

# Laborator 3: Stivă. Coadă. Liste Dublu Înlănțuite

## 1 Obiective

Scopul acestei sesiuni de laborator este de a ne familiariza cu implementarea operațiilor pe tipurile de date abstracte stivă și coadă, respectiv cu implementarea operațiilor pe liste dublu înlănțuite.

## 2 Noțiuni teoretice

### 2.1 Stive

Stiva este o listă cu o politică de acces specială: adăugarea sau ștergerea unui element se face la un singur capăt al listei, numit **vârful stivei**. Elementul introdus primul în stivă poartă numele de **baza stivei**. Stiva se poate asemăna unui vraf de farfurii așezat pe o masă: modalitatea cea mai comodă de a pune o farfurie este în vârful stivei, și tot de aici e cel mai simplu să se ia o farfurie.

Datorită locului unde se acționează asupra stivei, aceste structuri se mai numesc structuri de tip **LIFO (Last In First Out)**, adică *ultimul venit - primul ieșit*. Modelul unei **stive** implementat prin strategia înlănțuită este dat de Figura 1, iar modelul de implementare prin vector este schițat în Figura 2.

Principalele operații pe o stivă sunt următoarele:

**push** – adăugarea unui element în vârful stivei (*insert\_first* – listă înlănțuită, sau *insert\_last* – vector);

**pop** – ștergerea unui element din vârful stivei (*delete\_first* – listă înlănțuită, sau *delete\_last* – vector); operația poate să și returneze elementul sters (i.e. nu îl șterge fizic, doar îl elimină din stivă)

Pe lângă aceste operații se mai poate specifica o operație de inițializare a stivei, respectiv una care doar returnează primul element din stivă, fără a-l șterge (de regulă denumită *top*).

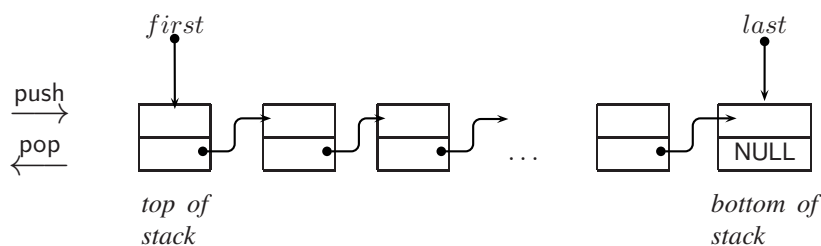


Figura 1: Modelul unei stive, implementată prin listă simplu înlănțuită.

**Ex. 1** — Implementați operațiile fundamentale pe stivă – **void push(STACK\* s, int key)** și **int pop(STACK\* s)**, utilizând lista simplu înlănțuită ca structură de bază (operația *pop* va returna conținutul nodului care a fost șters din stivă). Totodată, implementați o funcție de inițializare a stivei și una de afișare a elementelor acesteia. Testați operațiile implementate.

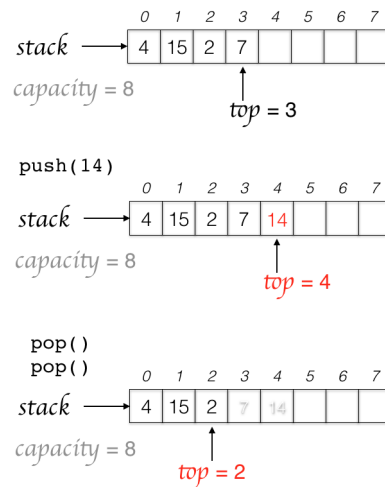


Figura 2: Modelul unei stive, implementată prin vector.

## 2.2 Cozi

Coadă reprezintă o altă categorie specială de listă, în care elementele se adaugă la un capăt (sfârșit) și se sterg de la celălalt capăt (început). Această politică de acces este cunoscută sub numele de **FIFO (First In First Out)**, adică *primul venit - primul servit*.

Modelul intuitiv al acestei structuri este coada care se formează la un magazin: lumea se așează la coadă la sfârșitul ei, cei care se găsesc la începutul cozii sunt serviți, părăsind apoi coada.

Figura 3 prezintă modelul de implementare înlănțuită a unei cozii, respectiv Figura 4 pe cel de vector. Operațiile principale sunt:

**enqueue** – introducerea unui element în coadă – *insert\_last*;

**dequeue** – scoaterea unui element din coadă – *delete\_first*; operația poate să și returneze elementul șters (i.e. nu îl șterge fizic, doar îl elimină din coadă)

Pe lângă aceste operații, se mai poate specifica o operație de inițializare a cozii, respectiv una care doar returnează primul element din coadă, fără a-l șterge (de regulă denumită *front*).

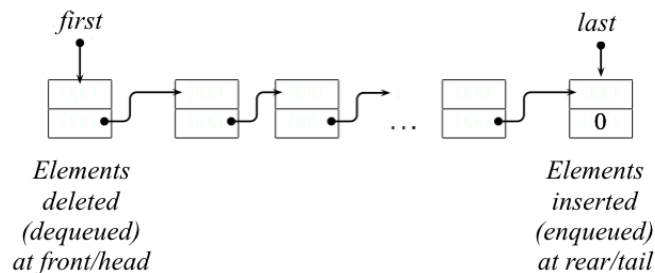


Figura 3: Modelul unei cozii, implementată ca lista simplu înlănțuită.

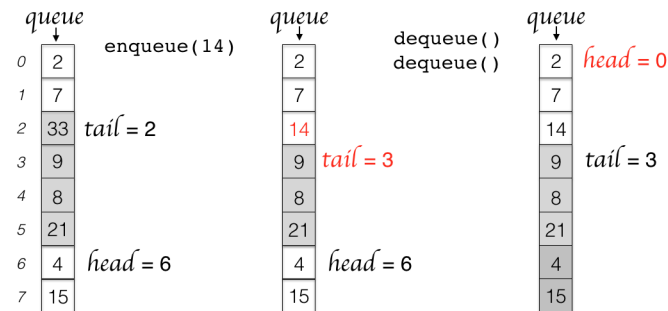


Figura 4: Modelul unei cozi implementate secvențial (cu vector). *tail* reprezintă următoarea poziție pentru inserare, respectiv *head* reprezintă poziția primului element din coadă. Conținutul cozii se află între *head* (inclusiv) și *tail* (exclusiv), interpretate circular. Inițial, coada conține elementele 4, 15, 2, 7; după `enqueue(14)`: 4, 15, 2, 7, 14; după cele două operații de `dequeue()`: 2, 7, 14

**Ex. 2** — Implementați operațiile fundamentale pe coadă – `void enqueue(QUEUE *Q, int key)` și `int dequeue(QUEUE *q)` (împreună cu o funcție de inițializare a cozii și una de afișare a elementelor acesteia), utilizând un **vector** ca structură de bază.

Utilizați exemplele furnizate în Figura 4.

Implementarea voastră ar trebui să considere cazurile de *overflow* la inserare (coada este plină, nu se mai poate insera), respectiv *underflow* la ștergere (coada este goală, nu se poate șterge element). **Sugestie:** Structura *queue* ar trebui să conțină un câmp *capacity*, reprezentând capacitatea cozii, adresa *array* a unui vector care se va alocă dinamic de dimensiunea *capacity*, cei doi indici - *head*, respectiv *tail*, și un câmp *size*.

```
typedef struct
{
    int* array;
    int capacity;
    int size;
    int head, tail;
} QUEUE;
```

## 2.3 Lista dublu înlănțuită

Lista *dublu înlănțuită* este lista dinamică între nodurile căreia s-a definit o dublă relație: de succesor și de predecesor. Modelul unei astfel de liste este dat în figura 5.

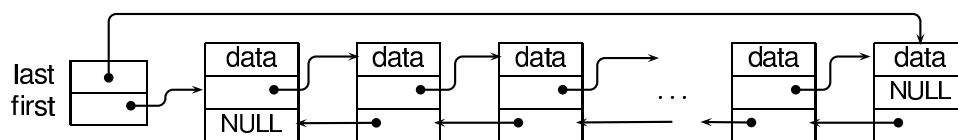


Figura 5: Modelul unei liste dublu înlănțuite.

Structura de nod într-o listă dublu înlănțuită se poate defini astfel:

```
typedef struct _DLL_NODE {
    int key;
    struct _DLL_NODE *next;
    struct _DLL_NODE *prev;
} DLL_NODE;
```

Principalele operații cu liste dublu înlănțuite sunt următoarele:

- inserarea unui nod (la început, la sfârșit, înainte/după un anumit nod);
- ștergerea unui nod (de la început, de la sfârșit, nodul având o anumită cheie),
- căutarea nodului cu o anumită informație/cheie.

Adițional, mai putem specifica operații de creare și ștergere a întregii liste, operație de calculare a dimensiunii listei, etc.

### 2.3.1 Implementarea operațiilor cu liste dublu înlănțuite

Vom identifica lista ca fiind dată de o structură de tip *DL\_LIST*, care conține cei doi pointeri spre începutul și sfârșitul listei (i.e. încapsulăm informația de început/sfârșit al listei într-o structură nouă):

```
typedef struct {
    DLL_NODE *first;
    DLL_NODE *last;
}DL_LIST;
```

Această grupare a celor doi pointeri într-o structură nouă ne permite accesarea acestora printr-o singură variabilă, fiind o modalitate mai elegantă de a specifica lista.

Multumită celor două referințe menținute la nivelul fiecărui nod, o listă dublu înlănțuită poate fi parcursă în două direcții, așa cum arată tabelul 1.

Parcursere secvențial înainte	Parcursere secvențial înapoi
<pre>for ( p = L-&gt;first; p != NULL; p = p-&gt;next ) {     /*perform some operation on current node, p*/ }</pre>	<pre>for ( p = L-&gt;last; p != NULL; p = p-&gt;prev ) {     /*perform some operation on current node, p*/ }</pre>

Tabela 1: Parcurgerile pentru o listă dublu înlănțuită

### Inserarea unui nod într-o listă dublu înlănțuită

Ca la lista simplu înlănțuită, pentru a insera un nod nou în structură se alocă mai întâi spațiu pentru nodul nou și se populează câmpurile de date din nod (setăm și câmpurile *prev* și *next* la valoarea *NULL*).

Din nou, ca la lista simplu înlănțuită, la operația de inserare trebuie să verificăm dacă lista nu este goală; dacă lista este goală, nodul nou va reprezenta atât începutul, cât și sfârșitul listei:

```
if ( L->first == NULL )
{ /* the list is empty */
    L->first = L->last = p;
}
```

*Observație:* Având în vedere că o funcție de inserare poate modifica valoarea lui *L* – fie începutul, fie sfârșitul listei – lista trebuie transmisă prin referință, prin urmare în corpul funcției accesul la câmpurile *first* și *last* se face fie utilizând *L->*, sau *(\*L)*. (i.e. *L->first*, sau *(\*L).first*).

În funcție de poziția din listă unde dorim să inserăm nodul nou, întâlnim situațiile de inserare descrise în tabelul 2:

### Ștergerea unui nod dintr-o listă dublu înlănțuită

Într-o listă dublu înlănțuită se poate șterge primul element, ultimul element, un element care are o cheie dată. Tabelul 3 prezintă primele 2 cazuri, iar pentru ultimul caz se da pseudocodul.

La ștergerea unui nod precizat prin cheia *givenKey*, presupunem că nodul există și are adresa *p* (găsită apelând căutarea după cheie). Se dă mai jos pseudocodul pentru ștergerea nodului *p*, urmând ca implementarea efectivă să fie lăsată ca exercițiu:

```
function DELETE(L, p)
```

Înainte primul nod	După ultimul nod	După un nod de cheie dată <i>afterKey</i> , presupunând că acesta există și are adresa <i>q</i> :
<pre>/* the list is not empty */ p-&gt;next = L-&gt;first; L-&gt;first-&gt;prev = p; L-&gt;first = p;</pre>	<pre>/* the list is not empty */ p-&gt;prev = L-&gt;last; L-&gt;last-&gt;next = p; L-&gt;last = p;</pre>	<pre>p-&gt;prev = q; p-&gt;next = q-&gt;next; if ( q-&gt;next != NULL )     q-&gt;next-&gt;prev = p; q-&gt;next = p; if ( L-&gt;last == q )     L-&gt;last = p;</pre>

Tabela 2: Inserare înainte și după primul nod

Ștergerea primului nod:	Ștergerea ultimului nod:
<pre>p = L-&gt;first; L-&gt;first = L-&gt;first-&gt;next; free( p ); /* free memory */ if ( L-&gt;first == NULL ) L-&gt;last = NULL; else L-&gt;first-&gt;prev = NULL;</pre>	<pre>p = L-&gt;last; L-&gt;last = L-&gt;last-&gt;prev; if ( L-&gt;last == NULL ) L-&gt;first = NULL; else L-&gt;last-&gt;next = NULL; free( p );</pre>

Tabela 3: Ștergerea primului / ultimului nod din lista dublu înlănțuită

```
if p.prev ≠ NIL then
    p.prev.next ← p.next
else
    L.first ← p.next
end if
if p.next ≠ NIL then
    p.next.prev ← p.prev
else
    L.last ← p.prev
end if
free(p)
end function
```

### Căutarea unui element într-o listă dublu înlănțuită

Căutarea unui nod după cheie se face identic ca pentru o listă simplu înlănțuită (vezi Laborator 2).

**Ex. 3** — Implementați următoarele funcții pentru lista dublu înlănțuită:

- Inserare: `void insert_first(DL_LIST *L, int givenKey)`, `void insert_last(DL_LIST *L, int givenKey)` și `void insert_after_key(DL_LIST *L, int afterKey, int givenKey)`.
- Parcurgere: `void print_forward(DL_LIST *L)` și `void print_backward(DL_LIST *L)` care afișează conținutul listei dublu înlănțuite de la primul la ultimul element, respectiv în ordine inversă.
- Căutare: `DLL_NODE * search(DL_LIST *L, int givenKey)` care caută în lista *L* nodul care are cheia *givenKey*. Funcția returnează adresa nodului, respectiv *NULL* dacă acesta nu există.
- Ștergere: `void delete_first(DL_LIST *L)`, `void delete_last(DL_LIST *L)` și `void delete_key(DL_LIST *L, int givenKey)` pentru lista dublu înlănțuită.

## 3 Mersul lucrării

Studiați codul prezentat în laborator și utilizați acest cod pentru rezolvarea exercițiilor obligatorii, prezentate pe parcursul lucrării. La finalul sesiunii de laborator, este obligatoriu ca fiecare student să prezinte codul

(compilabil, executabil) cerut în exercițiile de pe parcursul lucrării de laborator.

### 3.1 Probleme

1. Rezolvați exercițiile obligatorii din laborator - marcate cu chenar gri !
2. Implementați operațiile fundamentale pe stivă (**void push(int \*stack, int \*top, int key)** și (**int pop(int \*stack, int \*top)**), utilizând un vector ca structură de bază.
3. Stiva permite inserarea și stergerea la un singur capăt, pe când coada permite inserarea la un capăt și stergerea de la celălalt capăt. O coadă cu două capete (*en. double-ended queue*) permite inserarea și stergerea de la ambele capete. Implementați cele patru operații de inserare/ștergere pentru o astfel de structură, utilizând un vector ca structură de bază. Operațiile voastre ar trebui să aibă complexitate  $O(1)$ .
4. (\*) Implementați o coada circulară, utilizând un sir (de capacitate dată, constantă) ca și structura de bază, și doi indici - *head* și *tail*. Spre deosebire de Ex. 2 din sarcinile obligatorii, nu aveți voie să utilizați câmp de *size* în structura voastră. Porniți de la pseudocodul disponibil în Th. Cormen et. al, "Introduction to Algorithms" (pag 235, sect 10.1), tratați cazurile de overflow și underflow. Se cere implementarea operațiilor de *init*, *enqueue*, *dequeue* și *print\_queue*.
5. Se da un garaj pentru camioane. Drumul de acces al garajului poate să conțină oricâte camioane. Garajul are o singură ușă astfel încât doar ultimul camion care a intrat poate să iasă primul (conform modelului stivă). Fiecare camion este identificat de un număr întreg pozitiv (*truck\_id*). Scrieți un program care să trateze diferite tipuri de mutări ale camioanelor, și să permită următoarele comenzi:

- (a) *Pe\_drum* (*truck\_id*);
- (b) *Intra\_in\_garaj* (*truck\_id*);
- (c) *Iese\_din\_garaj* (*truck\_id*);
- (d) *Afiseaza\_camioane* (*garaj* sau *drum*);

Dacă se dorește scoaterea unui camion care nu este cel mai aproape de intrarea garajului se va afișa un mesaj de eroare: Camionul *x* nu este la ușa garajului.

6. Se considera problema anterioară a camioanelor dintr-un garaj, dar de această dată garajul are două uși legate printr-un drum circular. O ușă este folosită doar pentru intrarea camioanelor în garaj, iar alta ușă este folosită pentru ieșirea din garaj. Camioanele pot ieși din garaj doar în ordinea în care au intrat (conform modelului coadă).

Date de intrare:

```
Pe_drum(2)
Pe_drum(5)
Pe_drum(10)
Pe_drum(9)
Pe_drum(22)
Afiseaza_camioane(drum)
Afiseaza_camioane(garaj)
Intra_in_garaj(2)
Afiseaza_camioane(drum)
Afiseaza_camioane(garaj)
Intra_in_garaj(10)
Intra_in_garaj(25)
Iese_din_garaj(10)
Iese_din_garaj(2)
Afiseaza_camioane(drum)
Afiseaza_camioane(garaj)
```

Date de iesire:

```
drum:_2_5_10_9_22
garaj:_nimic
drum:_5_10_9_22
garaj:_2
error:_25_nu_este_pe_drum!
error:_10_nu_este_la_iesire!
drum:_2_5_9_22
garaj:_10
```

7. De la tastatură se citesc  $n$  cuvinte; să se creeze o listă dublu înlănțuită, care să conțină în noduri cuvintele distincte și frecvența lor de apariție. Lista va fi ordonată alfabetic. Se vor afișa cuvintele și frecvența lor de apariție a) în ordine alfabetică crescătoare și b) în ordine alfabetică descrescătoare.