laborator 4

>> Structuri liniare III (cu restricţii) – stive şi cozi

Conţinut

- Structuri liniare cu restricţii la operaţiile de inserare şi ştergere
- Stive
- Stive în alocare statică şi în alocare dinamică
- Cozi
- Cozi în alocare statică şi în alocare dinamică
- Cozi circulare în alocare statică

Referinţe

- **T.H. Cormen, C.E. Leiserson, R.L. Rivest.** *Introducere în algoritmi: cap 11.1*, Editura Computer Libris Agora, 2000 (şi ediţiile ulterioare)
- **R. Ceterchi.** *Materiale de curs: curs 3*, Anul universitar 2012-2013
- **http://laborator.wikispaces.com/**, *Tema 4*

Structuri liniare cu restricţii la operaţiile de inserare şi ştergere

Acestea sunt structuri de date în care locul în care se inserează un element nou şi locul în care se poate efectua o ştergere sunt prestabilite. În cazul unei **stive** inserările se vor face mereu în *vârful* stivei, iar elementul şters va fi întotdeauna vârful stivei (cel mai recent inserat). Spunem că stiva implementează principiul *last-in, first-out* (LIFO). Similar, într-o **coadă** elementul şters va fi primul element al cozii, dar inserările se fac mereu la sfârşit. Deci ştergerea într-o coadă se face pe elementul „cel mai vechi", cel care se află de cel mai mult timp în coadă. Coada implementează principiul *first-in, first-out* (FIFO).

Prin vârful stivei *ne referim la primul element/capul stivei.*

1. Stive (Stacks)

„ultimul introdus este primul extras"

Vom spune că primul element al stivei este vârful stivei („top"), iar ultimul element reprezintă *baza* stivei. Inserările şi ştergerile se fac întotdeauna într-un loc comun, pe vârful stivei.

Independent de varianta aleasă pentru implementare, procedurile de inserare şi de ştergere vor urma acelaşi mod de gândire. Precizăm şi că privim stiva ca o structură ce nu poate reţine decât un număr maxim (`max`) de elemente.

Operaţia de inserare, denumită `push`, va încerca să insereze elementul nou pe prima poziţie doar dacă nu s-a atins deja numărul maxim de elemente. Dacă inserarea nu se poate efectua, spunem că ne aflăm într-o situaţie de **overflow** (s-a încercat o inserare într-o stivă plină).

*Cazul de **overflow** se mai numeşte şi supradepăşire sau depăşire superioară. În mod normal el reprezintă o eroare.*

►► PUSH(stiva, val)

```
1. dacă stiva nu este plină atunci
2.   inserează valoarea val pe prima poziţie din stivă
3. altfel
4.   anunţă depăşirea numărului maxim de elemente
```

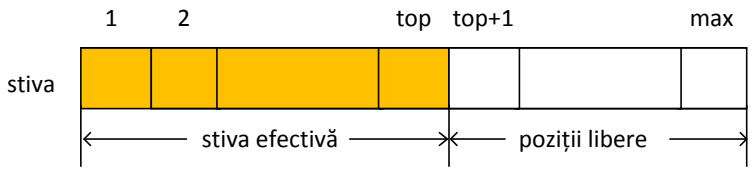
Operaţia de ştergere, denumită `pop`, va încerca să extragă elementul de pe prima poziţie doar dacă stiva conţine elemente. Dacă ştergerea nu se poate efectua, spunem că ne aflăm într-o situaţie de **underflow** (s-a încercat o extragere dintr-o stivă goală).

*Cazul de **underflow** se mai numeşte şi subdepăşire sau depăşire inferioară. În mod normal el reprezintă o eroare.*

►► POP(stiva, x)

```
1. dacă stiva nu este vidă atunci
2.   extrage în x elementul de pe prima poziţie din stivă
3. altfel
4.   anunţă depăşirea numărului minim de elemente
```

1.1. Stive în alocare statică (folosind vectori)



Remarcăm că o consecinţă a implementării cu vectori este faptul că trebuie să reţinem poziţia care reprezintă *vârful stivei*, adică `top`. Toate procedurile pe stivă trebuie să cunoască şi valoarea lui `top` pentru a putea efectua operaţia. Se transmit deci, atât colecţia `stiva` de elemente, cât şi poziţia unde se află vârful, în procedurile pe stivă.

Verificarea dacă o stivă este vidă se poate face prin următorul algoritm.

►► EST–VID(stiva, top)

```
1. if top = 0 then
2.   stiva este vidă
3. else
4.   stiva nu este vidă
5. endif
```

Algoritmul de inserare

►► PUSH(stiva, top, val)

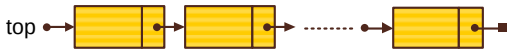
```
1. if top ≠ max then
2.   top ← top+1
3.   stiva[top] ← val
4. else
5.   overflow
6. endif
```

Algoritmul de ştergere

►► POP(stiva, top, x)

```
1. if top ≠ 0 then
2.   x ← stiva[top]
3.   top ← top-1
4. else
5.   underflow
6. endif
```

1.2. Stive în alocare dinamică (folosind liste simplu înlănţuite)



Observăm că pointerul `top` către capul listei îl înlocuieşte pe `first`. Pentru verificarea situaţiei de **overflow** avem două variante:

1. Putem să nu impunem nicio restricţie de dimensiune listei `stiva`, şi să considerăm că ea poate creşte atât cât îi permite spaţiul disponibil în memorie. Dacă nu mai există spaţiu în memorie, atunci alocarea de spaţiu pentru nodul `nou` (linia 1), va returna **`NIL`** . Atunci verificarea de supradepăşire devine:

   ```
   if nou = NIL then
   ```

   Acest caz este cel dat în procedura `push` de mai jos.

2. Putem să reţinem într-o variabilă `n` numărul de noduri din listă, şi să considerăm situaţia de overflow atunci când `n` depăşeşte numărul maxim `max`. În acest caz, verificarea de supradepăşire ar fi:

   ```
   if n = max then
   ```

   Remarcaţi şi că în acest caz trebuie să transmitem funcţiei `push` şi numărul `n` de noduri, şi să incrementăm `n` cu o unitate, de fiecare dată când adăugăm un element în listă, respectiv să îl decrementăm cu o unitate, atunci când facem o extragere.

Amintiţi-vă că operatorul `new` din C++ alocă un spaţiu de o anumită dimensiune şi returnează adresa din memorie de unde începe zona alocată. Dacă alocarea eşuează (nu mai există spaţiu) va returna `null`

►► PUSH(top, val)

```
1.  creează nod nou
2.  if nou ≠ NIL then
3.    nou->info ← val
4.    nou->next ← top
5.    top ← nou
6.  else
7.    overflow
8.  endif
```

►► POP(top, x)

```
1.  if top ≠ NIL then
2.    x ← top->info
3.    temp ← top
4.    top ← top->next
5.    distruge temp
6.  else
7.    underflow
8.  endif
```

2. Cozi (Queues)

„primul introdus este primul extras"

Vom folosi termenul de „front" pentru a ne referi la primul element al cozii, respectiv de „rear" pentru ultimul element. Inserările se fac întotdeauna în spate, pe ultimul element (rear), iar ştergerile se fac întotdeauna în faţă, pe primul element (front).

Independent de varianta aleasă pentru implementare, procedurile de inserare şi de ştergere vor urma acelaşi mod de gândire. Precizăm şi că privim coada ca o structură ce nu poate reţine decât un număr maxim (`max`) de elemente.

Operaţia de inserare, denumită `push`, va încerca să insereze elementul nou la sfârşitul cozii doar dacă nu s-a atins deja numărul maxim de elemente. Dacă inserarea nu se poate efectua, spunem că ne aflăm într-o situaţie de **overflow** (s-a încercat o inserare într-o coadă plină).

►► PUSH(coada, val)

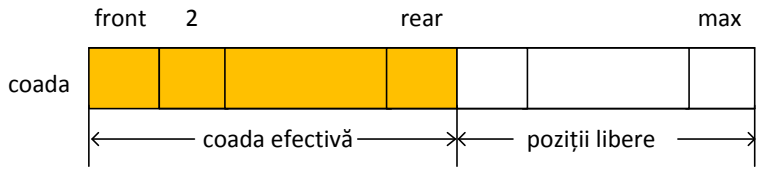
```
1. dacă coada nu este plină atunci
2.    inserează valoarea val la sfârşitul cozii
3. altfel
4.    anunţă depăşirea numărului maxim de elemente
```

Operaţia de ştergere, denumită `pop`, va încerca să extragă elementul de pe prima poziţie doar dacă structura de coadă conţine elemente. Dacă ştergerea nu se poate efectua, spunem că ne aflăm într-o situaţie de **underflow** (s-a încercat o extragere dintr-o coadă goală).

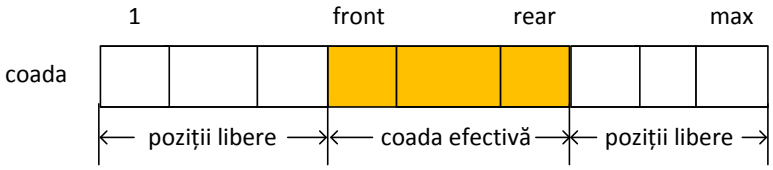
►► POP(coada, x)

```
1. dacă coada nu este vidă atunci
2.    extrage în x elementul de pe prima poziţie din coadă
3. altfel
4.    anunţă depăşirea numărului minim de elemente
```

2.1 Cozi în alocare statică (folosind vectori)



Dacă efectuăm doar inserări în coadă, atunci coada va arăta ca în imaginea anterioară. Însă, cum extragerile se fac în faţă, coada poate arăta în modul următor:



Din imaginea de mai sus reiese că nu avem garanţia că primul element al cozii se află pe poziţia `1` din vector, din acest motiv trebuie să reţinem în `front` poziţia de unde încep practic elementele din coadă. În plus dacă ne uităm la testul de supradepăşire (linia `1` în procedura `push`), observăm şi că în momentul în care ajungem la overflow putem să nu avem `max` elemente în coadă. La orice moment, în acest tip de coadă implementat cu vectori, vom aveam `rear - front + 1` elemente.

Iniţial, când coada este vidă vom avea: `front = rear = 0`.

►► PUSH(coada, front, rear, val)

```
1.  if rear ≠ max then
2.    rear ← rear+1
3.    coada[rear] ← val
4.    if front = 0 then
5.      front = rear
6.    endif
7.  else
8.    overflow
9.  endif
```

►► POP(coada, front, rear, x)

```
1.  if front ≠ 0 and front ≠ (max + 1) then
2.    x ← coada[front]
3.    front ← front+1
4.  else
5.    underflow
6.  endif
```

Am menţionat că în coadă sunt `rear - front + 1` elemente, deci putem utiliza următorul algoritm de verificare dacă o coadă este vidă.

►► ESTE–VIDĂ(coada, front, rear)

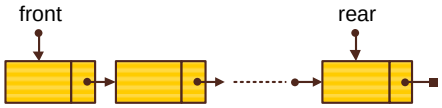
```
1.  if rear = 0 or rear - front + 1 = 0 then
2.    coada este vidă
3.  else
4.    coada nu este vidă
5.  endif
```

Dacă folosim algoritmii de mai sus, atunci după `max` inserări şi extrageri obţinem `rear = max` şi `front = max + 1`. Atunci coada este vidă, dar nu mai putem folosi structura. Putem evita acest lucru dacă adăugăm o reiniţializare în algoritmul de ştergere, atunci când s-a şters ultimul element.

►► POP(coada, front, rear, x)

```
1.  if front ≠ 0 and rear - front + 1 ≠ 0 then
2.    x ← coada[front]
3.    front ← front+1
4.    IF front = max + 1 then
5.      front ← 0
6.      rear ← 0
7.  else
8.    underflow
9.  endif
```

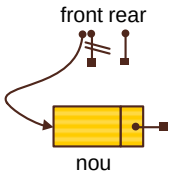
2.2 Cozi în alocare dinamică (folosind liste simplu înlănţuite)



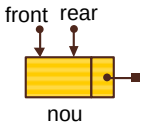
Observăm că:

1. dacă `front = rear =` **`NIL`** , atunci coada este vidă, şi
2. dacă `front = rear` $\neq$ **`NIL`** , atunci coada are un singur element.

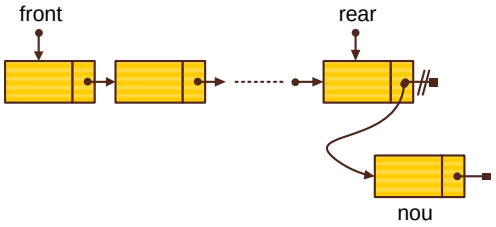
Raţionamentul pentru cazul de overflow este acelaşi ca la stivele în alocare dinamică. Liniile 2–10 reprezintă inserarea propriu-zisa. Dacă `rear =` **`NIL`** este clar că ne aflăm în prima din cele două situaţii de mai sus (coada este vidă) şi inserăm primul element în coadă. Legătura creată la linia 6 (`front` $\leftarrow$ `nou`) este cea din imaginea următoare:



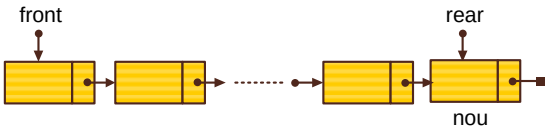
După ce se mută şi pointerul `rear` prin instruţiunea de la linia 10, obţinem:



Pe de altă parte, dacă `rear` $\neq$ **`NIL`** , atunci se va crea legătura `rear->next` $\leftarrow$ `nou`:



Astfel, efectuăm mereu inserări la sfârşitul cozii şi adăgând mutarea pointerul `rear` pe ultimul nod introdus (`nou`), obţinem:



►► PUSH(front, rear, val)

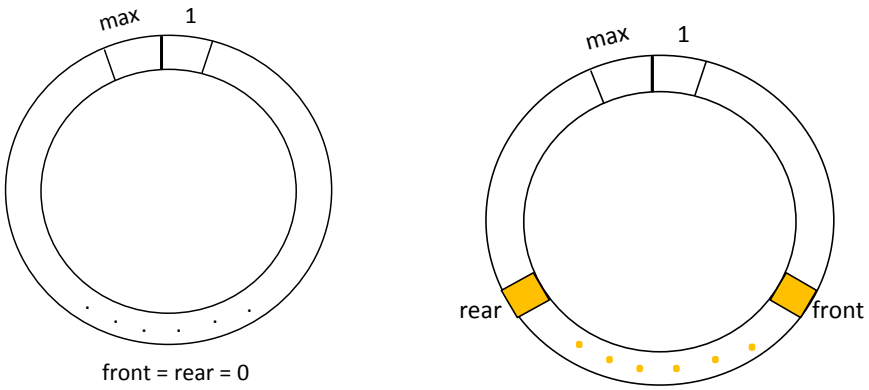
```
1.  creează nod nou
2.  if nou ≠ NIL then
3.    nou->info ← val
4.    nou->next ← NIL
5.    if rear = NIL then
6.      front ← nou
7.    else
8.      rear->next ← nou
9.    endif
10.   rear ← nou
11. else
12.   overflow
13. endif
```

În cazul ştergerii, folosim un pointer auxiliar `temp` către primul element din coadă, pentru a putea elibera memoria alocată nodului de şters (primul). Legătura necesară pentru ştergerea nodului `front` constă în mutarea lui `front` pe al doilea element din coadă (`front->next`) (linia 4). În cazul în care coada conţinea un singur element, prin atribuirea `front ← front->next`, `front` devine **`NIL`** . Atunci, trebuie să modificăm şi adresa reţinută de `rear`, atribuindu-i acestuia **`NIL`** . Şi obţinem `front = rear =` **`NIL`** , adică o coadă vidă.

►► POP(front, rear, x)

```
1.  if front ≠ NIL then
2.    x ← front->info
3.    temp ← front
4.    front ← front->next
5.    distruge temp
6.    if front = NIL then
7.      rear ← NIL
8.    endif
9.  else
10.   underflow
11. endif
```

2.3 Caz particular: Cozi circulare în alocare statică



Imaginea din stânga arată o coadă circulară în alocare statică cu maxim `max` elemente în momentul iniţializării, când este vidă. Imagine din dreapta conţine coada după ce s-au efectuat inserări şi ştergeri.

Iniţial, când coada este vidă vom avea: `front = rear = 0`.

Ca şi în cazul cozilor simple alocate secvenţial, inserarea unui nou element se face la poziţia imediat următoare lui `rear`. Atunci dacă următoarea poziţie este chiar `front`, înseamnă că avem o coadă plină (`rear + 1 = front` mod `(max)`). Dacă un singur element se află în coadă, atunci `rear = front` $\neq$ `0`.

Algoritmul de inserare

►► PUSH(coada, front, rear, val)

```
1.  if rear mod (max) + 1 ≠ front then
2.    if front = 0 then
3.      front = 1
4.    endif
5.    rear ← rear mod max + 1
6.    coada[rear] ← val
7.  else
8.    overflow
9.  endif
```

Algoritmul de ştergere

►► POP(coada, front, rear, x)

```
1.  if front ≠ 0 then
2.    x ← coada[front]
3.    if front = rear then
4.      front ← 0
5.      rear ← 0
6.    else
7.      front ← front mod (max) + 1
8.    endif
9.  else
10.   underflow
11.  endif
```

1. **(2p)** Să se implementeze o stivă de numere întregi, cu următoarele operaţii:
   - **a.** `void push (a, stiva)` care adaugă elementul `a` în vârful stivei;
   - **b.** `int pop (stiva)` care scoate elementul din vârful stivei şi îl returnează ca rezultat al funcţiei;
   - **c.** `int peek(stiva)` care citeşte elementul din vârful stivei, fără a-l scoate;
   - **d.** `bool isEmpty(stiva)` care verifică dacă stiva este vidă sau nu;
   - **e.** `int search(a, stiva)` care întoarce `-1` dacă elementul `a` nu se află în stivă. Dacă `a` apare în stivă, atunci funcţia întoarce distanţa de la vârful stivei până la apariţia cea mai apropiată de vârf. Se va considera că vârful se află la distanţă `0`.
   - **f.** `void print(stiva)` care afişează stiva, pornind de la vârful ei şi continuând spre bază.

2. **(2p)** Dat un şir $w = w_1 w_2 \cdots w_n$ (n număr par) de caractere `'a'` şi `'b'`, să se decidă dacă în şirul w numărul de caractere `'a'` este acelaşi cu numărul de caractere `'b'`. Şirul de intrare se poate parcurge doar o singură dată, iar pentru a decide rezultatul se va folosi o stivă. Nu se permite numărarea aparţiiilor caracterelor `'a'`, `'b'`.

3. **(2p)** Dat un şir $w = w_1 w_2 \cdots w_n$ de caractere `'('` şi `')'`, să se folosească o stivă pentru a decide dacă acest şir este corect parantezat (adică, pentru orice subşir $w_1 \cdots w_i$, cu $i = \overline{1,n}$, avem că numărul de caractere `'('` este mai mare sau egal decât numărul de caractere `')'`). În caz că w nu este parantezat corect, se va indica poziţia primei paranteze `')'` care nu are corespondent.

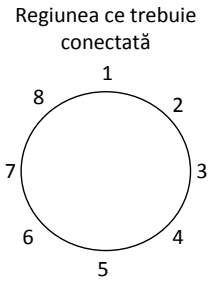
4. **(2p)** Considerăm următoarea problemă (**problema conectării pinilor**): se dă o suprafaţă circulară cu un număr n de pini (ţăruşi) pe margini (numerotaţi de la 1 la n), împreună cu o listă de perechi de pini ce trebuie conectaţi cu fire metalice.

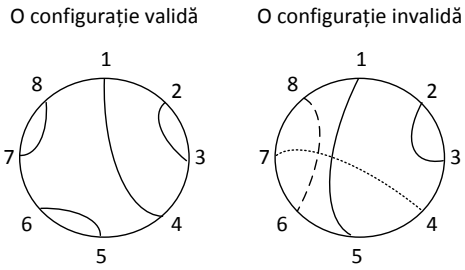
   Problema cere sa determinaţi în timp $O(n)$ dacă pentru o configuraţie ca mai sus, pinii pereche pot fi conectaţi, fără ca acestea să se intersecteze. La intrare se vor citi:

   - n: numărul de pini;
   - `pereche[n]`: un vector de n componente, unde `pereche[i]==pereche[j]`, $1 \leqslant i < j \leqslant n$, dacă pinii i şi j trebuie conectaţi.

   Ex. 1. Pentru $n = 8$ şi vectorul `pereche = (1, 2, 2, 1, 3, 3, 4, 4)` avem configuraţia validă din imagine.

   Ex. 2. Pentru $n = 8$ şi vectorul `pereche = (1, 2, 2, 3, 1, 4, 3, 4)` avem configuraţia invalidă din imagine.





5. **(2p)** Să se implementeze o coadă de numere întregi, cu următoarele operaţii:
   - **a.** `void push (a, coada)` care adaugă elementul `a` în coadă;
   - **b.** `int pop (coada)` care scoate primul elementul din coadă şi îl returnează ca rezultat al funcţiei;
   - **c.** `int peek(coada)` care citeşte primul element din coadă, fără a-l scoate;
   - **d.** `bool isEmpty(coada)` care returnează `true` atunci când coada este vidă şi `false` altfel;

e. `int search(a, coada)` care întoarce `-1` dacă elementul `a` nu se află în coadă. Dacă `a` apare în coadă, atunci funcţia întoarce distanţa de la primul element al cozii până la apariţia cea mai apropiată de primul element al cozii. Se va considera că primul element se află la distanţă `0`.

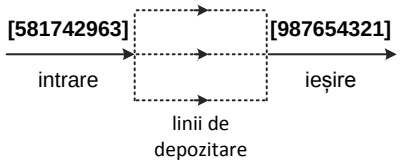
f. `void print(coada)` care afişează coada, pornind de la primul element şi continuând spre ultimul.

6. **(2p)** Spunem că o imagine digitală binară M este o matrice de $m \times m$ elemente (pixeli) 0 sau 1. Un element `a` al matricei este adiacent cu `b`, dacă `b` se alfă deasupra, la dreapta, dedesubtul sau la stânga lui `a` în imaginea M. Spunem că doi pixeli 1 adiacenţi aparţin aceleiaşi componente. Problema va cere să etichetaţi pixelii imaginii astfel încât doi pixeli primesc aceeaşi etichetă dacă şi numai dacă aparţin aceleiaşi componente.

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

7. **(2p)** Un depou feroviar constă dintr-o linie ferată de intrare, k linii auxiliare de depozitare, şi o linie de ieşire. Fiecare linie operează pe un sistem de coadă (FIFO). În plus, vagoanele se pot deplasa doar dinspre linia de intrare spre linia de ieşire. Să se scrie un program care, dat un şir de vagoane pe linia de intrare (numerotate de la 1 la n şi aranjate în orice ordine), descrie o strategie de a obţine pe linia de ieşire şirul de vagoane $n; n-1; \ldots; 2; 1$, folosind liniile de depozitare. În caz că nu există o astfel de strategie, se va afişa acest lucru.



8. Presupunem că avem n persoane numerotate de la 1 la n dispuse pe un cerc şi că eliminăm circular fiecare a doua persoană, până când rămâne o singură persoană. Care este numărul acestei persoane? De exemplu, pentru $n = 10$, vom elimina persoanele: 2, 4, 8, 10, 3, 7, 1, 9 – în această ordine. Supravieţuitorul va fi 5.

a. **(10ps)** Cine este supravieţuitorul pentru $n = 2^{100} + 6$?

Generalizând problema de mai sus, considerăm că eliminăm fiecare a k-a persoană. Cine este supravieţuitorul pentru:

b. **(10ps)** $n = 1000$ şi $k = 7$? Implementarea se va face folosind liste alocate dinamic.

c. **(10ps)** $n = 104857600$ şi $k = 7$?

■ TERMEN DE PREDARE: **Săptămâna 7 (12–16 noiembrie) inclusiv.**

■ DETALII: **Studenţii pot obţine un maxim de 24 puncte. Problemele 1 şi 5 sunt obligatorii. Problemele 2-4, 6–7 sunt suplimentare.Problema 8 este facultativă, iar termenul de predare pentru ea este săptămâna 6 (5–9 noiembrie). Este punctată rezolvarea unei singure probleme dintre 8a, 8b şi 8c. Un singur student poate rezolva problema facultativă.**