Laborator Algoritmi și Structuri de Date
Tema 8

Tema săptămânii 8.

Stive & Cozi

Atenție! Problemele la stive și cozi se vor rezolva toate utilizând implementări bazate pe liste!

Mare atenție! Din cauză că avem operația `pop` (= ștergerea unui element), suntem obligați ca ștergerea să o facem de la începutul listei, pentru că este singurul caz în care ștergerea este în $O(1)$ timp constant .

Deci pentru stivă `push` va fi adăugare la început în listă, `pop` va fi ștergere (super simplificată, doar de la început - două cazuri particulare, ori stiva e vidă, ori am ce să șterg).

Pentru coadă ne trebuie și pointer `ultim` ca să facem `push` la final în listă (care de fapt reprezintă adăugare la început în coadă). `pop` va rămâne la fel ca și la stivă. Pentru stive ajunge doar `prim`.

Precizare: `pop` pe stivă sau coadă vidă trebuie să întoarcă o valoare specială definită anterior, de exemplu -1. Pentru ca în momentul când implementăm programe cu stive și cozi să putem testa cu ușurință cazul când structura e vidă, folosind doar `pop`.

1 Stive și cozi, implementări cu vectori(array)

Aceste implementări bazate pe vectori sunt de preferat când dimensiunea problemei este mică din cauză ca sunt mai rapide decât cele dinamice cu liste.

Ele sunt prezentate aici cu scop orientativ.

Sunt si foarte ușor de implementat, cel puțin implementarea pentru stivă este imediată.

1.1 Stivă, implementare clasică cu vector

Un vector în care ținem minte vârful stivei folosind indicele top.

Baza stivei e localizată la indicele 0, iar vârful este către dreapta. Orice operație de inserție sau extragere se face doar în vârf.

```
1  #define MAX_STACK_SIZE 100
2  int stiva[MAX_STACK_SIZE];
3  int top=0;
4  void push (int x){
5    if (top == MAX_STACK_SIZE)
6      {cout<<"STACK_OVERFLOW"; return;}
7    stiva[top]=x;
8    top++;
9  }
10 int pop(){
11   if (top==0)
12     {cout<<"STACK_UNDERFLOW"; return -1;}
13   top--;
14   return stiva[top];
15 }
16 int peek(){
17   if (top==0) {cout<<"STACK_UNDERFLOW";return -1;}
18   return stiva[top-1];
19 }
```

1.2 Coadă, implementare (naivă) cu vector

Folosim doi indici în vector pentru a ține minte la ce indice în vector se găsește capătul la care inserăm și capătul de la care extragem.

De ce implentare naivă? pentru că suportă maximum 100 inserții după care, indiferent dacă există mai puțin de 100 de elemente în coadă, nu se mai poate insera.

```
1  #define MAX_QUEUE_SIZE 100
2  int queue[MAX_QUEUE_SIZE];
3  int left=0, right=0; //left for OUT, right for IN
4  void push (int x){
5    if (right == MAX_QUEUE_SIZE)
6      {cout<<"QUEUE_OVERFLOW"; return;}
7    queue[right]=x;
8    right++;
9  }
10 int pop(){
11   if (left==right)
12     {cout<<"QUEUE_UNDERFLOW"; return -1;}
13   left++;
14   return queue[left-1];
15 }
16 int peek(){
17   if (left==right) {cout<<"QUEUE_UNDERFLOW";return -1;}
18   return queue[left];
19 }
```

1.3 Coadă, implementare clasică cu vector

O variantă îmbunatățită în care o luăm de la început în vector când capătul drept depășește dimensiunea maximă a vectorului. Eventualele operații de eliminare din coadă vor fi mutat capătul stâng al cozii ca să permită inserarea de noi elemente la începutul vectorului. În acest mod coada se tot "deplasează circular" prin vector.

Coada se umple în momentul când suntem aproape de a "încăleca dreapta peste stânga". Deși mai există un loc liber pentru 1 element, în această implementare nu se poate adăuga din cauză că se face confuzie cu cazul în care coada este vidă. Deci de fapt aici, capacitatea cozii este -1 față de întregul specificat.

```
1  #define MAX_QUEUE_SIZE 100
2  int queue[MAX_QUEUE_SIZE];
3  int left=0, right=0; //left for OUT, right for IN
4  void push (int x){
5    if (right == (left-1+MAX_QUEUE_SIZE)%MAX_QUEUE_SIZE)
6  //pentru ca -1 % 100 nu e ok, vrem ca -1 sa devina 99
7      {cout<<"QUEUE_OVERFLOW"; return;}
8    queue[right]=x;
9    right=(right+1)%MAX_QUEUE_SIZE; //100 -> 0
10 }
11 int pop(){
12   if (left==right)
13     {cout<<"QUEUE_UNDERFLOW"; return -1;}
14   int ret=queue[left];
15   left=(left+1)%MAX_QUEUE_SIZE; //100 -> 0
16   return ret;
17 }
```

1.4 (facultativ) Coadă, implementare cu vector pentru smecheri

Se pot înlocui operațiile de adunare modulo dimensiune (practic este aritmetică în Z_{100}, în exemplul considerat) cu operații de adunare + "și pe biți" dacă dimensiunea este putere de 2. De fapt, după fiecare operație, se iau doar ultimii p biți în considerație, pentru dimensiunea egală cu 2^p.

E o implementare care pune accent pe viteză pentru probleme care conțin calcul intensiv (se înlocuiește împărțirea pe întregi cu & pe biți care este mult mai rapid).

```
1  #define MAX_QUEUE_SIZE 256
2    //daca dimensiunea maxima e putere de 2
3    //se poate folosi & (pe biti) in loc de % (modulo)
4    //cand adunam 1 sau scadem, e suficient sa luam
5    //ultimii (puterea lui 2, aici 8) biti
6  #define BITMASK 255
7    // 32b= ....0000 0000 1111 1111
8  int queue[MAX_QUEUE_SIZE];
9  int left=0, right=0; //left for OUT, right for IN
10 void push (int x){
11   if (right == (left-1)&BITMASK)
12     {cout<<"QUEUE_OVERFLOW"; return;}
13   queue[right]=x;
14   right=(right+1)&BITMASK;
15     // 256 = .... 0000 0001 0000 0000
16     //     & .... 0000 0000 1111 1111
17     // =0
18 }
19 int pop(){
20   if (left==right)
21     {cout<<"QUEUE_UNDERFLOW"; return -1;}
22   int ret=queue[left];
23   left=(left+1)&BITMASK;
24   return ret;
25 }
```

2 Implementare la stivă și coadă

(2p) 1. Să se implementeze cu alocare dinamică (folosind o listă simplificată) o **stivă** de numere întregi, cu următoarele operații (implementate prin funcții similare ca și declarație cu cele de la implementările de sus):

push. adăugarea unui nou element în vârful stivei;

pop. eliminarea elementului din vârful stivei dacă există și întoarcerea lui; dacă stiva e vidă se întoarce -1;

peek. întoarcerea elementului din vârful stivei dacă există fără eliminare; dacă stiva e vidă se întoarce -1;

empty. funcție care întoarce adevărat dacă stiva este vidă;

search. funcție care parcurge stiva dinspre vârf pentru a vedea dacă conține un element dat ca argument; dacă elementul se găsește funcția întoarce distanța de la vârful stivei (elementul din vârf e la distanță 0), altfel se întoarce -1;

afișare. funcție care afișează stiva, precizând convenabil și care e baza și care e vârful (prin orice metodă inteligibilă);

Exemplu:

afisare()	stiva vida
push(1)	
push(2)	
push(3)	
afisare()	varf 3 2 1 baza
cout<<search(2)	1
cout<<search(4)	-1
cout<<empty()	0
cout<<pop()	3
afisare()	varf 2 1 baza
cout<<peek()	2
afisare()	varf 2 1 baza
pop()	
pop()	
afisare()	stiva vida
cout<<pop()	-1
cout<<empty()	1

(2p) 2. Să se implementeze cu alocare dinamică (folosind o listă simplificată) o **coadă** de numere întregi, cu următoarele operații (implementate prin funcții similare ca și declarație cu cele de la implementările de sus):

push. adăugarea unui nou element în coadă;

pop. eliminarea celui mai vechi element adăugat dacă există și întoarcerea lui; dacă coada e vidă se întoarce -1;

peek. întoarcerea elementului cel mai vechi din coadă dacă există fără eliminare; dacă coada e vidă se întoarce -1;

empty. funcție care întoarce adevărat dacă coada este vidă;

search. funcție care parcurge coada pentru a vedea dacă conține un element dat ca argument; dacă elementul se găsește funcția întoarce numărul de `pop` necesar pentru a-l scoate (elementul cel mai vechi este primul în ordinea ieșirii, deci e nevoie de 1 `pop` pentru a-l scoate), altfel se întoarce -1;

afișare. funcție care afișează coada, precizând convenabil care sunt capetele (prin orice metodă inteligibilă);

Exemplu:

afisare()	coada vida
push(1)	
push(2)	
push(3)	
afisare()	out 1 2 3 in
cout<<search(2)	2
cout<<search(4)	-1
cout<<empty()	0
cout<<pop()	1
afisare()	out 2 3 in
cout<<peek()	2
afisare()	out 2 3 in
pop()	
pop()	
afisare()	coada vida
cout<<pop()	-1
cout<<empty()	1

3 Probleme la stive

(1p) 3. Să se determine dacă un șir de paranteze citit se închide corect (fiecare paranteză închisă corespunde corect unei paranteze deschise) utilizând o stivă (cu liste). Pentru fiecare paranteză deschisă se face `push` în stivă, fiecare paranteză închisă face `pop`.

Au semnificație specială: `pop` în stiva vidă, stivă nevidă după citirea tuturor parentezelor de la intrare.

După implementare, observați că stiva poate conține doar paranteze deschise, din care cauză putem să o reprezentăm mai compact ținând minte un contor care reprezintă lungimea stivei (numărul de paranteze deschise).

Exemple:

(()(())) corect

(()()))(() incorect

(2p+1) 4. Fie un număr par n de țăruși distribuiți echidistant pe un cerc numerotați de la 1 la n (vezi figura 1). Țărușii trebuie legați cu fire metalice în așa fel încât aceste fire să nu se intersecteze.

Firele sunt reprezentate prin numere întregi, iar o soluție pentru problemă este dată printr-un vector de n întregi care reprezintă id-ul fiecărui fir asociat fiecărui țăruș. Doi țăruși i și j sunt conectați dacă au în dreptul lor același întreg (`pereche[i]==pereche[j]`).

Determinați în timp $O(n)$ dacă o soluție dată este corectă sau nu.

Atenție că pentru a determina dacă un fir a mai apărut anterior, e nevoie să-l căutam în $O(n)$ ceea ce mărește timpul; alternativ se poate folosi un vector de apariții; dar se poate determina dacă soluția este corectă și fără (+1p).

Exemplu1: Pentru n = 8 si vectorul pereche = (1, 2, 2, 1, 3, 3, 4, 4) avem configuratia valida din Figura 1(b).

Exemplu2: Pentru n = 8 si vectorul pereche = (1, 2, 2, 3, 1, 4, 3, 4) avem configuratia invalida din Figura 1(c).

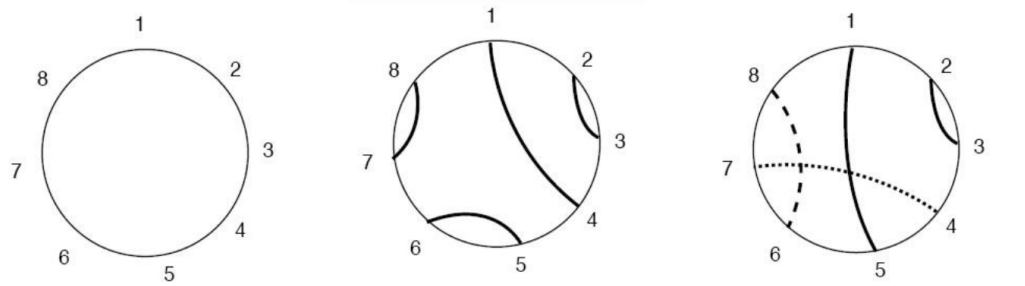

Figure 1: Exemplu problema pinilor

4 Probleme la cozi

(3p) 5. Spunem ca o imagine digitala binara M este o matrice de m x m elemente (pixeli) 0 sau 1. Un element a al matricei este adiacent cu b, daca b se afla deasupra, la dreapta, dedesubtul, sau la stanga lui a in imaginea M.

Spunem ca doi pixeli 1 adiacenti apartin aceleiasi componente.

Problema va cere sa etichetati pixelii imaginii astfel incat doi pixeli primesc aceeasi eticheta daca si numai daca apartin aceleiasi componente.

Hint: folositi o coadă sau o pereche de cozi ca să introduceți coordonatele i și j ale primei apariții de 1. După care, câtă vreme coada nu e vidă scoateți coordonatele, colorați și puneți toți vecinii care au 1 în coadă. Repetare pentru fiecare 1 rămas în matrice.

(Informatie de cultura generala) Exista o structura numita padure de multimi disjuncte (disjoint set forest) descrisa in Cormen, care permite producerea colorarii prin doua treceri prin matrice: prima verifica doar vecinii de sus si stanga a fiecarui element, iar a doua trecere recoloreaza fiecare element in culoarea lui finala (+4p).

(3p) 6. Un depou feroviar consta dintr-o linie ferata de intrare, k linii auxiliare de depozitare, si o linie de iesire. Fiecare linie opereaza pe un sistem de coada (FIFO). In plus, vagoanele se pot deplasa doar dinspre linia de intrare spre linia se iesire (Figura 3).

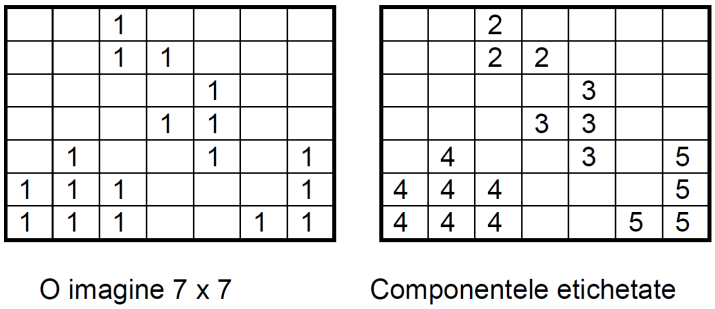

Figure 2: Exemplu problema imagine binara

Sa se scrie un program care, dat un sir de vagoane pe linia de intrare (numerotate de la 1 la n si aranjate in orice ordine), descrie o strategie de a obtine pe linia de iesire sirul de vagoane n; $n-1$;... ; 2; 1, folosind liniile de depozitare.

In caz ca nu exista o astfel de strategie, se va afisa acest lucru.

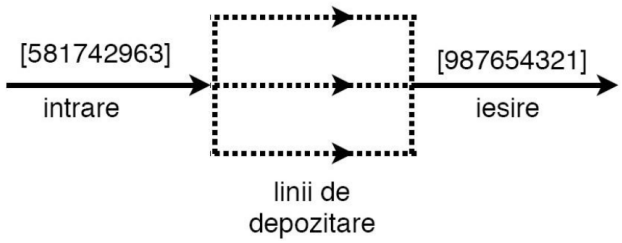

Figure 3: Exemplu problema vagoane

5 Problemă facultativă

(+4p) 7. Fie v un vector de n componente intregi, neordonate.

Spunem ca un element x este **majoritar** in v daca apare de cel putin $n/2+1$ ori in v.

Descrieti si implementati un algoritm ce ruleaza in timp $O(n)$ care sa decida daca exista un element majoritar, si, daca da, sa il afiseze.