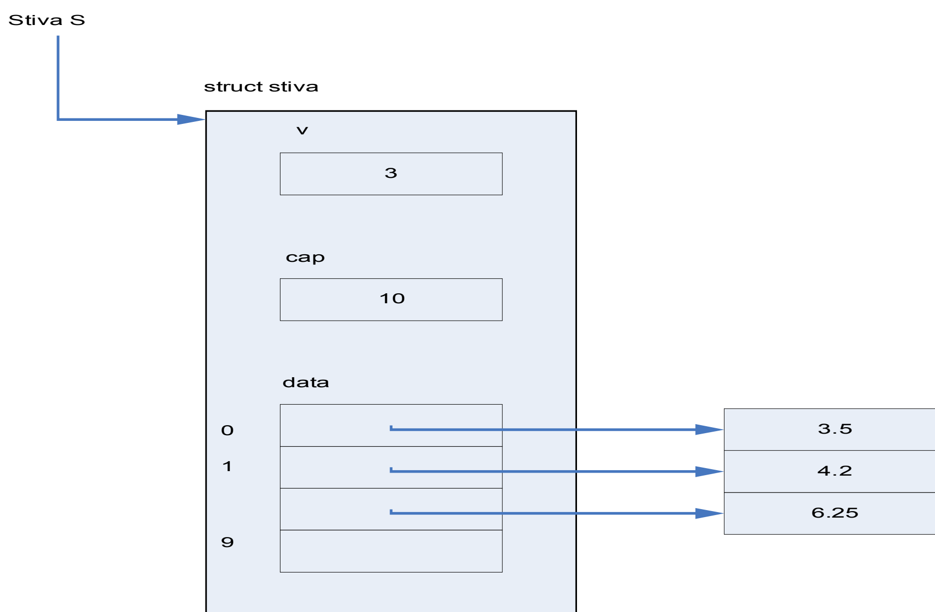
- să aloce dinamic memorie pentru datele care se vor pune în stivă
- să plaseze datele în memoria alocată dinamic

După scoaterea datelor din stivă şi folosirea lor, utilizatorul va elibera memoria alocată dinamic, pentru a preveni ”scurgerea (risipa) de memorie”

- *implementatorul se ocupă şi de gestiunea memoriei*

Pentru a putea aloca şi elibera memorie, implementatorul va trebui să ştie cantitatea de memorie alocată sau eliberată (**sizeof**(`tip`)). Această informație este preluată sub forma unui parametru, transmis, de exemplu prin constructor, la inițializarea tipului abstract de date.

a) **Implementare cu tablouri, cu gestiunea memoriei făcută de utilizator.**



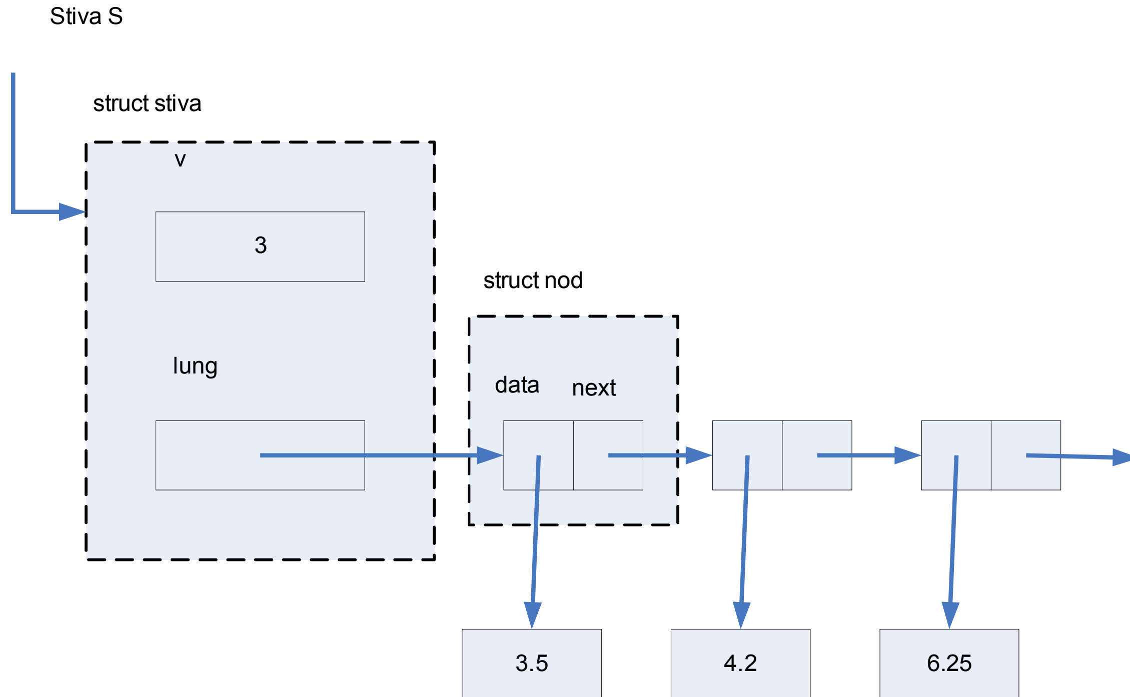
Stivă alocată cu tablou de pointeri

În implementarea cu tablouri vom face distincția între *capacitatea* stivei – dimensiunea maximă a stivei şi *dimensiunea* stivei, care precizează numărul efectiv de elemente din stivă. Constructorul va avea ca parametru capacitatea stivei **c**, şi va aloca pentru stiva vidă **c** pointeri.

```
// stiva.c - implementare stiva cu tablouri
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stiva.h"

// struct stiva{
   int v;         //varf stiva
   int cap;       //capacitate stiva
   void **data;   //tablou de pointeri
};
//varful este plasat deasupra ultimului element
//constructor - aloca c elemente
```

```
Stiva S_New(int c){
   Stiva S;
   S = (Stiva)malloc(sizeof(struct stiva));
   S->cap = c;
   S->v = 0;
   S->data = (void**)malloc(c*sizeof(void*));
   return S;
}

//destructor - elibereaza memoria ocupata de stiva

void S_Delete(Stiva *pS){
   int i;
   for(i=0; i<(*pS)->v; i++)
        free((*pS)->data[i]);
   free((*pS)->data);
   free(*pS);
}

//functie de acces - dimensiune stiva
int S_Size(Stiva S) { return S->v; }

//functie de acces - test stiva vida
int S_Empty(Stiva S){ return S->v == 0; }

//modificator - pune un element in stiva
//   preconditie stiva neplina

void Push(Stiva S, void *x){
   assert(S->v < S->cap);  // verificare preconditie
   S->data[S->v] = x;
   S->v++;
}

//modificator - scoate un element din stiva si-l intoarce
//   preconditie stiva nevida

void *Pop(Stiva S){
   assert(S->v > 0);        //verificare preconditie
   S->v--;
   void *x = S->data[S->v];
   return x;
}

//functie de acces - intoarce elementul din varful stivei
//   preconditie stiva nevida

void *Top(Stiva S){
   assert(S->v > 0);              //verificare preconditie
   return S->data[S->v-1];
}
```

b) Implementare cu liste înlănțită, cu gestiunea memoriei făcută de utilizator.



Stiva alocata dinamic cu lista inlantuita

Se defineşte mai întâi un nod al listei:

```
struct nod {
   void   *data;
   struct nod *next;
};
```

şi structura stivă:

```
struct stiva{
   struct Nod *v;
   int    lung;
};

Stiva S_New (){
   Stiva S = (Stiva)malloc(sizeof(struct stiva));
   S->v = 0;
   S->lung = 0;
   return S;
}

int S_Empty(Stiva S) { return S->lung==0; }

int S_Size(Stiva S){ return S->lung; }

void *Top(Stiva S){
   assert(!S_Empty(S));
   return S->v->data;
}

void Push(Stiva S, void *x){
```

```
   struct nod *nou=(struct nod*)malloc(sizeof(struct nod));
   nou->data = x;
   nou->next = S->v;
   S->v = nou;
   S->lung++;
}
void *Pop(Stiva S){
   assert(!S_Empty(S));
   struct nod *sters = S->v;
   void *x = sters->data;
   S->v = sters->next;
   free(sters);
   S->lung--;
   return x;
}
```

Probleme propuse.

1. Scrieți un program care foloseste o stivă pentru a verifica închiderea corectă a parantezelor într-o expresie. Expresia constă dintr-o linie având până la 80 de caractere şi conține 4 tipuri de paranteze: **`{ }   [ ]   < >   ( )`**

Se consideră că expresia este parantezată corect, dacă la închiderea unei paranteze de un tip ultima paranteză deschisă întalnită să fie de acelaşi tip. Astfel expresia **`{A[B<C><D>(E)F](G)}`** este corecta, în timp ce **`{A[B}]`** nu este corecta.
Programul va citi mai multe expresii şi va determina dacă sunt corect parantezate.
Datele se termină printr-un punct. Se va folosi o stivă de caractere.

2. Folosiți o stivă pentru a simula o maşină simplă de adunat. Programul citeste umere reale şi le pune în stiva. La citirea caracterului '+' se scot numerele din stiva, se adună şi se afisează rezultatul.
În plus maşina recunoaste urmatoarele instrucți uni:
',' - anulează ultima intrare
'.' - anulează toate intrarile (sterge stiva).

3. Scrieți un program având ca intrare o expresie infixată, care are ca ieşire expresia postfixată echivalentă.
Intrarea poate conține numere, variabile, operațiile aritmetice +,-,*,/ şi paranteze.
Expresiile nu sunt complet parantezate, ordinea de evaluare fiind dată de precedența operatorilor.

4. Pentru afişarea verticală a unui număr întreg (câte o cifră pe rând) se poate folosi o stivă în care se vor depune resturile împărțirilor repetate la 10.
Scrieți o funcție avînd ca argument un număr întreg, care îl afişează vertical, folosind o stivă.
Scrieți o funcție recursivă având acelaşi efect ca funcția de mai sus.

5. Pentru afişarea unui număr întreg **`n`** într-o altă bază **`B`** se poate folosi o stivă în care se pun resturile parțiale ale împărțirii repetate a numărului cu **`B`**.
Scrieți un program care citeşte un număr întreg şi pozitiv şi o bază **`2<=B<=9`** şi afişează reprezentarea numărului în baza **`B`**.
Definiți o funcție pentru afişarea rezultatului astfel încât să putem folosi baze între 2 şi 16
Scrieți o funcție recursivă pentru afişarea unui număr întreg zecimal **`n`** în baza **`B`**.

6. Implementați comanda **verifitag numef** care verifică dacă fişierul XML cu numele **numef** are marcajele imbricate corect.
Un fişier XML este un fişier text care conține diverse şiruri de caractere încadrate de marcaje. Un marcaj ("tag") este un şir între parantezele ascuțite '**<**' şi '**>**'. Marcajele se folosesc în perechi: un marcaj de început (ex: **<a>**) şi un marcaj de sfârşit (ex: ****). Perechile de marcaje pot fi incluse unele în altele.
Exemplu: **<a> ... ...** Nu sunt permise construcții de forma: **<a> ... ...**

Indicație: Fişierul poate fi citit complet în memorie sau prelucrarea se poate face linie cu linie. Intâlnirea unui marcaj de început, pune numele marcajului în stivă. La întâlnirea unui marcaj de sfârşit se scoate v ârful stivei şi se compară numele marcajului de sfârşit cu numele marcajului din vârful stivei. Dacă diferă marcajele nu sunt imbricate corect.

7. O expresie simbolica complet parantezata contine ca termini litere, separate prin operatori si paranteze.Pentru a obtine expresia postfixata se procedeaza astfel:
- o litera se pune in sirul postfixat
- un operator sau '(' se pune in stiva de operatori
- o ')' scoate un operator din stiva si-l pune in sirul postfixat si scoate de asemeni din stiva o '('.

In final, daca expresia a fost scrisa corect, stiva de operatori se va goli.
De exemplu expresia:
(((A + B) * (C / D)) - (E * F))

```
                    /
        +       (   (             *
    (   (     * *   *   *     (   (       stiva de
  ( (   (   ( ( (   (   (   - -   -   -
( ( (   (   ( ( (   (   ( ( ( (   (   (       operatori

      A   B +     C   D / *     E   F * -   sir postfixat
```

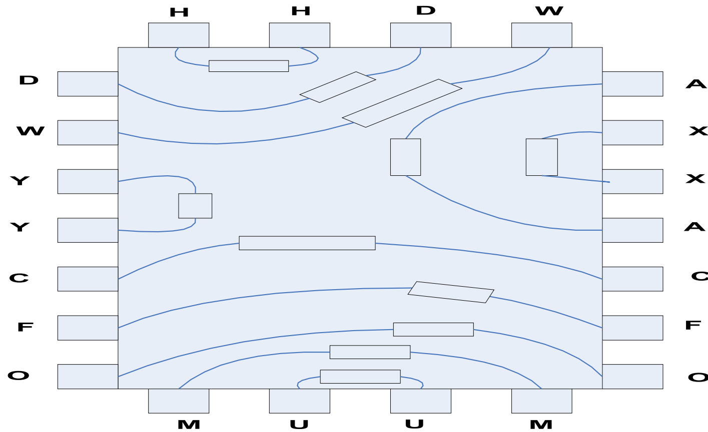
8. Să se evalueze o expresie complet parantezată, ştiind că:
- toți termenii sunt pozitivi
- fiecare operator se aplică numai la doi termeni.

De exemplu expresia complet parantezată **(((23+5)/4)+2)*((38-2)/12)** va fi evaluată la 27.
Evaluarea expresiei presupune folosirea a două stive, una de numere si cealalta de operatori:

```
numar      →  push (stiva de numere)
operator   →  push (stiva de operatori)
'(',' '    →   se ignora
')'        →   pas de evaluare
       n2 ← pop (stiva de numere)
       n1 ← pop (stiva de numere)
       op ← pop (stiva de operatori)
       n = n1 op n2
       n  → push (stiva de numere)
'\n'       →   oprire
```

9. Un circuit integrat de formă dreptunghiulară conține mai multe componente. Fiecare componentă este accesibilă prin două terminale scoase pe laturile circuitului, identificate prin

aceeaşi literă. Scrieţi o funcţie având ca parametru un şir de caractere, care identifică componentele, în ordinea în care acestea sunt dispuse pe circuitul integrat, funcţie care întoarce numărul componentelor care se intersectează. De exemplu pentru circuitul AXBCCDEEDXBA se întoarce 2. Se va folosi o stivă.



6. Cozi.

Coada este o listă la care inserările se fac pe la un capăt – *baza* cozii, iar ştergerile se fac pe la celălalt capăt – *vârful* cozii.

Cozile sunt utile în aplicațiile de simulare, în traversarea grafurilor în lățime, în algoritmi cu arbori şi grafuri.

Ordinea de extragere din coadă este aceeaşi cu ordinea de introducere în coadă, ceea ce sugerează şi aplicațiile pentru o asemenea structură: simularea unor procese de servire de tip *producător - consumator* sau *vânzător - client*. În astfel de situații coada de aşteptare este necesară pentru a acoperi o diferență temporară între ritmul de servire şi ritmul cererilor solicitanților (clienților).

Cozi de servire se pot forma la comunicarea de date între un emițător şi un receptor care au viteze diferite sau la stațiile de servire, pentru a memora temporar mesaje sau cereri de servire care nu pot fi încă satisfăcute, dar care nu trebuie pierdute.

Specificarea TAD Coadă.

Domenii: **Coada, Elem, int**

Semnături:

```
new   :                 → Coada_Vidă
front :           Coada → Elem
back  :           Coada → Elem
enq   :    Coada × Elem → Coada
deq   :           Coada → Coada
empty :           Coada → int
```

Axiome:

```
front(enq(e, Coada_Vida)) = e
front(deq(q, e)) = Front(q),          dacă q nu e vida
front(Coada_Vida) = eroare
deq(enq(Coada_Vida, e)) = Coada_Vida
deq(enq (q, e)) = Enq (Deq(q), e),    dacă q nu e vida
deq(Coada_Vida) = eroare
empty(Coada_Vida) = 1
empty(Enq (q, e)) = 0
```

Operații specifice cozilor.

- Crearea unei cozi vide: **new ()**
- Copierea elementului din vârful cozii fără a-l şterge din coadă; (dacă coada este vidă se produce eroare): **front(Q)**
- Inserarea unui element în coadă: **enq(Q, x)**
- Preluarea elementului de la inceputul cozii şi ştergerea lui din coadă; (dacă coada este vidă se produce eroare): **deq(Q)**
- Test coadă vidă: **empty(Q)**
- Test coadă plină: **full(Q)**
- Inspectarea elementului de la începutul cozii. **front(Q)**
- Inspectarea elementului de la sfârşitul cozii. **back(Q)**

Interfața TAD Coadă.

```
// Fisierul coada.h
#ifndef _Q_H
#define _Q_H
struct coada;
typedef struct coada *Coada;
Coada Q_New(int);              //constructor cu tablou circular
Coada Q_New();                 //constructor cu lista inlantuita
void  Q_Delete(Coada *pQ);     //destructor
int   Q_Empty(Coada  Q);       //test coada vida
int   Q_Full(Coada Q);         //test coada plina
void  Enq(Coada Q, void *x); //pune din coada
void  *Front(Coada Q);         //citeste primul element
void  *Back(Coada Q);          //citeste ultimul element
void  *Deq(Coada Q);           //citeste si sterge
#endif
```

Aplicații cu cozi.

1. Problema lui Josephus: **n** copii se aşează în cerc, se numerotează în sens orar cu `1,2,...,n` şi rostesc o poezie formată din **c** cuvinte (de tipul "ala bala portocala..."). Fiecăruia, începând cu primul i se asociază un cuvânt din poezie. Cel care primeşte ultimul cuvânt este eliminat din cerc. Jocul continuă, începând poezia de la următorul copil, până când se elimină toți copiii. Folosind numerotarea inițială, să se afişeze ordinea ieşirii copiilor din joc.

Rezolvare: Se va folosi o coadă în care se introduc numerele **1,2,...,n**.
Rostirea unui cuvânt din poezie se traduce printr-o permutare circulară (o scoatere a unui element urmată de o inserare a lui în coadă). După **c** permutări circulare, elementul scos nu mai este pus la loc în coadă, ci afişat. Programul se termină în momentul în care coada ajunge vida.

```
#include "coada.h"
#include <stdio.h>
#define MAX 20
int main(){
  char nume[MAX][30];
  int np, i, *pj, *pi, c, n;
  scanf("%d", &n);
  printf("Numele celor %2d persoane\n", n);
  for (i=0; i<n; i++)
     scanf("%s",nume[i]);
  //citeste numar de cuvinte poezie
  scanf("%d", &c);
  //pune persoanele in coada
  Coada Joc = Q_New(n);
  for (i=0; i<n; i++){
     pi = (int*)malloc(sizeof(int));
     *pi = i;
     Enq(Joc, pi);
  }
  while(!Q_Empty(Joc)) {
     for (i=0; i<c-1; i++) {
        pj = Deq(Joc); //permutare circulara
```

```
            Enq(Joc, pj);
        }
        pj = Deq(Joc);    //scoate ultimul din joc
        printf("%s\n", nume[*pj]);
        free(pj);

    }
}
```

2. ***Sortarea pe ranguri (radix sort):*** Pentru a sorta un şir de numere întregi **x** prin metoda "radix sort" se folosesc 10 cozi corespunzătoare cifrelor 0,1,...,9, cozi iniţial vide.
Sortarea are loc în următorii paşi:
1. Pentru fiecare număr din şirul **x** se separă cifra din rangul **r=0,1,2,...**, fie **i** valoarea cifrei si se distribuie numărul în coada **i**.
2. Se concatenează numerele din cozile cifrelor în şirul **x**. Se repetă operaţiile 1 si 2 pentru toate rangurile numerelor.Operaţia se încheie în momentul în care s-au testat toate rangurile. În final în şirul **x** se vor afla numerele sortate.
Exemplu:

```
X: 7295, 136, 24, 965, 12345
Distribuire după rang 0:
     Q4:  24
     Q5:  7295  965  12345
     Q6:  136
Distribuire după rang 1:
     Q2:  24
     Q3:  136
     Q4:  12345
     Q6:  965
     Q9:  7295
Distribuire după rang 2:
     Q0:  024
     Q1:  136
     Q2:  7295
     Q3:  12345
     Q9:  965
Distribuire după rang 3:
     Q0:  0024  0136  0965
     Q2:  12345
     Q7:  7295
Distribuire după rang 4:
     Q0:  00024  00136  00965  07295
     Q1:  12345
```

Scrieţi o funcţie având ca parametru coada principală, care sortează şirul prin metoda descrisă.

Rezolvare: Pentru fiecare rang au loc două operaţii:
- o distribuire a elementelor şirului în cele 10 cozi
- o concatenare a elementelor din cele 10 cozi in şirul de sortat

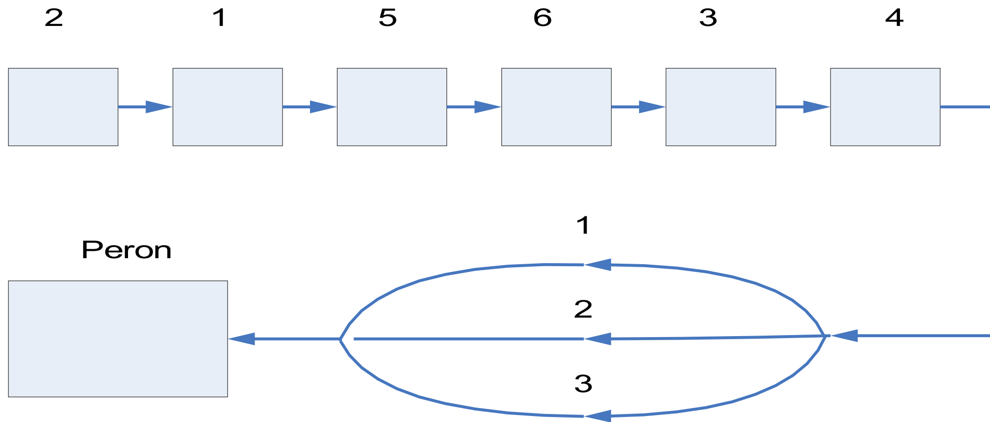
Cifra din rangul `r` a unui număr `n` este: $n / 10^r \% 10$.
Numărul de ranguri testate reprezintă numărul de ranguri al celui mai mare număr din şir.

Funcția **`Distribuie()`** are ca parametri: tabloul de sortat **`X`**, lungimea lui **`n`**, 10 cozi în care se distribuie numerele orespunzător cifrei dintr-un rang dat **`r`**. În momentul în care toate valorile sunt plasate într-o singură coadă, şirul este sortat.

```
#include "QC.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int rangmax(int n, int *X){
   int xm = X[0];
   int k, r;
   for (k=0; k<n; k++)
      if(fabs(X[k]) > xm)
         xm = fabs(X[k]);
   for(r=0; xm > 0; r++,xm/=10)
                    ;
   return r;
}

void Distribuie(int n, int* X, Coada* QC, int r){
   int nc=0, ind, i, j, *pi;
   for(i=0; i<n; i++){
      int t=X[i];
      for(j=0; j<r; j++)
         t /= 10;
      ind = t % 10;
      if(Q_Empty(QC[ind]))
         nc++;
      pi = (int*)malloc(sizeof(int));
      *pi = X[i];
      Enq(QC[ind], pi);
   }
}

void Colecteaza(Coada* QC, int* X){
   int j, i=0, *pi;
   for (j=0; j<10; j++)
      while(!Q_Empty(QC[j])){
         pi = Deq(QC[j]);
         X[i] = *pi;
         i++;
         free(pi);
      }
}


void RadixSort(int n, int* X){
   int rmx, i, r;
   Coada QC[10];
   for(i=0; i<10; i++)
      QC[i]= Q_New (5);
   rmx = rangmax(n, X);
   for (r=0; r<=rmx; r++){
```

```
      Distribuie(n, X, QC, r);
      Colecteaza(QC, X);
   }
}
int main(){
   int n, i;
   int *x;
   scanf("%d", &n);
   x = (int*)malloc(n*sizeof(int));
   for(i=0; i<n; i++)
      scanf("%d", &x[i]);
   RadixSort(n, x);
   for(i=0; i<n; i++)
      printf("%5d  ", x[i]);
   printf("\n");
   free(x);
   return 0;
}
```

3. Într-un depou se află pe o linie **n** vagoane, identificate cu numere, începând de la 1 până la **n**, şi plasate arbitrar. La ieşirea din depou se află un triaj, format prin ramificarea liniei în **p** linii paralele, care se reunesc apon într-o singură linie la peron. Se doreşte formarea unei garnituri la peron, cu toate cele **n** vagoane, cu numere plasate în ordine crescătoare. În acest stop, vagoanele scoase din depou sunt distribuite pe liniile de triaj şi de acolo, la linia de peron.
Precizați comenzile de dirijare pe liniile de triaj de tipul: "Linia **i**!" şi apoi cele de formare a garniturii la peron de tipul "la peron!" . Este posibil ca garnitura să un poată fi formată, caz în care programul afişează "Stop!" şi se opreşte.
Exemplu:



Linia 1	3 la peron
Linia 2	3 la peron
Linia 1	2 la peron
Linia 2	1 la peron
Linia 3	2 la peron
Linia 3	1 la peron

Se dau **n, p** şi numerele celor **n** vagoane din depou. Se vor folosi cozi şi funcțiile specifice acestora:

```
#include <stdio.h>
#include <stdlib.h>
#include "Coada.h"

int main(){
   int tren[20], n, p, i, j, plasat, nlf=0, *v;
   Coada triaj[10];
   scanf("%d%d", &n, &p);
   for(i=0; i<n; i++)
      scanf("%d", &tren[i]);
   for(i=0; i<p; i++)
      triaj[i] = Q_New();
   for(i=0; i<n; i++){
      /* incearca plasarea pe o linie deja folosita */
      plasat = 0;
      for(j=0; j<nlf && !plasat; j++)
         if(*(int*)Back(triaj[j]) < tren[i]){
            plasat = 1;
            Enq(triaj[j], &tren[i]);
            printf("Linia %2d\n", j);
         }
      if(!plasat){
         if(nlf < p){
            nlf++;
            plasat = 1;
            Enq(triaj[nlf-1], &tren[i]);
            printf("Linia %2d\n", nlf);
         }
         else
         {  printf("nu putem plasa\n");
            return 0;
         }
      }
   }
   for(i=0; i<nlf; i++)
      while(!Q_Empty(triaj[i])){
         v = (int*)Deq(triaj[i]);
         printf("%2d la peron\n", *v);
      }
   return 0;
}
```

Implementarea cozilor.

a) Implementare cu tablou circular de lungime fixă.

Coada este păstrată într-un tablou **`data`**, alocat la inițializarea cozii, la o valoare fixată **`c`** şi accesat prin doi indici:
`v` - indicele primului element din coadă (vârful cozii)
`b` - indicele ultimului element din coadă (baza cozii)

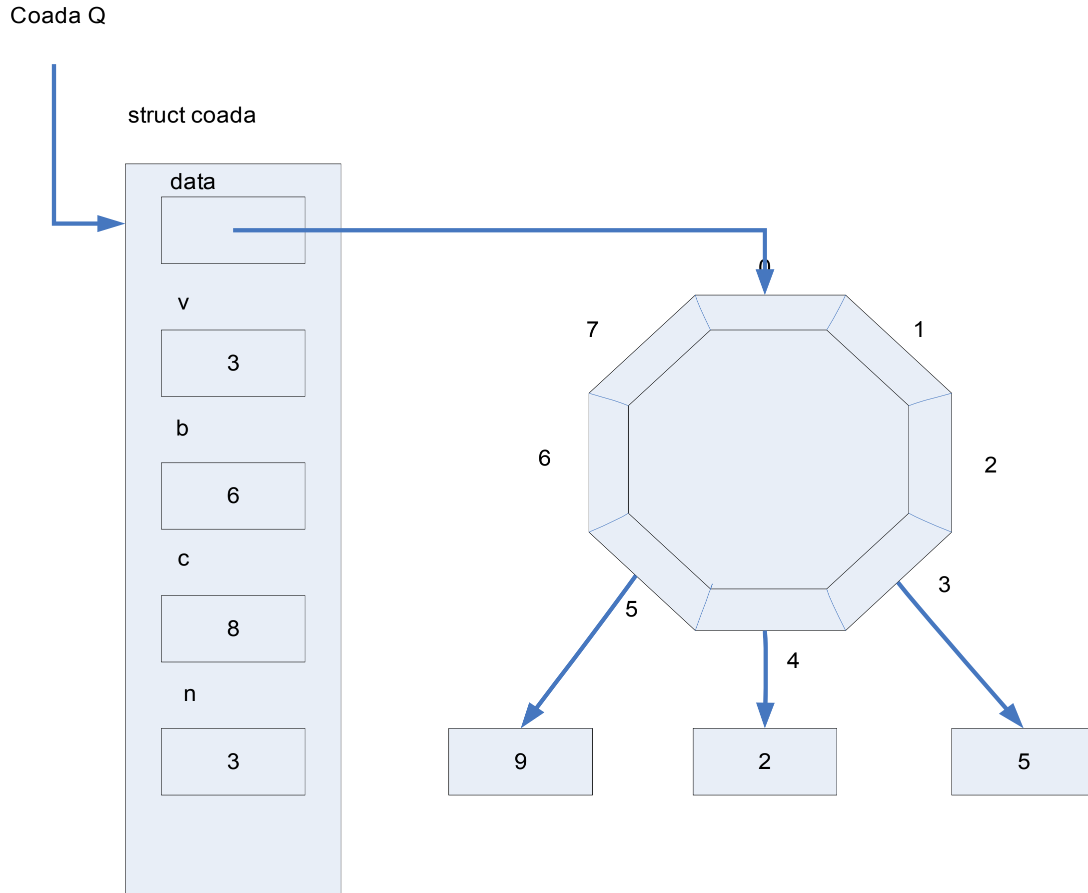
Dacă indicii **`v`** şi **`b`** marchează primul, respectiv ultimul element din coadă, atunci coada cu un element ar fi caracterizată prin condiția **`v==b`**, iar *coada vidă* ar avea **`v`** înaintea lui **`b`**, adică **`avans(b)==v`**; pe de altă parte *coada plină* ar fi caracterizată prin aceeaşi condiție şi nu am putea face distincția între cele două situații.

Pentru a disemina între cele două situații (coadă vidă / coadă plină) unul dintre indici nu va indica un element ci o poziție neocupată (după ultimul element pus sau înaintea primului scos) ceeace ar face coada să nu fie utilizată la întreaga sa capacitate **`c`**.
Putem evita problema de mai sus dacă păstrăm numărul de elemente **`n`** din coadă .
Inițial coada este vidă, având **`v=0, b=0`** şi **`n=0`**
Ştergerea unui element se poate face numai dintr-o coadă nevidă (în caz contrar se produce o eroare) şi se traduce prin **`v++`** şi **`n--`**
Adăugarea unui element este posibilă dacă nu s-a umplut coada şi (**`n < c`**) şi se reprezintă prin **`data[b++]=x`** şi **`n++.`**
Inserarea şi ştergerea repetată de **`c-1`** ori duce la imposibilitatea folosirii cozii, deşi ea este vidă.
Soluția o constituie utilizarea unui tablou circular, la care cei doi indici avansează modulo **`c`**:

```
v=(v+1) % c
b=(b+1) % c
```

Caracterul circular permite reutilizarea locațiilor eliberate prin extragerea unor valori din coadă.
Câmpul **`b`** (*ultim*) conține indicele din vector unde se va adăuga un nou element, iar **`v`** (*prim*) este indicele primului (celui mai vechi) element din coadă.

Memorarea numărului de elemente existente la un moment dat în coadă ne permite să deosebim situațiile de *coadă goală* si *coadă plină*, ambele caracterizate prin valori egale pentru indicii **`v`** şi **`b`**. (dacă nu s-ar păstra **`n`**, pentru a face distincție între cele două situații limită, ar trebui ca indicele **b** să fie situat după ultimul mereu introdus (nu ar indica o valoare) şi capacitatea cozii ar fi **`c-1`** în loc de **`c`**.
Dimensiunea cozii se calculează ca **`(c-v+b)%c`** sau mai simplu **`n`**.
Operațiile se realizează în timp constant (complexitate **`O(1)`**).



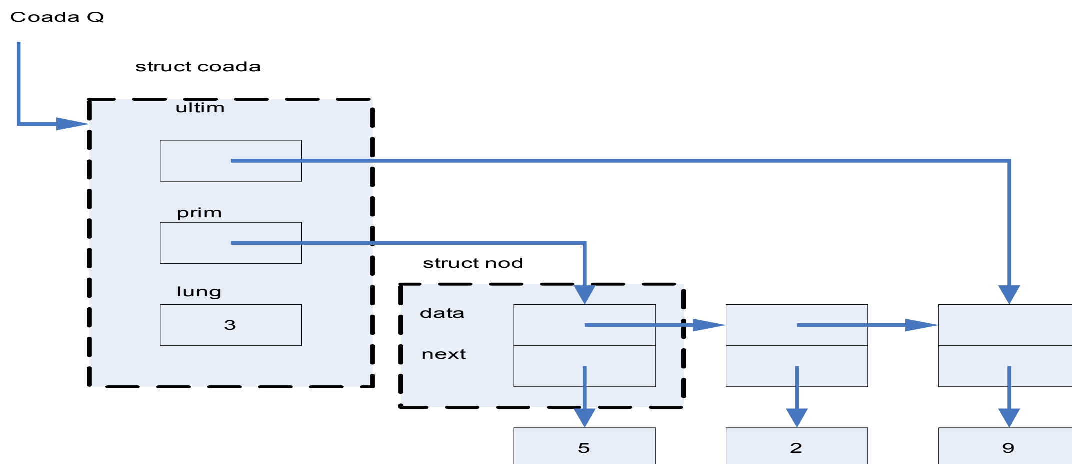
Coada implementata cu tablou circular

```
//Implementare coada cu tablou circular Coada.c
#include "Coada.h"
#include <assert.h>
#include <stdlib.h>

struct coada {
   int v;          //varf (inceput) coada
   int b;          //baza (sfarsit) coada
   int n;          //numar de elemente din coada
   int c;          //dimens.memorie alocata
   void **data;    //tablou elemente
};

Coada Q_New(int c){
   Coada Q;
   Q=(Coada)malloc(sizeof(struct coada));
   Q->v = Q->b = Q->n = 0;
   Q->c = c;
   Q->data = (void**)malloc(c*sizeof(void*));
   return Q;
}
```

```
int Q_Empty(Coada Q) { return Q->n == 0; }

int Q_Full(Coada Q) { return Q->n == Q->c; }

void Enq(Coada Q, void *x){
   assert(!Q_Full(Q));
   Q->data[Q->v++] = x;
   if(Q->v == Q->c)
      Q->v = 0;
   Q->n++;
}

void *Deq(Coada Q){
   assert(!Q_Empty(Q));
   void *x = Q->data[Q->b++];
   if(Q->b == Q->c)
      Q->b = 0;
   Q->n--;
   return x;
}

void *Front(Coada Q){
   assert(!Q_Empty(Q));
   return Q->data[Q->b];
}
```

b) Implementare cu listă înlănţuită.

Am folosit aceeaşi convenţie ca şi în cazul stivelor: constructorul cozii are ca parametru capacitatea cozii pentru implementarea cozii cu tablou circular şi nu are parametri, la implementarea cozii cu listă înlănţuită.



Implementarea cozii cu lista inlantuita

```
#include "Coada.h"
#include <stdlib.h>
#include <assert.h>

struct nod{
   void    *data;
```

```
   struct  nod *next;
};

struct coada{
   int lung;
   struct nod *prim;
   struct nod *ultim;
};

Coada Q_New (){
   Coada Q=(Coada)malloc(sizeof(struct coada));
   Q->lung=0;
   Q->prim=Q->ultim=0;
   return Q;
}

int Q_Size(Coada Q){ return Q->lung; }

int Q_Empty(Coada Q){ return Q->lung==0; }

void *Front(Coada Q){
   assert(!Q_Empty(Q));
   return Q->prim->data;
}

void *Back(Coada Q){
   assert(!Q_Empty(Q));
   return Q->ultim->data;
}

void Enq(Coada Q, void *x){
   struct nod *nou = (struct nod*)malloc(sizeof(struct nod));
   nou->data = x;
   nou->next = 0;
   if(Q->ultim==0)
      Q->prim = nou;
   else
      Q->ultim->next = nou;
   Q->ultim = nou;
   Q->lung++;
}

void *Deq(Coada Q){
   assert(!Q_Empty(Q));
   void *x = Q->prim->data;
   struct nod *sters = Q->prim;
   Q->prim = Q->prim->next;
   if(Q->prim==0)
      Q->ultim = 0;
   free(sters);
   Q->lung--;
   return x;
}
```

Probleme propuse.

1. O coadă conține elementele $a_1, a_2, ..., a_n$ cu a_1 în fruntea cozii şi a_n în spatele cozii. Scrieți un algoritm cu complexitate $O(n)$ care plasează aceste elemente într-o stivă, cu a_1 în vârful stivei şi a_n în baza stivei, păstrând ordinea relativă a elementelor. Se vor folosi numai operațiile specifice tupurilor de date abstracte stivă şi coadă.

Dintr-un fişier text, să se afişeze liniile care reprezintă palindroame. In acest scop se pun caracterele litere dintr-o linie, convertite în majuscule, într-o stivă şi într-o coadă. Se descarcă stiva şi coada şi se numără diferențele. Dacă nu există diferențe, am găsit un palindrom.