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 sterge 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 sterge (de regulă denumită *top*).

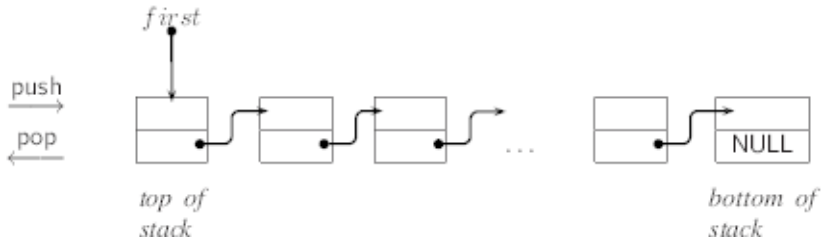

Figura 1: Modelul unei stive, implementată prin listă simplu înlănțuită.

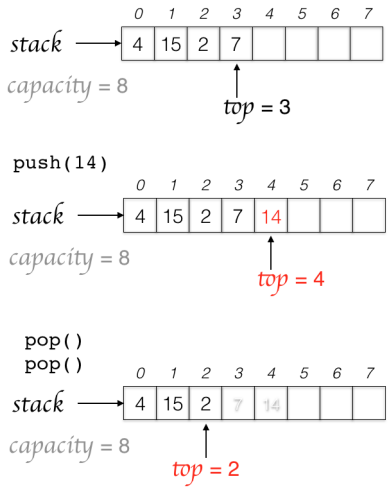

Figura 2: Modelul unei stive, implementată prin vector.

Ex. 1 — Implementați operatiile 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.

```
typedef struct _STACK_NODE {
    int key;
    struct _STACK_NODE *next;
} STACK_NODE;

typedef struct {
    STACK_NODE *first;
}STACK;
```

2.2 Cozi

Coada reprezinta 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 cozi, 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*).

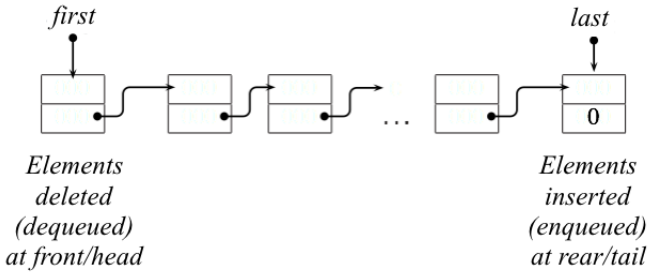

Figura 3: Modelul unei cozi, implementată ca lista simplu înlănțuită.

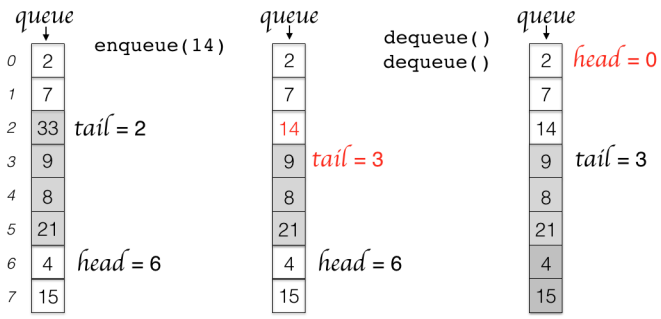

Figura 4: Modelul unei cozi implementate secvențial (cu vector). *tail* reprezintă următoarea poziție pentru inserare, respectiv *head* reprezintă poziția primului element din coadă. Conținutul cozii se află între *head* (inclusiv) și *tail* (exclusiv), interpretate circular. Inițial, coada conține elementele 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)** si **int dequeue(QUEUE *q)** (împreună cu o funcție de inițializare a cozii și una de afișare a elementelor acesteia), utilizand un **vector** ca structură de bază.
Utilizati 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 aloca dinamic de dimesiunea *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 înlănţuită

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

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

```
typedef struct _DLL_NODE {
    int key;
```

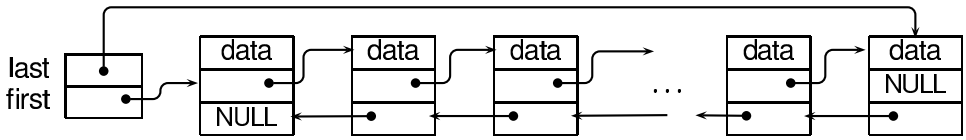

Figura 5: Modelul unei liste dublu înlănţuite.

```
    struct _DLL_NODE *next;
    struct _DLL_NODE *prev;
} DLL_NODE;
```

Principalele operaţii cu liste dublu înlănţuite sunt următoarele:

- inserarea unui nod (la început, la sfârșit, înainte/după un anumit nod);
- ştergerea 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 stergere a întregii liste, operație de calculare a dimensiunii listei, etc.

2.3.1 Implementarea operaţiilor cu liste dublu înlănţuite

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 înlănţuită poate fi parcursă în două direcții, așa cum arată tabelul 1.

Parcurgere secvenţial înainte	Parcurgere secvenţial înapoi

`for ( p = L->first; p != NULL; p = p->next )`
`{`
`  /*perform some operation on current node, p*/`
`}`
	`for ( p = L->last; p != NULL; p = p->prev )`
	`{`
	`  /*perform some operation on current node, p*/`
	`}`

Tabela 1: Parcurgerile pentru o listă dublu înlănțuită

Inserarea unui nod într-o listă dublu înlănţuită

Ca la lista simplu înlănțuită, pentru a insera un nod nou în structură se alocă mai întâi spaţiu pentru nodul nou şi 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 inceputul, fie sfarsitul listei – lista trebuie transmisa prin referinta, prin urmare in corpul functiei accesul la campurile $first$ si $last$ se face fie utilizand *L->*, sau *(*L).* (i.e. *L->first*, sau *(*L).first*).

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

Înaintea primului nod	După ultimul nod	După un nod de cheie dată $afterKey$, presupunând că acesta există şi are adresa q:

`/* the list is not empty */`
`p->next = L->first;`
`L->first->prev = p;`
`L->first = p;`
	`/* the list is not empty */`
	`p->prev = L->last;`
	`L->last->next = p;`
	`L->last = p;`
		`p->prev = q;`
		`p->next = q->next;`
		`if ( q->next != NULL )`
		`  q->next->prev = p;`
		`q->next = p;`
		`if ( L->last == q )`
		`  L->last = p;`

Tabela 2: Inserare înainte și după primul nod

Ştergerea unui nod dintr-o listă dublu înlănţuită

Într-o lista 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.

Ştergerea primului nod:	Ştergerea ultimului nod:

`p = L->first;`
`L->first = L->first->next;`
`free( p ); /* free memory */`
`if ( L->first == NULL )  L->last == NULL;`
`else L->first->prev = NULL;`
	`p = L->last;`
	`L->last = L->last->prev;`
	`if ( L->last == NULL ) L->first = NULL;`
	`else L->last->next = NULL;`
	`free( p );`

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

La ștergerea unui nod precizat prin cheia $givenKey$, presupunem că nodul există şi 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)
  if $p.prev \neq NIL$ **then**
    $p.prev.next \leftarrow p.next$
  else
    $L.first \leftarrow p.next$
  end if
  if $p.next \neq NIL$ **then**
    $p.next.prev \leftarrow p.prev$
  else
    $L.last \leftarrow p.prev$
  end if
  $free(p)$
end function

Căutarea unui element într-o listă dublu înlănţuită

Căutarea unui nod dupa cheie se face identic ca pentru o lista simplu înlănţuită (vezi Laborator 2).

Ex. 3 — Implementati urmatoarele functii pentru lista dublu inlantuita:

- 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)** si **void print_backward(DL_LIST *L)** care afiseaza continutul listei dublu inlantuite de la primul la ultimul element, respectiv in ordine inversa.
- Cautare: **DLL_NODE * search(DL_LIST *L, int givenKey)** care cauta in lista $*L$ nodul care are cheia $givenKey$. Functia returneaza adresa nodului, respectiv $NULL$ daca acesta nu exista.
- Stergere: **void delete_first(DL_LIST *L)**, **void delete_last(DL_LIST *L)** si **void delete_key(DL_LIST *L, int givenKey)** pentru lista dublu inlantuita.

3 Mersul lucrării

Studiați codul prezentat în laborator și utilizati 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. Implementati operațiile fundamentale pe stivă (**void push(int *stack, int *top, int key)** si (**int pop(int *stack, int *top)**), utilizand 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 ștergerea 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. (*) Implementati o coada circulara, utilizand un sir (de capacitate data, constanta) ca si structura de baza, si doi indici - *head* si *tail*. Spre deosebire de Ex. 2 din sarcinile obligatorii, nu aveti voie sa utilizati camp de *size* in structura voastra. Porniti de la pseudocodul disponibil in Th. Cormen et. al, "Introduction to Algorithms" (pag 235, sect 10.1), tratati cazurile de overflow si underflow. Se cere implementarea operatiilor de *init*, *enqueue*, *dequeue* si *print_queue*.
5. Se da un garaj pentru camioane. Drumul de access al garajului poate sa contina oricâte camioane. Garajul are o singura usa astfel încât doar ultimul camion care a intrat pote sa iasa primul (conform modelului stiva). Fiecare camion este identificat de un numar întreg pozitiv (truck_id). Scrieti un program care sa trateze diferite tipuri de mutari ale camioanelor, si sa permita urmatoarele comenzi:

   (a) `Pe_drum` (truck_id);

   (b) `Intra_in_garaj` (truck_ id);

   (c) `Iese_din_garaj` (truck_id);

   (d) `Afiseaza_camioane` (`garaj` sau `drum`);

   Daca se doreste scoaterea unui camion care nu este cel mai aproape de intrarea garajului se va afisa un mesaj de eroare `Camionul` x `nu este la usa garajului`.

6. Se considera problema anterioara a camioanelor dintr-un garaj, dar de aceasta data garajul are doua usi legate printr-un drum circular. O usa este folosita doar pentru intrarea camioanelor în garaj, iar alta usa este folosita pentru iesirea din garaj. Camioanele pot iesi din garaj doar în ordinea în care au intrat (conform modelului coada).

   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 înlănţuită, care să conţină în noduri cuvintele distincte şi frecvenţa lor de apariţie. Lista va fi ordonată alfabetic. Se vor afişa cuvintele şi frecvenţa lor de apariţie a) în ordine alfabetică crescătoare şi b) în ordine alfabetică descrescătoare.