

# Structuri de Date

## Laboratorul 3: Stive și Cozi

Dan Novischi

9 martie 2019

### 1. Introducere

Scopul acestui laborator îl reprezintă lucrul cu stive și cozi. Acesta are în vedere următoarele obiective:

- implementarea unei interfețe de lucru pentru stiva bazată pe liste;
- implementarea unei interfețe de lucru pentru coadă bazată pe liste;
- rezolvarea unei probleme simple cu ajutorul interfetei pentru stive;
- rezolvarea unei probleme simple cu ajutorul interfetei pentru coadă.

### 2. Structura Stive/Cozi

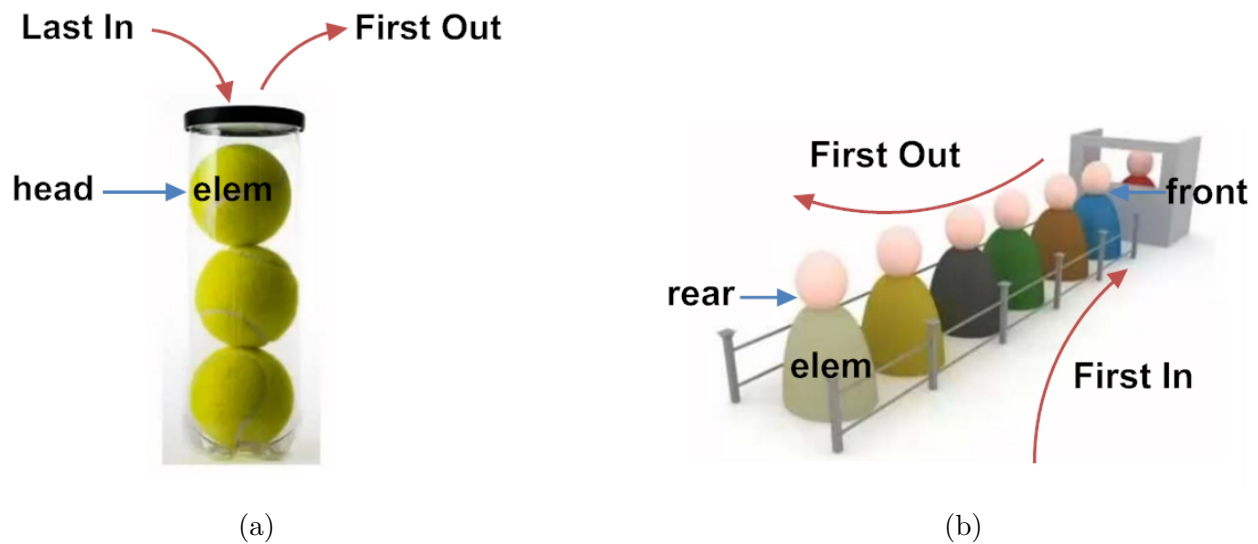


Figura 1: Structură stive și cozi: (a) – *Last In First Out*; (b) – *First In First Out*

În cadrul structurilor de date o stivă este o instanță a unui tip de date abstract ce formalizează conceptul de colecție cu acces restricționat. Restricția respectă regula LIFO (Last In, First Out). Astfel, accesul la elementele stivei se face doar prin vârful acesteia (**head**) după cum se poate observa în Figura 1a. Reprezentarea acesteia în cadrul laboratorului având următoarele definiții:

```

1  typedef struct StackNode{
2      Item elem;
3      struct StackNode *next;
4  }StackNode;

```

```

1  typedef struct Stack{
2      StackNode* head;
3      long size;
4  }Stack;

```

Analog structura de coadă este tot o instanță a unui tip de date abstract. În acest caz structura modelează comportamentul unui buffer de tip FIFO (First In, First Out). Astfel, primul element introdus în coadă va fi și primul care va fi scos din coadă, în timp ce accesul este restricționat doar la primul element (front). Reprezentarea acestei structuri în cadrul laboratorului are următoarele definiții:

```

1  typedef struct QueueNode{
2      Item elem;
3      struct QueueNode *next;
4  }QueueNode;

```

```

1  typedef struct Queue{
2      QueueNode *front;
3      QueueNode *rear;
4      long size;
5  }Queue;

```

### 3. Cerințe

În acest laborator dispuneți de mai multe fișiere inclusiv scheletul de cod după cum urmează:

- **Stack.h** – interfața generică a unei stive, care trebuie implementată conform cerințelor de mai jos.
- **Queue.h** – interfața generică a unei cozi, care trebuie implementată conform cerințelor de mai jos.
- **paranteses.c** – aplicația pentru problema cu stive din cerințele de mai jos.
- **testStack.c** – checker pentru validarea implementării interfetei pentru stive.
- **testQueue.c** – checker pentru validarea implementării interfetei pentru stive.
- **parantheses.c** – aplicația pentru problema din cerințele de mai jos care folosește interfața stivă.
- **input-parantheses.txt** – fișier ce conține input-ul în format text pentru problema de la punctul anterior.

- `radix_sort.c` – aplicatia pentru problema din cerintele de mai jos care foloseste interfața de coadă.
- `input-radix-sort.csv` – fisier ce contine input-ul in format .csv pentru problema de la punctul anterior.
- `Makefile` – fiserul pe baza caruia se vor compila si rula testele (interfata si probleme).

Pentru compilarea tuturor aplicatiilor folositi comanda `"make build"`. Aceasta are urmatorul output pentru un program fără erori de sintaxă sau warning-uri:

```
$ make build
gcc -std=c9x -g -O0 parentheses.c -o parentheses -lm
gcc -std=c9x -g -O0 radix_sort.c -o radix_sort -lm
gcc -std=c9x -g -O0 testStack.c -o testStack -lm
gcc -std=c9x -g -O0 testQueue.c -o testQueue -lm
```

Iar pentru stergerea automata a fisierelor generate prin compilare folositi comanda `"make clean"`:

```
$ make clean
rm -f parentheses radix_sort testStack testQueue
```

Pentru testarea completă (inclusiv memory leaks) puteti folosi:

- `"make test-stack"` pentru interfața de stivă
- `"make test-queue"` pentru interfața de coadă

**Cerința 1 (4p)** In fișierul `Stack.h` implementați funcțiile de interfata ale stivei urmărind atât indicațiile/prototipurile din platforma/schelet cât și ordinea de mai jos:

- a) `createStack` – creează o stivă prin alocare dinamică.
- b) `isEmpty` – verifică dacă o stivă este sau nu goală.
- d) `push` – introduce un nou element (`elem`) în stivă respectând regula LIFO.
- e) `top` – returnează elementul din vârful stivei (`head`).
- f) `pop` – extrage un element din stivă respectând regula LIFO.
- g) `destroyStack` – distruge stiva.

Pentru validarea completă a corectitudinii implementării folosiți comanda `make test-stack`. În cazul unei implementări corecte a interfetei acesta generează următorul output:

```
$ make test-stack
valgrind --leak-check=full ./testStack
==4275== Memcheck, a memory error detector
==4275== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4275== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4275== Command: ./testStack
==4275==
. Testul Create a fost trecut cu succes! Puncte: 0.03
. Testul IsStackEmpty a fost trecut cu succes! Puncte: 0.02
. Testul Push      a fost trecut cu succes! Puncte: 0.10
. Testul Top       a fost trecut cu succes! Puncte: 0.05
. Testul Pop       a fost trecut cu succes! Puncte: 0.10
.  *Destroy se va verifica cu valgrind* Puncte: 0.10.

Scor total: 0.40 / 0.40

==4275==
==4275== HEAP SUMMARY:
==4275==    in use at exit: 0 bytes in 0 blocks
==4275== total heap usage: 15 allocs, 15 frees, 1,248 bytes allocated
==4275==
==4275== All heap blocks were freed -- no leaks are possible
==4275==
==4275== For counts of detected and suppressed errors, rerun with: -v
==4275== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

**Cerința 2 (4p)** In fișierul `Queue.h` implementați funcțiile de interfata ale cozii urmărind atât indicațiile/prototipurile din platforma/schelet cât și ordinea de mai jos:

- a) `createQueue` – creează o coadă prin alocare dinamică.
- b) `isEmpty` – verifică dacă o coadă este sau nu goală.
- d) `enqueue` – introduce un nou element (`elem`) în coadă respectând regula FIFO.  
**Atentie:** elementele se introduc pe la **rear**!
- e) `front` – returnează valoarea primului element din coadă.

f) `dequeue` – extrage un element din coadă respectând regula FIFO.

**Atentie:** elementele se extrag de la **front**!

g) `destroyQueue` – distruge coada.

Pentru validarea completă a corectitudinii implementării folosiți comanda `make test-queue`.  
În cazul unei implementări corecte a interfetei acesta generează următorul output:

```
$ make test-queue
valgrind --leak-check=full ./testQueue
==4505== Memcheck, a memory error detector
==4505== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4505== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4505== Command: ./testQueue
==4505==
. Testul Create a fost trecut cu succes! Puncte: 0.03
. Testul IsQueueEmpty a fost trecut cu succes! Puncte: 0.02
. Testul Enqueue a fost trecut cu succes! Puncte: 0.10
. Testul Front a fost trecut cu succes! Puncte: 0.05
. Testul Dequeue a fost trecut cu succes! Puncte: 0.10
. *Destroy se va verifica cu valgrind* Puncte: 0.10.

Scor total: 0.40 / 0.40

==4505==
==4505== HEAP SUMMARY:
==4505==    in use at exit: 0 bytes in 0 blocks
==4505== total heap usage: 15 allocs, 15 frees, 1,256 bytes allocated
==4505==
==4505== All heap blocks were freed -- no leaks are possible
==4505==
==4505== For counts of detected and suppressed errors, rerun with: -v
==4505== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

**Cerința 3 (1p)** În fișierul `parentheses.c` implementați funcția `isBalanced` cu ajutorul interfetei de stivă. Funcția determină dacă un șir de caractere format numai din paranteze deschise și închise este balansat sau nu. Un șir de paranteze este balansat dacă fiecare paranteză deschisă "(" are asociată o paranteză închisă ")".

**Observație:** În cadrul acestei probleme consultați fișierul `input-parentheses.txt` și clarificați eventualele neclarități cu asistentul.

Pentru validarea implementării rulați programul folosind comanda `./parentheses`. O implementare corectă a soluției va genera următorul output:

```
$ ./parentheses
((())) ---> is balanced.
((() ) ---> not balanced.
(( )) ---> not balanced.
(( ))( )(( ))( ) ---> is balanced.
( )(( ))(( ))( )(( )) ---> not balanced.
( )(( ))(( ))( )(( )) ---> not balanced.
( )(( ))(( ))(( )) ---> not balanced.
(( ))( )(( ))(( ))( ) ---> is balanced.
(( ))( )(( ))(( ))(( )) ---> is balanced.
```

**Cerința 4 (1p)** Radix Sort este un algoritm de sortare care ține cont faptul că un șir de numere poate fi sortat prin sortarea succesivă a acestora după cifrele (digits) individuale ale elementelor (i.e. counting). Aceste elemente pot fi nu doar numere, ci orice altceva ce

se poate reprezenta prin întregi. Majoritatea calculatoarelor digitale reprezintă datele în memorie sub formă de numere binare, astfel că procesarea cifrelor din această reprezentare se dovedește a fi cea mai convenabilă. Există două tipuri de astfel de sortare: LSD (Least Significant Digit) și MSD (Most Significant Digit). LSD procesează reprezentările dinspre cea mai puțin semnificativă cifră spre cea mai semnificativă, iar MSD invers.

O versiune simplă a algoritmului *radix sort* este cea care folosește 10 cozi (câte una pentru fiecare cifră de la 0 la 9). În fisierul `radix_sort.c` implemențati funcția `radixSort` urmărind pașii de mai jos (și indicațiile din fisier):

1. Determina numărul maxim de cifre pentru sirul de numerele furnizate ca input.
2. Iterazează peste numărul maxim de cifre
  - (a) Memorează succesiv numerele cu cifra corespunzătoare în coada aferentă.
  - (b) Reconstruiește sirul de numere (parțial sortat) folosind cozile rezultate în pasul anterior.

### Observație:

- în cadrul acestei probleme consultați fisierul `input-radix-sort.csv` și clarificați eventualele neclarități cu asistentul.
- va puteți folosi de funcțiile ajutoare furnizate în fisierul `radix_sort.c`

Pentru validarea implementării rulați programul folosind comanda `./radix_sort`. O implementare corectă a soluției va genera următorul output:

```
$ ./radix_sort
Input array 1: 0 32 1 499 657 2 12 80 27 512 45 20 32 15 54 9 41 8
Sorted array 1: 0 1 2 8 9 12 15 20 27 32 32 41 45 54 80 499 512 657

Input array 2: 51023 4341 1224 190 564 76 23 478 589 5 23 67 20123
Sorted array 2: 5 23 23 67 76 190 478 564 589 1224 4341 20123 51023

Input array 3: 5689 237 2347 789 0 1289 78 5 23 4 712 7238 897 453
Sorted array 3: 0 4 5 23 78 237 453 712 789 897 1289 2347 5689 7238
```