Structuri de Date
Laboratorul 4: Stive și Cozi

Dan Novischi

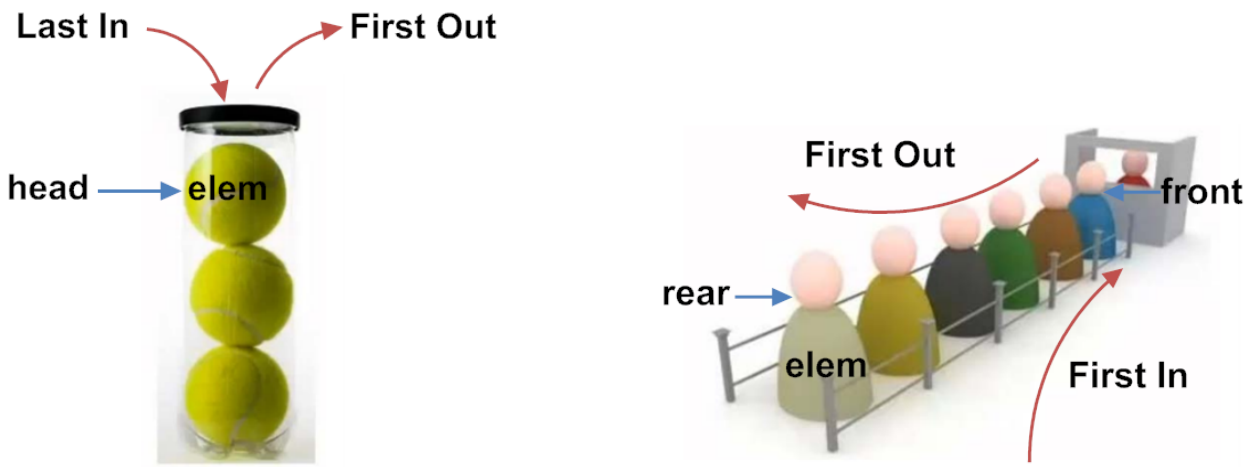
March 9, 2020

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 interfeței pentru stive;
- rezolvarea unei probleme simple cu ajutorul interfeței petru coadă.

2. Structura Stive/Cozi



(a) (b)

Figure 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 colecţie 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 1 (a). Reprezentarea acesteia în cadrul laboratorului având următoarele definiții:

```
typedef struct StackNode{
  Item elem;
  struct StackNode *next;
}StackNode;
```

```
typedef struct Stack{
  StackNode* head;
  long size;
}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). Reprezezentarea acestei structuri în cadrul laboratorului are următoarele definiții:

```
typedef struct QueueNode{
  Item elem;
  struct QueueNode *next;
}QueueNode;
```

```
typedef struct Queue{
        QueueNode *front;
        QueueNode *rear;
        long size;
}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 cerințelelor de mai jos.
- `Queue.h` – interfața generică a unei cozi, care trebuie implementată conform cerințelelor de mai jos.
- `paranteses.c` – apilicația pentru problema cu stive din cerințele de mai jos.
- `testStack.c` – checker pentru validarea implementării interfeței pentru stive.
- `testQueue.c` – checker pentru validarea implementării interfeței pentru stive.
- `parantheses.c` – apilicația 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.

- `Makefile` – fișierul pe baza căruia se vor compila și rula testele (interfața și probleme).

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

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

Iar pentru ștergerea automată a fișierelor generate prin compilare folosiți comanda "`make clean`":

```
$ make clean
rm -f parantheses testStack testQueue
```

Pentru testarea completă (inclusiv memory leaks) puteți folosi:

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

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

a) `createStack` – creează o stivă prin alocare dinamică.

b) `isStackEmpty` – 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 interfeței 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) În fișierul `Queue.h` implementați funcțiile de interfață ale cozii urmărind atât indicațiile/prototipurile din platformă/schelet cât și ordinea de mai jos:

a) `createQueue` – creează o coadă prin alocare dinamică.

b) `isQueueEmpty` – verifică dacă o coadă este sau nu goală.

d) `enqueue` – introduce un nou element (`elem`) în coadă respectând regula FIFO.

Atenție: elementele se introduc pe la `rear`!

e) `front` – returnează valoarea primului element din coadă.

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

Atenție: 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 interfeței 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 (2p) În fișierul `parantheses.c` implementați funcția `isBalanced` cu ajutorul interfeței 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 inchisă ")".

Observație: În cadrul acestei probleme consultați fisierul `input-parantheses.txt` și clarificați eventualele neclarități cu asistentul.
Pentru validarea implementării rulați programul folosind comanda "`./parantheses `". O implementare corectă a soluției va genera următorul output:

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